

Software Engineering

TENTH EDITION

Ian Sommerville



ALWAYS LEARNING

PEARSON



SOFTWARE ENGINEERING

Tenth Edition

Ian Sommerville

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor in Chief: Michael Hirsch
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Chelsea Bell
Assistant Acquisitions Editor, Global Edition: Murchana Borthakur
Associate Project Editor, Global Edition: Binita Roy
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd
Director of Marketing: Margaret Waples
Marketing Coordinator: Kathryn Ferranti
Senior Manufacturing Buyer: Carol Melville
Senior Manufacturing Controller, Production, Global Edition: Trudy Kimber
Text Designer: Susan Raymond
Cover Art Designer: Lumina Datamatics
Cover Image: © Andrey Bayda/Shutterstock
Interior Chapter Opener: © graficart.net/Alamy
Full-Service Project Management: Rashmi Tickyani, Aptara®, Inc.
Composition and Illustrations: Aptara®, Inc.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2016

The rights of Ian Sommerville to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Software Engineering, 10th edition, ISBN 978-0-13-394303-0, by Ian Sommerville, published by Pearson Education © 2016.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN 10: 1-292-09613-6
ISBN 13: 978-1-292-09613-1

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

Typeset in 9 New Aster LT Std by Aptara®, Inc.

Printed and bound by Courier Westford in the United States of America.



PREFACE

Progress in software engineering over the last 50 years has been astonishing. Our societies could not function without large professional software systems. National utilities and infrastructure—energy, communications and transport—all rely on complex and mostly reliable computer systems. Software has allowed us to explore space and to create the World Wide Web—the most significant information system in the history of mankind. Smartphones and tablets are ubiquitous and an entire 'apps industry' developing software for these devices has emerged in the past few years.

Humanity is now facing a demanding set of challenges—climate change and extreme weather, declining natural resources, an increasing world population to be fed and housed, international terrorism, and the need to help elderly people lead satisfying and fulfilled lives. We need new technologies to help us address these challenges and, for sure, software will have a central role in these technologies. Software engineering is, therefore, critically important for our future on this planet. We have to continue to educate software engineers and develop the discipline so that we meet the demand for more software and create the increasingly complex future systems that we need.

Of course, there are still problems with software projects. Systems are still sometimes delivered late and cost more than expected. We are creating increasingly complex software systems of systems and we should not be surprised that we encounter difficulties along the way. However, we should not let these problems conceal the real successes in software engineering and the impressive software engineering methods and technologies that have been developed.

This book, in different editions, has now been around for over 30 years and this edition is based around the essential principles that were established in the first edition:

1. I write about software engineering as it is practiced in industry, without taking an evangelical position on particular approaches such as agile development or formal methods. In reality, industry mixes techniques such as agile and plan-based development and this is reflected in the book.

2. I write about what I know and understand. I have had many suggestions for additional topics that might be covered in more detail such as open source development, the use of the UML and mobile software engineering. But I don't really know enough about these areas. My own work has been in system dependability and in systems engineering and this is reflected in my selection of advanced topics for the book.

I believe that the key issues for modern software engineering are managing complexity, integrating agility with other methods and ensuring that our systems are secure and resilient. These issues have been the driver for the changes and additions in this new edition of my book.

Changes from the 9th edition

In summary, the major updates and additions in this book from the 9th edition are:

- I have extensively updated the chapter on agile software engineering, with new material on Scrum. I have updated other chapters as required to reflect the increasing use of agile methods of software engineering.
- I have added new chapters on resilience engineering, systems engineering, and systems of systems.
- I have completely reorganized three chapters covering reliability, safety, and security.
- I have added new material on RESTful services to the chapter covering service-oriented software engineering.
- I have revised and updated the chapter on configuration management with new material on distributed version control systems.
- I have moved chapters on aspect-oriented software engineering and process improvement from the print version of the book to the web site.
- New supplementary material has been added to the web site, including a set of supporting videos. I have explained key topics on video and recommended related YouTube videos.

The 4-part structure of the book, introduced in earlier editions, has been retained but I have made significant changes in each part of the book.

1. In Part 1, Introduction to software engineering, I have completely rewritten Chapter 3 (agile methods) and updated this to reflect the increasing use of Scrum. A new case study on a digital learning environment has been added to Chapter 1 and is used in a number of chapters. Legacy systems are covered in more detail in Chapter 9. Minor changes and updates have been made to all other chapters.

2. Part 2, which covers dependable systems, has been revised and restructured. Rather than an activity-oriented approach where information on safety, security and reliability is spread over several chapters, I have reorganized this so that each topic has a chapter in its own right. This makes it easier to cover a single topic, such as security, as part of a more general course. I have added a completely new chapter on resilience engineering which covers cybersecurity, organizational resilience, and resilient systems design.

3. In Part 3, I have added new chapters on systems engineering and systems of systems and have extensively revised the material on service-oriented systems engineering to reflect the increasing use of RESTful services. The chapter on aspect-oriented software engineering has been deleted from the print version but remains available as a web chapter.

4. In Part 4, I have updated the material on configuration management to reflect the increasing use of distributed version control tools such as Git. The chapter on process improvement has been deleted from the print version but remains available as a web chapter.

An important change in the supplementary material for the book is the addition of video recommendations in all chapters. I have made over 40 videos on a range of topics that are available on my YouTube channel and linked from the book's web pages. In cases where I have not made videos, I have recommended YouTube videos that may be useful.

I explain the rationale behind the changes that I've made in this short video:

http://software-engineering-book/videos/10th-edition-changes

Readership

The book is primarily aimed at university and college students taking introductory and advanced courses in software and systems engineering. I assume that readers understand the basics of programming and fundamental data structures.

Software engineers in industry may find the book useful as general reading and to update their knowledge on topics such as software reuse, architectural design, dependability and security and systems engineering.

Using the book in software engineering courses

I have designed the book so that it can be used in three different types of software engineering course:

1. *General introductory courses in software engineering.* The first part of the book has been designed to support a 1-semester course in introductory software engineering. There are 9 chapters that cover fundamental topics in software engineering.

If your course has a practical component, management chapters in Part 4 may be substituted for some of these.

2. *Introductory or intermediate courses on specific software engineering topics.* You can create a range of more advanced courses using the chapters in parts 2–4. For example, I have taught a course in critical systems using the chapters in Part 2 plus chapters on systems engineering and quality management. In a course covering software-intensive systems engineering, I used chapters on systems engineering, requirements engineering, systems of systems, distributed software engineering, embedded software, project management and project planning.
3. *More advanced courses in specific software engineering topics.* In this case, the chapters in the book form a foundation for the course. These are then supplemented with further reading that explores the topic in more detail. For example, a course on software reuse could be based around Chapters 15–18.

Instructors may access additional teaching support material from Pearson's website. Some of this is password-protected and instructors using the book for teaching can obtain a password by registering at the Pearson website. The material available includes:

- Model answers to selected end of chapter exercises.
- Quiz questions and answers for each chapter.

You can access this material at:

www.pearsonglobaleditions.com/Sommerville

Book website

This book has been designed as a hybrid print/web text in which core information in the printed edition is linked to supplementary material on the web. Several chapters include specially written 'web sections' that add to the information in that chapter. There are also six 'web chapters' on topics that I have not covered in the print version of the book.

You can download a wide range of supporting material from the book's website (software-engineering-book.com) including:

- A set of videos where I cover a range of software engineering topics. I also recommend other YouTube videos that can support learning.
- An instructor's guide that gives advice on how to use the book in teaching different courses.
- Further information on the book's case studies (insulin pump, mental health care system, wilderness weather system, digital learning system), as well other case studies, such as the failure of the Ariane 5 launcher.

- Six web chapters covering process improvement, formal methods, interaction design, application architectures, documentation and aspect-oriented development.
- Web sections that add to the content presented in each chapter. These web sections are linked from breakout boxes in each chapter.
- PowerPoint presentations for all of the chapters in the book and additional PowerPoint presentations covering a range of systems engineering topics are available at pearsonglobaleditions.com/Sommerville.

In response to requests from users of the book, I have published a complete requirements specification for one of the system case studies on the book's web site. It is difficult for students to get access to such documents and so understand their structure and complexity. To avoid confidentiality issues, I have re-engineered the requirements document from a real system so there are no restrictions on its use.

Contact details

Website: software-engineering-book.com
Email: name: software.engineering.book; domain: gmail.com
Blog: iansommerville.com/systems-software-and-technology
YouTube: youtube.com/user/SoftwareEngBook
Facebook: facebook.com/sommerville.software.engineering
Twitter: @SoftwareEngBook or @iansommerville (for more general tweets)

Follow me on Twitter or Facebook to get updates on new material and comments on software and systems engineering.

Acknowledgements

A large number of people have contributed over the years to the evolution of this book and I'd like to thank everyone (reviewers, students and book users) who have commented on previous editions and made constructive suggestions for change. I'd particularly like to thank my family, Anne, Ali, and Jane, for their love, help and support while I was working on this book (and all of the previous editions).

Ian Sommerville,
September 2014

Contents at a glance

	Preface	3
Part 1	Introduction to Software Engineering	15
Chapter 1	Introduction	17
Chapter 2	Software processes	43
Chapter 3	Agile software development	72
Chapter 4	Requirements engineering	101
Chapter 5	System modeling	138
Chapter 6	Architectural design	167
Chapter 7	Design and implementation	196
Chapter 8	Software testing	226
Chapter 9	Software evolution	255
Part 2	System Dependability and Security	283
Chapter 10	Dependable systems	285
Chapter 11	Reliability engineering	306
Chapter 12	Safety engineering	339
Chapter 13	Security engineering	373
Chapter 14	Resilience engineering	408
Part 3	Advanced Software Engineering	435
Chapter 15	Software reuse	437
Chapter 16	Component-based software engineering	464
Chapter 17	Distributed software engineering	490
Chapter 18	Service-oriented software engineering	520
Chapter 19	Systems engineering	551
Chapter 20	Systems of systems	580
Chapter 21	Real-time software engineering	610
Part 4	Software Management	639
Chapter 22	Project management	641
Chapter 23	Project planning	667
Chapter 24	Quality management	700
Chapter 25	Configuration management	730
	Glossary	757
	Subject index	777
	Author index	803

Pearson wishes to thank and acknowledge the following people for their work on the Global Edition:

Contributor

Sherif G. Aly, The American University in Cairo
Muthuraj M., Android developer

Reviewers

Mohit P. Tahiliani, National Institute of Technology Karnataka, Surathkal
Chitra Dhawale, P. R. Patil Group of Educational Institutes, Amravati
Sanjeevni Shantaiya, Disha Institute of Management & Technology



CONTENTS

Preface 3

Part 1 Introduction to Software Engineering 15

Chapter 1 Introduction 17

1.1 Professional software development 19
1.2 Software engineering ethics 28
1.3 Case studies 31

Chapter 2 Software processes 43

2.1 Software process models 45
2.2 Process activities 54
2.3 Coping with change 61
2.4 Process improvement 65

Chapter 3 Agile software development 72

3.1 Agile methods 75
3.2 Agile development techniques 77
3.3 Agile project management 84
3.4 Scaling agile methods 88

Chapter 4 Requirements engineering 101

4.1 Functional and non-functional requirements 105
4.2 Requirements engineering processes 111
4.3 Requirements elicitation 112
4.4 Requirements specification 120
4.5 Requirements validation 129
4.6 Requirements change 130

Chapter 5 System modeling 138

5.1 Context models 141
5.2 Interaction models 144
5.3 Structural models 149
5.4 Behavioral models 154
5.5 Model-driven architecture 159

Chapter 6 Architectural design 167

6.1 Architectural design decisions 171
6.2 Architectural views 173
6.3 Architectural patterns 175
6.4 Application architectures 184

Chapter 7 Design and implementation 196

7.1 Object-oriented design using the UML 198
7.2 Design patterns 209
7.3 Implementation issues 212
7.4 Open-source development 219

Chapter 8 Software testing 226

8.1 Development testing 231
8.2 Test-driven development 242

8.3 Release testing 245
8.4 User testing 249

Chapter 9 Software evolution 255

9.1 Evolution processes 258
9.2 Legacy systems 261
9.3 Software maintenance 270

Part 2 System Dependability and Security 283

Chapter 10 Dependable systems 285

10.1 Dependability properties 288
10.2 Sociotechnical systems 291
10.3 Redundancy and diversity 295
10.4 Dependable processes 297
10.5 Formal methods and dependability 299

Chapter 11 Reliability engineering 306

11.1 Availability and reliability 309
11.2 Reliability requirements 312
11.3 Fault-tolerant architectures 318
11.4 Programming for reliability 325
11.5 Reliability measurement 331

Chapter 12 Safety engineering 339

12.1 Safety-critical systems 341
12.2 Safety requirements 344
12.3 Safety engineering processes 352
12.4 Safety cases 361

Chapter 13 Security engineering 373

13.1 Security and dependability 376
13.2 Security and organizations 380
13.3 Security requirements 382
13.4 Secure systems design 388
13.5 Security testing and assurance 402

Chapter 14 Resilience engineering 408

14.1 Cybersecurity 412
14.2 Sociotechnical resilience 416
14.3 Resilient systems design 424

Part 3 Advanced Software Engineering 435

Chapter 15 Software reuse 437

15.1 The reuse landscape 440
15.2 Application frameworks 443
15.3 Software product lines 446
15.4 Application system reuse 453

Chapter 16 Component-based software engineering 464

16.1 Components and component models 467
16.2 CBSE processes 473
16.3 Component composition 480

Chapter 17 Distributed software engineering 490

17.1 Distributed systems 492
17.2 Client–server computing 499

17.3 Architectural patterns for distributed systems 501
17.4 Software as a service 512

Chapter 18 Service-oriented software engineering 520

18.1 Service-oriented architecture 524
18.2 RESTful services 529
18.3 Service engineering 533
18.4 Service composition 541

Chapter 19 Systems engineering 551

19.1 Sociotechnical systems 556
19.2 Conceptual design 563
19.3 System procurement 566
19.4 System development 570
19.5 System operation and evolution 574

Chapter 20 Systems of systems 580

20.1 System complexity 584
20.2 Systems of systems classification 587
20.3 Reductionism and complex systems 590
20.4 Systems of systems engineering 593
20.5 Systems of systems architecture 599

Chapter 21 Real-time software engineering 610

21.1 Embedded system design 613
21.2 Architectural patterns for real-time software 620
21.3 Timing analysis 626
21.4 Real-time operating systems 631

Part 4 Software Management 639

Chapter 22 Project management 641

22.1 Risk management 644
22.2 Managing people 652
22.3 Teamwork 656

Chapter 23 Project planning 667

23.1 Software pricing 670
23.2 Plan-driven development 672
23.3 Project scheduling 675
23.4 Agile planning 680
23.5 Estimation techniques 682
23.6 COCOMO cost modeling 686

Chapter 24 Quality management 700

24.1 Software quality 703
24.2 Software standards 706
24.3 Reviews and inspections 710
24.4 Quality management and agile development 714
24.5 Software measurement 716

Chapter 25 Configuration management 730

25.1 Version management 735
25.2 System building 740
25.3 Change management 745
25.4 Release management 750

Glossary 757
Subject index 777
Author index 803



PART 1 Introduction to Software Engineering

My aim in this part of the book is to provide a general introduction to software engineering. The chapters in this part have been designed to support a one-semester first course in software engineering. I introduce important concepts such as software processes and agile methods, and describe essential software development activities, from requirements specification through to system evolution.

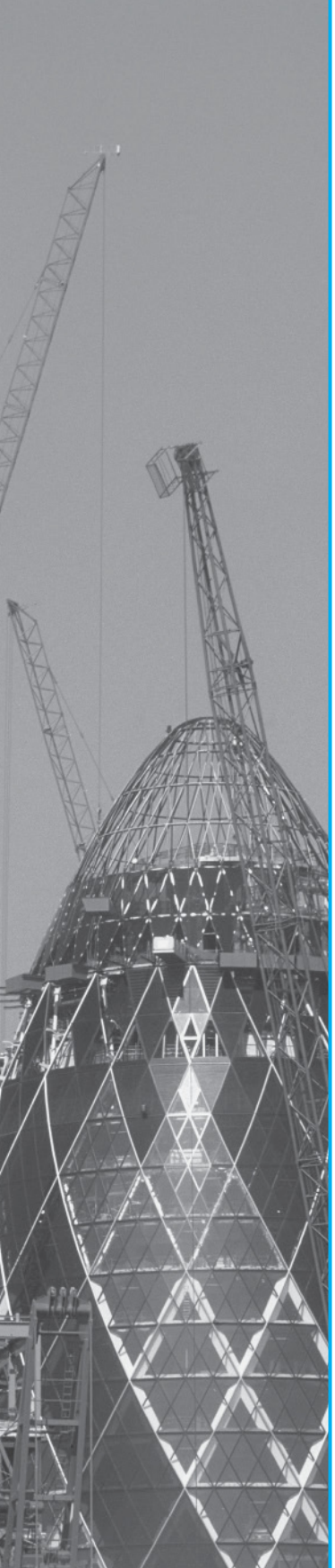
Chapter 1 is a general introduction that introduces professional software engineering and defines some software engineering concepts. I have also included a brief discussion of ethical issues in software engineering. It is important for software engineers to think about the wider implications of their work. This chapter also introduces four case studies that I use in the book. These are an information system for managing records of patients undergoing treatment for mental health problems (Mentcare), a control system for a portable insulin pump, an embedded system for a wilderness weather station and a digital learning environment (iLearn).

Chapters 2 and 3 cover software engineering processes and agile development. In Chapter 2, I introduce software process models, such as the waterfall model, and I discuss the basic activities that are part of these processes. Chapter 3 supplements this with a discussion of agile development methods for software engineering. This chapter had been

extensively changed from previous editions with a focus on agile development using Scrum and a discussion of agile practices such as stories for requirements definition and test-driven development.

The remaining chapters in this part are extended descriptions of the software process activities that are introduced in Chapter 2. Chapter 4 covers the critically important topic of requirements engineering, where the requirements for what a system should do are defined. Chapter 5 explains system modeling using the UML, where I focus on the use of use case diagrams, class diagrams, sequence diagrams and state diagrams for modeling a software system. In Chapter 6, I discuss the importance of software architecture and the use of architectural patterns in software design.

Chapter 7 introduces object oriented design and the use of design patterns. I also introduce important implementation issues here—reuse, configuration management and host-target development and discuss open source development. Chapter 8 focuses on software testing from unit testing during system development to the testing of software releases. I also discuss the use of test-driven development—an approach pioneered in agile methods but which has wide applicability. Finally, Chapter 9 presents an overview of software evolution issues. I cover evolution processes, software maintenance and legacy system management.



1 Introduction

Objectives

The objectives of this chapter are to introduce software engineering and to provide a framework for understanding the rest of the book. When you have read this chapter, you will:

- understand what software engineering is and why it is important;
- understand that the development of different types of software system may require different software engineering techniques;
- understand ethical and professional issues that are important for software engineers;
- have been introduced to four systems, of different types, which are used as examples throughout the book.

Contents

1.1 Professional software development
1.2 Software engineering ethics
1.3 Case studies

Software engineering is essential for the functioning of government, society, and national and international businesses and institutions. We can't run the modern world without software. National infrastructures and utilities are controlled by computer-based systems, and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Entertainment, including the music industry, computer games, and film and television, is software-intensive. More than 75% of the world's population have a software-controlled mobile phone, and, by 2016, almost all of these will be Internet-enabled.

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they governed by physical laws or by manufacturing processes. This simplifies software engineering, as there are no natural limits to the potential of software. However, because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change.

There are many different types of software system, ranging from simple embedded systems to complex, worldwide information systems. There are no universal notations, methods, or techniques for software engineering because different types of software require different approaches. Developing an organizational information system is completely different from developing a controller for a scientific instrument. Neither of these systems has much in common with a graphics-intensive computer game. All of these applications need software engineering; they do not all need the same software engineering methods and techniques.

There are still many reports of software projects going wrong and of "software failures." Software engineering is criticized as inadequate for modern software development. However, in my opinion, many of these so-called software failures are a consequence of two factors:

1. *Increasing system complexity* As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; and systems have to have new capabilities that were previously thought to be impossible. New software engineering techniques have to be developed to meet new the challenges of delivering more complex software.
2. *Failure to use software engineering methods* It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be. We need better software engineering education and training to address this problem.

Software engineers can be rightly proud of their achievements. Of course, we still have problems developing complex software, but without software engineering we would not have explored space and we would not have the Internet or modern telecommunications. All forms of travel would be more dangerous and expensive. Challenges for humanity in the 21st century are climate change, fewer natural



History of software engineering

The notion of software engineering was first proposed in 1968 at a conference held to discuss what was then called the software crisis (Naur and Randell 1969). It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding, and object-oriented development. Tools and standard notations were developed which are the basis of today's software engineering.

http://software-engineering-book.com/web/history/

resources, changing demographics, and an expanding world population. We will rely on software engineering to develop the systems that we need to cope with these issues.

1.1 Professional software development

Lots of people write programs. People in business write spreadsheet programs to simplify their jobs; scientists and engineers write programs to process their experimental data; hobbyists write programs for their own interest and enjoyment. However, most software development is a professional activity in which software is developed for business purposes, for inclusion in other devices, or as software products such as information systems and computer-aided design systems. The key distinctions are that professional software is intended for use by someone apart from its developer and that teams rather than individuals usually develop the software. It is maintained and changed throughout its life.

Software engineering is intended to support professional software development rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development. To help you to get a broad view of software engineering, I have summarized frequently asked questions about the subject in Figure 1.1.

Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, software is not just the programs themselves but also all associated documentation, libraries, support websites, and configuration data that are needed to make these programs useful. A professionally developed software system is often more than a single program. A system may consist of several separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the structure of the system, user documentation, which explains how to use the system, and websites for users to download recent product information.

This is one of the important differences between professional and amateur software development. If you are writing a program for yourself, no one else will use it

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. There are no methods and techniques that are good for everything.
What differences has the Internet made to software engineering?	Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an "app" industry for mobile devices which has changed the economics of software.

Figure 1.1 Frequently asked questions about software engineering

and you don't have to worry about writing program guides, documenting the program design, and so on. However, if you are writing software that other people will use and other engineers will change, then you usually have to provide additional information as well as the code of the program.

Software engineers are concerned with developing software products, that is, software that can be sold to a customer. There are two kinds of software product:

1. *Generic products* These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include apps for mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This kind of software also includes "vertical"

applications designed for a specific market such as library information systems, accounting systems, or systems for maintaining dental records.

2. *Customized (or bespoke) software* These are systems that are commissioned by and developed for a particular customer. A software contractor designs and implements the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

The critical distinction between these types of software is that, in generic products, the organization that develops the software controls the software specification. This means that if they run into development problems, they can rethink what is to be developed. For custom products, the specification is developed and controlled by the organization that is buying the software. The software developers must work to that specification.

However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as systems from SAP and Oracle, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

When we talk about the quality of professional software, we have to consider that the software is used and changed by people apart from its developers. Quality is therefore not just concerned with what the software does. Rather, it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation. This is reflected in the software's quality or non-functional attributes. Examples of these attributes are the software's response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, an aircraft control system must be safe, an interactive game must be responsive, a telephone switching system must be reliable, and so on. These can be generalized into the set of attributes shown in Figure 1.2, which I think are the essential characteristics of a professional software system.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. *Engineering discipline* Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively

Product characteristic	Description
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Software has to be secure so that malicious users cannot access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilization, etc.
Maintainability	Software should be written in such a way that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.

Figure 1.2 Essential attributes of good software

and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work within organizational and financial constraints, and they must look for solutions within these constraints.

2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software development.

Engineering is about getting results of the required quality within schedule and budget. This often involves making compromises—engineers cannot be perfectionists. People writing programs for themselves, however, can spend as much time as they wish on the program development.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances, so a more creative, less formal approach to development may be the right one for some kinds of software. A more flexible software process that accommodates rapid change is particularly appropriate for the development of interactive web-based systems and mobile apps, which require a blend of software and graphical design skills.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for professional software systems rather than just write programs as

a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. Four fundamental activities are common to all software processes.

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Different types of systems need different development processes, as I explain in Chapter 2. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail, depending on the type of software being developed.

Software engineering is related to both computer science and systems engineering.

1. Computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science are rarely relevant to large, complex problems that require a software solution.
2. System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design, and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system.

As I discuss in the next section, there are many different types of software. There are no universal software engineering methods or techniques that may be used. However, there are four related issues that affect many different types of software:

1. *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones and tablets. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.
2. *Business and social change* Businesses and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software. Many traditional software engineering techniques are time consuming, and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.
3. *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot successfully attack our software and that information security is maintained.
4. *Scale* Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

To address these challenges, we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

1.1.2 Software engineering diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. The specific methods, tools, and techniques used depend on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years. However, the SEMAT initiative (Jacobson et al. 2013) proposes that there can be a fundamental meta-process that can be instantiated to create different kinds of process. This is at an early stage of development and may be a basis for improving our current software engineering methods.

Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application being developed. There are many different types of application, including:

1. *Stand-alone applications* These are application systems that run on a personal computer or apps that run on a mobile device. They include all necessary functionality and may not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, travel apps, productivity apps, and so on.

2. *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own computers, phones, or tablets. Obviously, these include web applications such as e-commerce applications where you interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.

3. *Embedded control systems* These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls antilock braking in a car, and software in a microwave oven to control the cooking process.

4. *Batch processing systems* These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems are periodic billing systems, such as phone billing systems, and salary payment systems.

5. *Entertainment systems* These are systems for personal use that are intended to entertain the user. Most of these systems are games of one kind or another, which may run on special-purpose console hardware. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

6. *Systems for modeling and simulation* These are systems that are developed by scientists and engineers to model physical processes or situations, which include many separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.

7. *Data collection and analysis systems* Data collection systems are systems that collect data from their environment and send that data to other systems for processing. The software may have to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location. “Big data” analysis may involve cloud-based systems carrying out statistical analysis and looking for relationships in the collected data.

8. *Systems of systems* These are systems, used in enterprises and other large organizations, that are composed of a number of other software systems. Some of these may be generic software products, such as an ERP system. Other systems in the assembly may be specially written for that environment.

Of course, the boundaries between these system types are blurred. If you develop a game for a phone, you have to take into account the same constraints (power, hardware interaction) as the developers of the phone software. Batch processing systems are often used in conjunction with web-based transaction systems. For example, in a company, travel expense claims may be submitted through a web application but processed in a batch application for monthly payment.

Each type of system requires specialized software engineering techniques because the software has different characteristics. For example, an embedded control system in an automobile is safety-critical and is burned into ROM (read-only memory) when installed in the vehicle. It is therefore very expensive to change. Such a system needs extensive verification and validation so that the chances of having to recall cars after sale to fix software problems are minimized. User interaction is minimal (or perhaps nonexistent), so there is no need to use a development process that relies on user interface prototyping.

For an interactive web-based system or app, iterative development and delivery is the best approach, with the system being composed of reusable components. However, such an approach may be impractical for a system of systems, where detailed specifications of the system interactions have to be specified in advance so that each system can be separately developed.

Nevertheless, there are software engineering fundamentals that apply to all types of software systems:

1. They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, the specific process that you should use depends on the type of software that you are developing.
2. Dependability and performance are important for all types of system. Software should behave as expected, without failures, and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
3. Understanding and managing the software specification and requirements (what the software should do) are important. You have to know what different customers and users of the system expect from it, and you have to manage their expectations so that a useful system can be delivered within budget and to schedule.
4. You should make effective use of existing resources. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

These fundamental notions of process, dependability, requirements, management, and reuse are important themes of this book. Different methods reflect them in different ways, but they underlie all professional software development.

These fundamentals are independent of the program language used for software development. I don't cover specific programming techniques in this book because these vary dramatically from one type of system to another. For example, a dynamic language, such as Ruby, is the right type of language for interactive system development but is inappropriate for embedded systems engineering.

1.1.3 Internet software engineering

The development of the Internet and the World Wide Web has had a profound effect on all of our lives. Initially, the web was primarily a universally accessible information store, and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organization. Around 2000, the web started to evolve, and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of a vast range of new system products that delivered innovative services, accessed over the web. These are often funded by adverts that are displayed on the user's screen and do not involve direct payment from users.

As well as these system products, the development of web browsers that could run small programs and do some local processing led to an evolution in business and organizational software. Instead of writing software and deploying it on users' PCs, the software was deployed on a web server. This made it much cheaper to change and upgrade the software, as there was no need to install the software on every PC. It also reduced costs, as user interface development is particularly expensive. Wherever it has been possible to do so, businesses have moved to web-based interaction with company software systems.

The notion of software as a service (Chapter 17) was proposed early in the 21st century This has now become the standard approach to the delivery of web-based system products such as Google Apps, Microsoft Office 365, and Adobe Creative Suite. More and more software runs on remote "clouds" instead of local servers and is accessed over the Internet. A computing cloud is a huge number of linked computer systems that is shared by many users. Users do not buy software but pay according to how much the software is used or are given free access in return for watching adverts that are displayed on their screen. If you use services such as web-based mail, storage, or video, you are using a cloud-based system.

The advent of the web has led to a dramatic change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters. Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.

This change in software organization has had a major effect on software engineering for web-based systems. For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from preexisting software components and systems, often bundled together in a framework.
2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems are always developed and delivered incrementally.
3. Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services. I discuss this approach to software engineering in Chapter 18.
4. Interface development technology such as AJAX (Holdener 2008) and HTML5 (Freeman 2011) have emerged that support the creation of rich interfaces within a web browser.

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software, as they do to other types of software. Web-based systems are getting larger and larger, so software engineering techniques that deal with scale and complexity are relevant for these systems.

1.2 Software engineering ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.

It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. *Confidentiality* You should normally respect the confidentiality of your employers or clients regardless of whether or not a formal confidentiality agreement has been signed.
2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.3 The ACM/IEEE Code of Ethics (ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version. http://www.acm.org/about/se-code)

(© 1999 by the ACM, Inc. and the IEEE, Inc.)

4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine) to extremely serious (dissemination of viruses or other malware).

Professional societies and institutions have an important role to play in setting ethical standards. Organizations such as the ACM, the IEEE (Institute of Electrical and Electronic Engineers), and the British Computer Society publish a code of professional conduct or code of ethics. Members of these organizations undertake to follow that code when they sign up for membership. These codes of conduct are generally concerned with fundamental ethical behavior.

Professional associations, notably the ACM and the IEEE, have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form, shown in Figure 1.3, and a longer form (Gotterbarn, Miller, and Rogerson 1999) that adds detail and substance to the shorter version. The rationale behind this code is summarized in the first two paragraphs of the longer form:

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice[†].

The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer[†].

In any situation where different people have different views and objectives, you are likely to be faced with ethical dilemmas. For example, if you disagree, in principle, with the policies of more senior management in the company, how should you react? Clearly, this depends on the people involved and the nature of the disagreement. Is it best to argue a case for your position from within the organization or to resign in principle? If you feel that there are problems with a software project, when do you reveal these problems to management? If you discuss these while they are just a suspicion, you may be overreacting to a situation; if you leave it too long, it may be impossible to resolve the difficulties.

We all face such ethical dilemmas in our professional lives, and, fortunately, in most cases they are either relatively minor or can be resolved without too much difficulty. Where they cannot be resolved, the engineer is faced with, perhaps, another problem. The principled action may be to resign from their job, but this may well affect others such as their partner or their children.

A difficult situation for professional engineers arises when their employer acts in an unethical way. Say a company is responsible for developing a safety-critical system and, because of time pressure, falsifies the safety validation records. Is the engineer's responsibility to maintain confidentiality or to alert the customer or publicize, in some way, that the delivered system may be unsafe?

[†]ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version Preamble. http://www.acm.org/about/se-code Copyright © 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

The problem here is that there are no absolutes when it comes to safety. Although the system may not have been validated according to predefined criteria, these criteria may be too strict. The system may actually operate safely throughout its lifetime. It is also the case that, even when properly validated, the system may fail and cause an accident. Early disclosure of problems may result in damage to the employer and other employees; failure to disclose problems may result in damage to others.

You must make up your own mind in these matters. The appropriate ethical position here depends on the views of the people involved. The potential for damage, the extent of the damage, and the people affected by the damage should influence the decision. If the situation is very dangerous, it may be justified to publicize it using the national press or social media. However, you should always try to resolve the situation while respecting the rights of your employer.

Another ethical issue is participation in the development of military and nuclear systems. Some people feel strongly about these issues and do not wish to participate in any systems development associated with defense systems. Others will work on military systems but not on weapons systems. Yet others feel that national security is an overriding principle and have no ethical objections to working on weapons systems.

In this situation, it is important that both employers and employees should make their views known to each other in advance. Where an organization is involved in military or nuclear work, it should be able to specify that employees must be willing to accept any work assignment. Equally, if an employee is taken on and makes clear that he or she does not wish to work on such systems, employers should not exert pressure to do so at some later date.

The general area of ethics and professional responsibility is increasingly important as software-intensive systems pervade every aspect of work and everyday life. It can be considered from a philosophical standpoint where the basic principles of ethics are considered and software engineering ethics are discussed with reference to these basic principles. This is the approach taken by Laudon (Laudon 1995) and Johnson (Johnson 2001). More recent texts such as that by Tavani (Tavani 2013) introduce the notion of cyberethics and cover both the philosophical background and practical and legal issues. They include ethical issues for technology users as well as developers.

I find that a philosophical approach is too abstract and difficult to relate to everyday experience so I prefer the more concrete approach embodied in professional codes of conduct (Bott 2005; Duquenoy 2007). I think that ethics are best discussed in a software engineering context and not as a subject in its own right. Therefore, I do not discuss software engineering ethics in an abstract way but include examples in the exercises that can be the starting point for a group discussion.

1.3 Case studies

To illustrate software engineering concepts, I use examples from four different types of system. I have deliberately not used a single case study, as one of the key messages in this book is that software engineering practice depends on the type of systems

being produced. I therefore choose an appropriate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.

The system types that I use as case studies are:

1. *An embedded system* This is a system where the software controls some hardware device and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, and power management, etc. The example of an embedded system that I use is a software system to control an insulin pump for people who have diabetes.
2. *An information system* The primary purpose of this type of system is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system used is a medical records system.
3. *A sensor-based data collection system* This is a system whose primary purposes are to collect data from a set of sensors and to process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.
4. *A support environment.* This is an integrated collection of software tools that are used to support some kind of activity. Programming environments, such as Eclipse (Vogel 2012) will be the most familiar type of environment for readers of this book. I describe an example here of a digital learning environment that is used to support students' learning in schools.

I introduce each of these systems in this chapter; more information about each of them is available on the website (software-engineering-book.com).

1.3.1 An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system that collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition in which the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolizes glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels periodically using an external meter and then estimate the dose of insulin they should inject.

The problem is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. Irregular checking can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and,

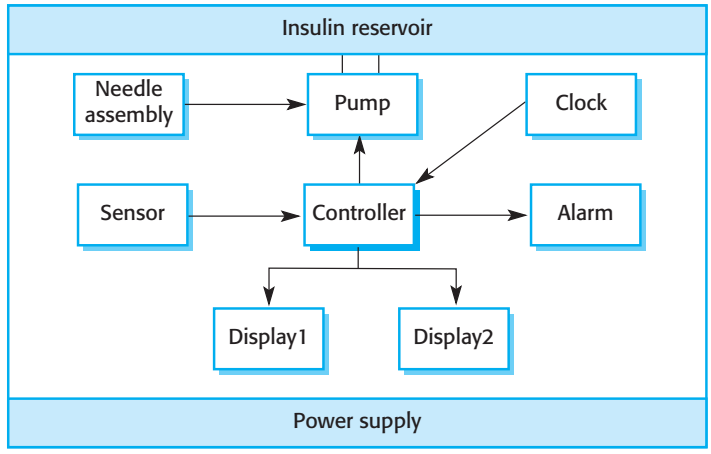


Figure 1.4 Insulin pump hardware architecture

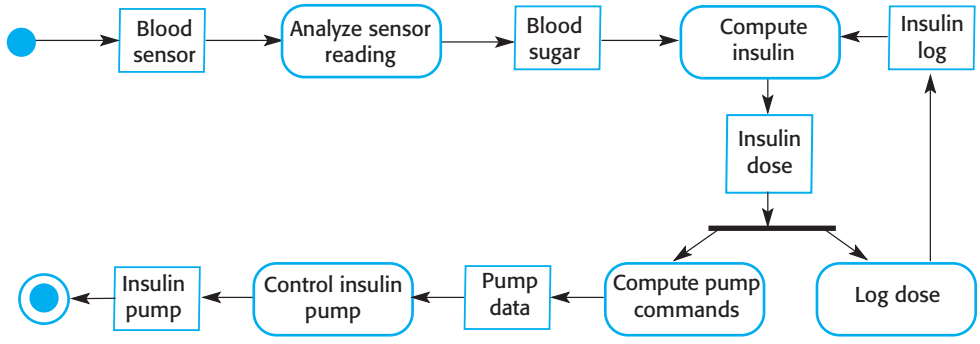


Figure 1.5 Activity model of the insulin pump

ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.

Advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this one are now available and are used by patients who find it difficult to control their insulin levels. In future, it may be possible for diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system uses a microsensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

Figure 1.4 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a controller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Figure 1.5 is a Unified Modeling

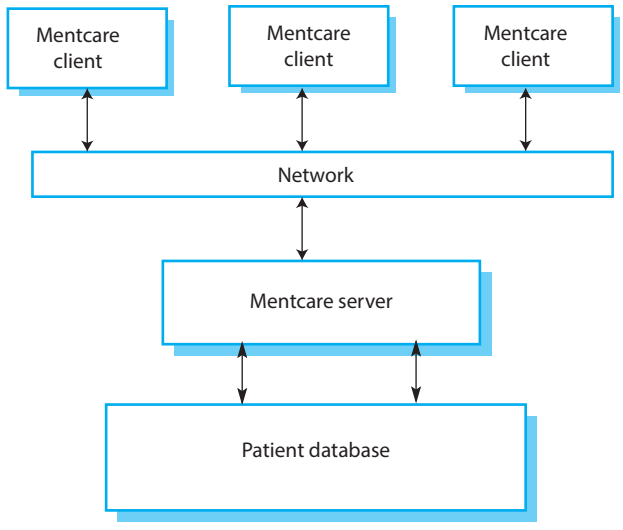


Figure 1.6 The organization of the Mentcare system

Language (UML) activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. This system must therefore meet two essential high-level requirements:

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

The system must therefore be designed and implemented to ensure that it always meets these requirements. More detailed requirements and discussions of how to ensure that the system is safe are discussed in later chapters.

1.3.2 A patient information system for mental health care

A patient information system to support mental health care (the Mentcare system) is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.

The Mentcare system (Figure 1.6) is a patient information system that is intended for use in clinics. It makes use of a centralized database of patient information but

has also been designed to run on a laptop, so that it may be accessed and used from sites that do not have secure network connectivity. When the local systems have secure network access, they use patient information in the database, but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system and so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems.

This system has two purposes:

1. To generate management information that allows health service managers to assess performance against local and government targets.
2. To provide medical staff with timely information to support the treatment of patients.

Patients who suffer from mental health problems are sometimes irrational and disorganized so may miss appointments, deliberately or accidentally lose prescriptions and medication, forget instructions and make unreasonable demands on medical staff. They may drop in on clinics unexpectedly. In a minority of cases, they may be a danger to themselves or to other people. They may regularly change address or may be homeless on a long-term or short-term basis. Where patients are dangerous, they may need to be "sectioned"—that is, confined to a secure hospital for treatment and observation.

Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.

The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments. Reports are generated at regular intervals for medical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients, whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.

The key features of the system are:

1. *Individual care management* Clinicians can create records for patients, edit the information in the system, view patient history, and so on. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. *Patient monitoring* The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.

3. *Administrative reporting* The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, the number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system: laws on data protection that govern the confidentiality of personal information and mental health laws that govern the compulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to strict legislative safeguards. One aim of the Mentcare system is to ensure that staff always act in accordance with the law and that their decisions are recorded for judicial review if necessary.

As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves. The Mentcare system is also a safety-critical system. Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.

The overall design of the system has to take into account privacy and safety requirements. The system must be available when needed; otherwise safety may be compromised, and it may be impossible to prescribe the correct medication to patients. There is a potential conflict here. Privacy is easiest to maintain when there is only a single copy of the system data. However, to ensure availability in the event of server failure or when disconnected from a network, multiple copies of the data should be maintained. I discuss the trade-offs between these requirements in later chapters.

1.3.3 A wilderness weather station

To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas. These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.

Wilderness weather stations are part of a larger system (Figure 1.7), which is a weather information system that collects data from weather stations and makes it available to other systems for processing. The systems in Figure 1.7 are:

1. *The weather station system* This system is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
2. *The data management and archiving system* This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.

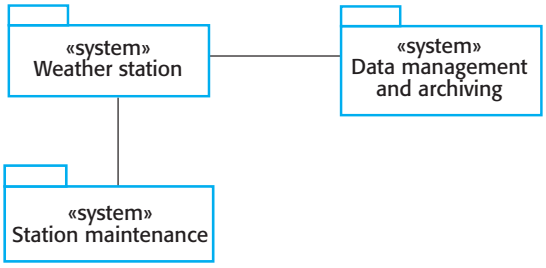


Figure 1.7 The weather station's environment

3. *The station maintenance system* This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control the weather station.

In Figure 1.7, I have used the UML package symbol to indicate that each system is a collection of components and the separate systems are identified using the UML stereotype «system». The associations between the packages indicate there is an exchange of information but, at this stage, there is no need to define them in any more detail.

The weather stations include instruments that measure weather parameters such as wind speed and direction, ground and air temperatures, barometric pressure, and rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

The weather station system operates by collecting weather observations at frequent intervals; for example, temperatures are measured every minute. However, because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data. It then transmits this aggregated data when requested by the data collection system. If it is impossible to make a connection, then the weather station maintains the data locally until communication can be resumed.

Each weather station is battery-powered and must be entirely self-contained; there are no external power or network cables. All communications are through a relatively slow satellite link, and the weather station must include some mechanism (solar or wind power) to charge its batteries. As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals. The station software is therefore not just concerned with data collection. It must also:

1. Monitor the instruments, power. and communication hardware and report faults to the management system.
2. Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.

3. Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

Because weather stations have to be self-contained and unattended, this means that the software installed is complex, even though the data collection functionality is fairly simple.

1.3.4 A digital learning environment for schools

Many teachers argue that using interactive software systems to support education can lead to both improved learner motivation and a deeper level of knowledge and understanding in students. However, there is no general agreement on the 'best' strategy for computer-supported learning, and teachers in practice use a range of different interactive, web-based tools to support learning. The tools used depend on the ages of the learners, their cultural background, their experience with computers, equipment available, and the preferences of the teachers involved.

A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded, plus a set of applications that are geared to the needs of the learners using the system. The framework provides general services such as an authentication service, synchronous and asynchronous communication services, and a storage service.

The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs. These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games, and simulations. They may also include specific content, such as content about the American Civil War and applications to view and annotate that content.

Figure 1.8 is a high-level architectural model of a digital learning environment (iLearn) that was designed for use in schools for students from 3 to 18 years of age. The approach adopted is that this is a distributed system in which all components of the environment are services that can be accessed from anywhere on the Internet. There is no requirement that all of the learning tools are gathered together in one place.

The system is a service-oriented system with all system components considered to be a replaceable service. There are three types of service in the system:

1. *Utility services* that provide basic application-independent functionality and that may be used by other services in the system. Utility services are usually developed or adapted specifically for this system.
2. *Application services* that provide specific applications such as email, conferencing, photo sharing, etc., and access to specific educational content such as scientific films or historical resources. Application services are external services that are either specifically purchased for the system or are available freely over the Internet.

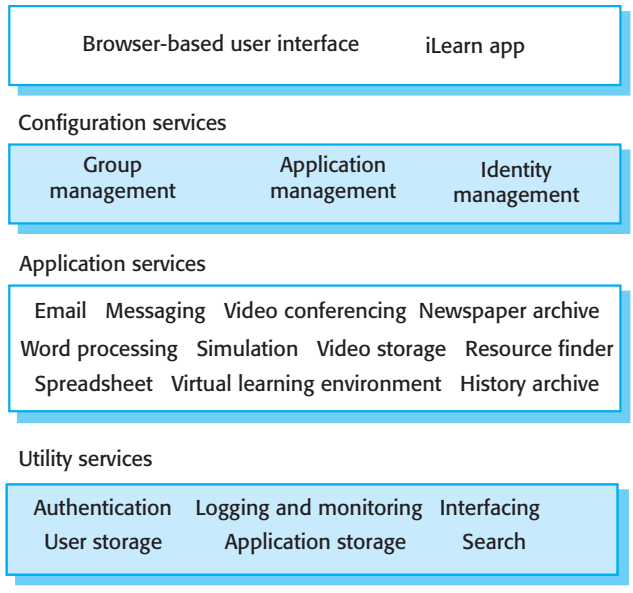


Figure 1.8 The architecture of a digital learning environment (iLearn)

3. *Configuration services* that are used to adapt the environment with a specific set of application services and to define how services are shared between students, teachers, and their parents.

The environment has been designed so that services can be replaced as new services become available and to provide different versions of the system that are suited for the age of the users. This means that the system has to support two levels of service integration:

1. *Integrated services* are services that offer an API (application programming interface) and that can be accessed by other services through that API. Direct service-to-service communication is therefore possible. An authentication service is an example of an integrated service. Rather than use their own authentication mechanisms, an authentication service may be called on by other services to authenticate users. If users are already authenticated, then the authentication service may pass authentication information directly to another service, via an API, with no need for users to reauthenticate themselves.
2. *Independent services* are services that are simply accessed through a browser interface and that operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; reauthentication may be required for each independent service.

If an independent service becomes widely used, the development team may then integrate that service so that it becomes an integrated and supported service.

KEY POINTS

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software is not just a program or programs but also includes all electronic documentation that is needed by system users, quality assurance staff, and developers. Essential software product attributes are maintainability, dependability and security, efficiency, and acceptability.
- The software process includes all of the activities involved in software development. The high-level activities of specification, development, validation, and evolution are part of all software processes.
- There are many different types of system, and each requires appropriate software engineering tools and techniques for their development. Few, if any, specific design and implementation techniques are applicable to all kinds of system.
- The fundamental ideas of software engineering are applicable to all types of software system. These fundamentals include managed software processes, software dependability and security, requirements engineering, and software reuse.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues but should be aware of the ethical issues that affect their work.
- Professional societies publish codes of conduct that embed ethical and professional standards. These set out the standards of behavior expected of their members.

FURTHER READING

"Software Engineering Code of Ethics Is Approved." An article that discusses the background to the development of the ACM/IEEE Code of Ethics and that includes both the short and long form of the code. (*Comm. ACM*, D. Gotterbarn, K. Miller, and S. Rogerson, October 1999). http://dx.doi.org/10.1109/MC.1999.796142

"A View of 20th and 21st Century Software Engineering." A backward and forward look at software engineering from one of the first and most distinguished software engineers. Barry Boehm identifies timeless software engineering principles but also suggests that some commonly used practices are obsolete. (B. Boehm, *Proc. 28th Software Engineering Conf.,* Shanghai. 2006). http://dx.doi.org/10.1145/1134285.1134288

"Software Engineering Ethics." Special issue of *IEEE Computer*, with several papers on the topic (*IEEE Computer,* 42 (6), June 2009).

Ethics for the Information Age. This is a wide-ranging book that covers all aspects of information technology (IT) ethics, not simply ethics for software engineers. I think this is the right approach as you really need to understand software engineering ethics within a wider ethical framework (M. J. Quinn, 2013, Addison-Wesley).

The Essence of Software Engineering: Applying the SEMAT kernel. This book discusses the idea of a universal framework that can underlie all software engineering methods. It can be adapted and used for all types of systems and organizations. I am personally skeptical about whether or not a universal approach is realistic in practice, but the book has some interesting ideas that are worth exploring. (I. Jacobsen, P-W Ng, P. E. McMahon, I. Spence, and S. Lidman, 2013, Addison-Wesley)

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-engineering/

Links to case study descriptions:

http://software-engineering-book.com/case-studies/

EXERCISES

1.1. Explain why professional software that is developed for a customer is not simply the programs that have been developed and delivered.

1.2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?

1.3. Briefly discuss why it is usually cheaper in the long run to use software engineering methods and techniques for software systems.

1.4. Software engineering is not only concerned with issues like system heterogeneity, business and social change, trust, and security, but also with ethical issues affecting the domain. Give some examples of ethical issues that have an impact on the software engineering domain.

1.5. Based on your own knowledge of some of the application types discussed in Section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.

1.6. Explain why the fundamental software engineering principles of process, dependability, requirements management, and reuse are relevant to all types of software system.

1.7. Explain how electronic connectivity between various development teams can support software engineering activities.

1.8. Noncertified individuals are still allowed to practice software engineering. Discuss some of the possible drawbacks of this.

1.9. For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.4, propose an appropriate example that illustrates that clause.

1.10. The "Drone Revolution" is currently being debated and discussed all over the world. Drones are unmanned flying machines that are built and equipped with various kinds of software systems that allow them to see, hear, and act. Discuss some of the societal challenges of building such kinds of systems.

REFERENCES

Bott, F. 2005. *Professional Issues in Information Technology*. Swindon, UK: British Computer Society.

Duquenoy, P. 2007. *Ethical, Legal and Professional Issues in Computing*. London: Thomson Learning.

Freeman, A. 2011. *The Definitive Guide to HTML5*. New York: Apress.

Gotterbarn, D., K. Miller, and S. Rogerson. 1999. "Software Engineering Code of Ethics Is Approved." *Comm. ACM* 42 (10): 102–107. doi:10.1109/MC.1999.796142.

Holdener, A. T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobson, I., P-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013. *The Essence of Software Engineering*. Boston: Addison-Wesley.

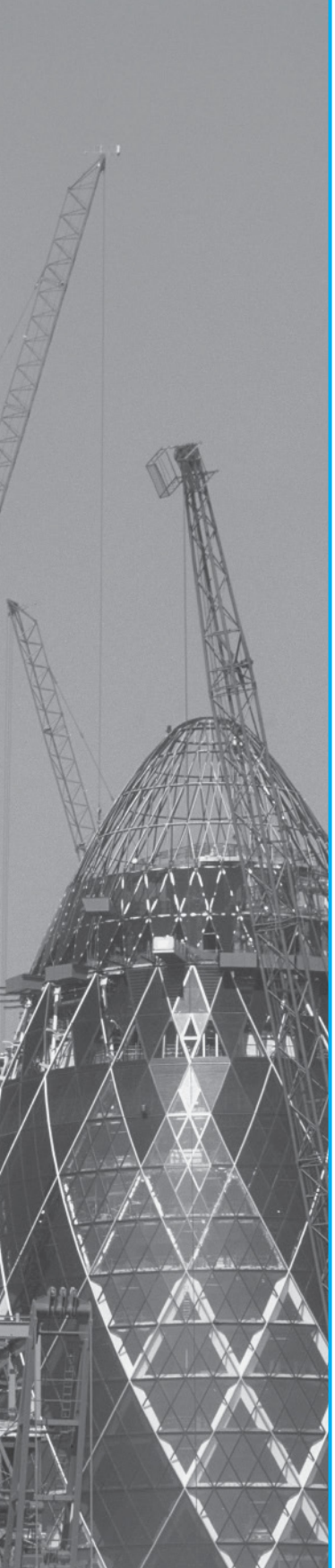
Johnson, D. G. 2001. *Computer Ethics*. Englewood Cliffs, NJ: Prentice-Hall.

Laudon, K. 1995. "Ethical Concepts and Information Technology." *Comm. ACM* 38 (12): 33–39. doi:10.1145/219663.219677.

Naur, P., and Randell, B. 1969. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Brussels. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf

Tavani, H. T. 2013. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing, 4th ed.* New York: John Wiley & Sons.

Vogel, L. 2012. *Eclipse 4 Application Development: The Complete Guide to Eclipse 4 RCP Development*. Sebastopol, CA: O'Reilly & Associates.



2 Software processes

Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter, you will:

- understand the concepts of software processes and software process models;
- have been introduced to three general software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand the notion of software process improvement and the factors that affect software process quality.

Contents

2.1 Software process models
2.2 Process activities
2.3 Coping with change
2.4 Process improvement

A software process is a set of related activities that leads to the production of a software system. As I discussed in Chapter 1, there are many different types of software systems, and there is no universal software engineering method that is applicable to all of them. Consequently, there is no universally applicable software process. The process used in different companies depends on the type of software being developed, the requirements of the software customer, and the skills of the people writing the software.

However, although there are many different software processes, they all must include, in some form, the four fundamental software engineering activities that I introduced in Chapter 1:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software development* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

These activities are complex activities in themselves, and they include subactivities such as requirements validation, architectural design, and unit testing. Processes also include other activities, such as software configuration management and project planning that support production activities.

When we describe and discuss processes, we usually talk about the activities in these processes, such as specifying a data model and designing a user interface, and the ordering of these activities. We can all relate to what people do to develop software. However, when describing processes, it is also important to describe who is involved, what is produced, and conditions that influence the sequence of activities:

1. Products or deliverables are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, and programmer.
3. Pre- and postconditions are conditions that must hold before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a precondition may be that the consumer has approved all requirements; after this activity is finished, a postcondition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. As there is no universal process that is right for all kinds of software, most software companies have developed their own

development processes. Processes have evolved to take advantage of the capabilities of the software developers in an organization and the characteristics of the systems that are being developed. For safety-critical systems, a very structured development process is required where detailed records are maintained. For business systems, with rapidly changing requirements, a more flexible, agile process is likely to be better.

As I discussed in Chapter 1, professional Professional software development is a managed activity, so planning is an inherent part of all processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and continual as the software is developed. It is therefore easier to change the process to reflect changing customer or product requirements. As Boehm and Turner (Boehm and Turner 2004) explain, each approach is suitable for different types of software. Generally, for large systems, you need to find a balance between plan-driven and agile processes.

Although there is no universal software process, there is scope for process improvement in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development. They can improve their process by introducing techniques such as UML modeling and test-driven development. I discuss software process improvement briefly later in thischapter text and in more detail in web Chapter 26.

2.1 Software process models

As I explained in Chapter 1, a software process model (sometimes called a Software Development Life Cycle or SDLC model) is a simplified representation of a software process. Each process model represents a process from a particular perspective and thus only provides partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called *process paradigms*) and present these from an architectural perspective. That is, we see the framework of the process but not the details of process activities.

These generic models are high-level, abstract descriptions of software processes that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The general process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, and testing.



The Rational Unified Process

The Rational Unified Process (RUP) brings together elements of all of the general process models discussed here and supports prototyping and incremental delivery of software (Krutchen 2003). The RUP is normally described from three perspectives: a dynamic perspective that shows the phases of the model in time, a static perspective that shows process activities, and a practice perspective that suggests good practices to be used in the process. Phases of the RUP are inception, where a business case for the system is established; elaboration, where requirements and architecture are developed; construction where the software is implemented; and transition, where the system is deployed.

http://software-engineering-book.com/web/rup/

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. *Integration and configuration* This approach relies on the availability of reusable components or systems. The system development process focuses on configuring these components for use in a new setting and integrating them into a system.

As I have said, there is no universal process model that is right for all kinds of software development. The right process depends on the customer and regulatory requirements, the environment where the software will be used, and the type of software being developed. For example, safety-critical software is usually developed using a waterfall process as lots of analysis and documentation is required before implementation begins. Software products are now always developed using an incremental process model. Business systems are increasingly being developed by configuring existing systems and integrating these to create a new system with the functionality that is required.

The majority of practical software processes are based on a general model but often incorporate features of other models. This is particularly true for large systems engineering. For large systems, it makes sense to combine some of the best features of all of the general processes. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Subsystems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process or may be bought in as off-the-shelf systems for configuration. Other parts of the system, which are difficult to specify in advance, should always be developed using an incremental approach. In both cases, software components are likely to be reused.

Various attempts have been made to develop "universal" process models that draw on all of these general models. One of the best known of these universal models is the Rational Unified Process (RUP) (Krutchen 2003), which was developed by Rational, a U.S. software engineering company. The RUP is a flexible model that

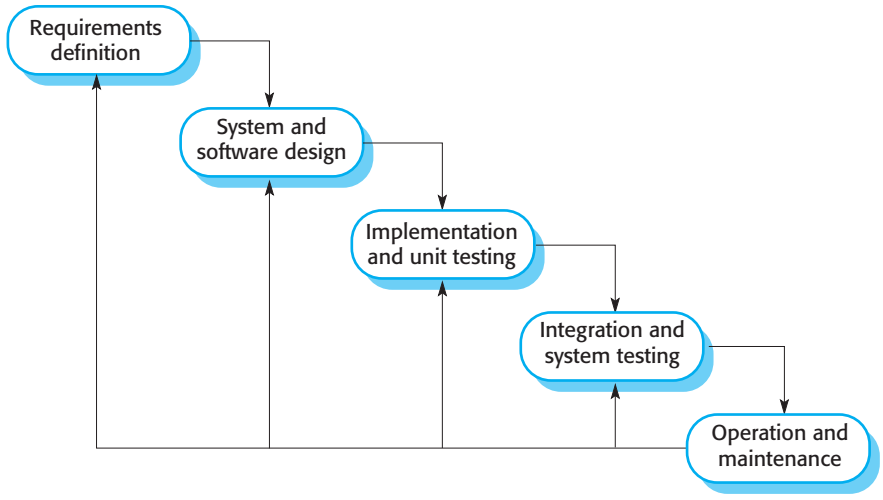


Figure 2.1 The waterfall model

can be instantiated in different ways to create processes that resemble any of the general process models discussed here. The RUP has been adopted by some large software companies (notably IBM), but it has not gained widespread acceptance.

2.1.1 The waterfall model

The first published model of the software development process was derived from engineering process models used in large military systems engineering (Royce 1970). It presents the software development process as a number of stages, as shown in Figure 2.1. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. The waterfall model is an example of a plan-driven process. In principle at least, you plan and schedule all of the process activities before starting software development.

The stages of the waterfall model directly reflect the fundamental software development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.



Boehm's spiral process model

Barry Boehm, one of the pioneers in software engineering, proposed an incremental process model that was risk-driven. The process is represented as a spiral rather than a sequence of activities (Boehm 1988).

Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

http://software-engineering-book.com/web/spiral-model/

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. *Operation and maintenance* Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase in the waterfall model is one or more documents that are approved ("signed off"). The following phase should not start until the previous phase has finished. For hardware development, where high manufacturing costs are involved, this makes sense. However, for software development, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found, and so on. The software process, in practice, is never a simple linear model but involves feedback from one phase to another.

As new information emerges in a process stage, the documents produced at previous stages should be modified to reflect the required system changes. For example, if it is discovered that a requirement is too expensive to implement, the requirements document should be changed to remove that requirement. However, this requires customer approval and delays the overall development process.

As a result, both customers and developers may prematurely freeze the software specification so that no further changes are made to it. Unfortunately, this means that problems are left for later resolution, ignored, or programmed around. Premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life-cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered.

Program and design errors emerge, and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

In reality, software has to be flexible and accommodate change as it is being developed. The need for early commitment and system rework when changes are made means that the waterfall model is only appropriate for some types of system:

1. Embedded systems where the software has to interface with hardware systems. Because of the inflexibility of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.
2. Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible. Safety-related problems in the specification and design are usually very expensive to correct at the implementation stage.
3. Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software. Furthermore, where several companies are involved, complete specifications may be needed to allow for the independent development of different subsystems.

The waterfall model is not the right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Formal development processes, such as that based on the B method (Abrial 2005, 2010), are mostly used in the development of software systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements. However, because of the high costs of developing a formal specification, this development model is rarely used except for critical systems engineering.

2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed (Figure 2.2). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

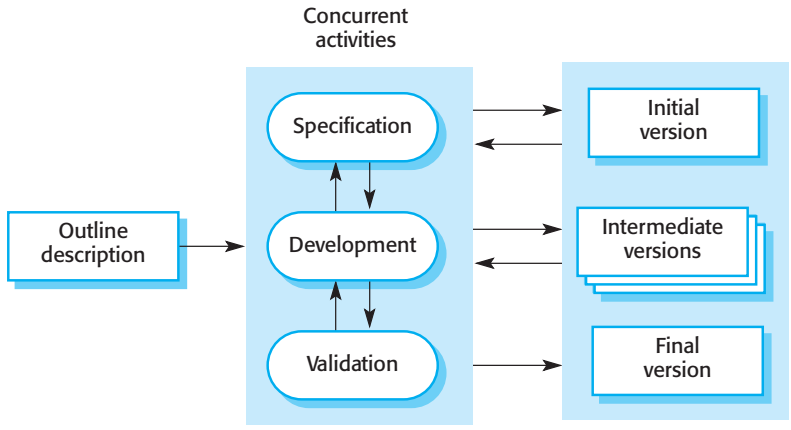


Figure 2.2 Incremental development

Incremental development in some form is now the most common approach for the development of application systems and software products. This approach can be either plan-driven, agile or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified, but the development of later increments depends on progress and customer priorities.

Incremental software development, which is a fundamental part of agile development methods, is better than a waterfall approach for systems whose requirements are likely to change during the development process. This is the case for most business systems and software products. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three major advantages over the waterfall model:

1. The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how



Problems with incremental development

Although incremental development has many advantages, it is not problem free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons. For example, there may be procedures to ensure that the software meets properly implements external regulations (e.g., in the United States, the Sarbanes Oxley accounting regulations). Changing these procedures may not be possible, so process conflicts may be unavoidable.

http://software-engineering-book.com/web/incremental-development/

much has been implemented. Customers find it difficult to judge progress from software design documents.

3. Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system. To reduce structural degradation and general code messiness, agile methods suggest that you should regularly refactor (improve and restructure) the software.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture, and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

Incremental development does not mean that you have to deliver each increment to the system customer. You can develop a system incrementally and expose it to customers and other stakeholders for comment, without necessarily delivering it and deploying it in the customer's environment. Incremental delivery (covered in Section 2.3.2) means that the software is used in real, operational processes, so user feedback is likely to be realistic. However, providing feedback is not always possible as experimenting with new software can disrupt normal business processes.

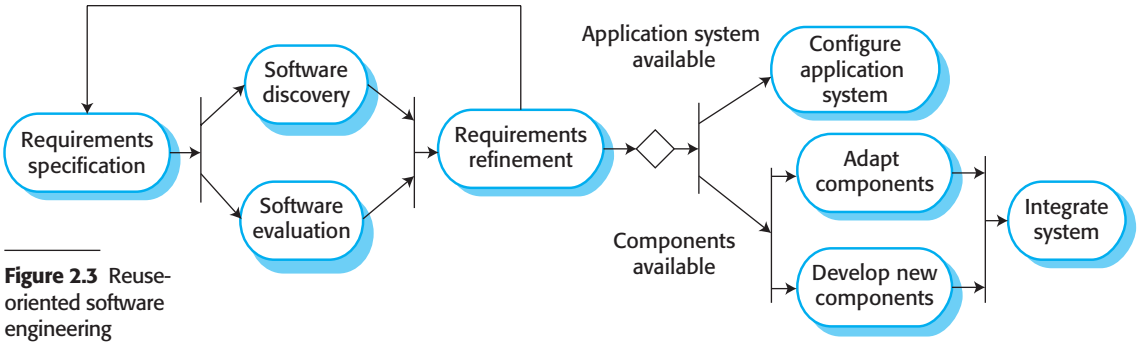


Figure 2.3 Reuse-oriented software engineering

2.1.3 Integration and configuration

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of or search for code that is similar to what is required. They look for these, modify them as needed, and integrate them with the new code that they have developed.

This informal reuse takes place regardless of the development process that is used. However, since 2000, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a base of reusable software components and an integrating framework for the composition of these components.

Three types of software components are frequently reused:

1. Stand-alone application systems that are configured for use in a particular environment. These systems are general-purpose systems that have many features, but they have to be adapted for use in a specific application.
2. Collections of objects that are developed as a component or as a package to be integrated with a component framework such as the Java Spring framework (Wheeler and White 2013).
3. Web services that are developed according to service standards and that are available for remote invocation over the Internet.

Figure 2.3 shows a general process model for reuse-based development, based on integration and configuration. The stages in this process are:

1. *Requirements specification* The initial requirements for the system are proposed. These do not have to be elaborated in detail but should include brief descriptions of essential requirements and desirable system features.
2. *Software discovery and evaluation* Given an outline of the software requirements, a search is made for components and systems that provide the functionality required. Candidate components and systems are evaluated to see if



Software development tools

Software development tools are programs that are used to support software engineering process activities. These tools include requirements management tools, design editors, refactoring support tools, compilers, debuggers, bug trackers, and system building tools.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. For example:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way.

http://software-engineering-book.com/web/software-tools/

they meet the essential requirements and if they are generally suitable for use in the system.

3. *Requirements refinement* During this stage, the requirements are refined using information about the reusable components and applications that have been discovered. The requirements are modified to reflect the available components, and the system specification is re-defined. Where modifications are impossible, the component analysis activity may be reentered to search for alternative solutions.
4. *Application system configuration* If an off-the-shelf application system that meets the requirements is available, it may then be configured for use to create the new system.
5. *Component adaptation and integration* If there is no off-the-shelf system, individual reusable components may be modified and new components developed. These are then integrated to create the system.

Reuse-oriented software engineering, based around configuration and integration, has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable, and this may lead to a system

that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important, and so several chapters in the third I have dedicated several chapters in the 3rd part of the book to this topic. General issues of software reuse are covered in Chapter 15, component-based software engineering in Chapters 16 and 17, and service-oriented systems in Chapter 18.

2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Generally, processes are now tool-supported. This means that software developers may use a range of software tools to help them, such as requirements management systems, design model editors, program editors, automated testing tools, and debuggers.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software being developed, the experience and competence of the developers, and the type of organization developing the software.

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process, as mistakes made at this stage inevitably lead to later problems in the system design and implementation.

Before the requirements engineering process starts, a company may carry out a feasibility or marketing study to assess whether or not there is a need or a market for the software and whether or not it is technically and financially realistic to develop the software required. Feasibility studies are short-term, relatively cheap studies that inform the decision of whether or not to go ahead with a more detailed analysis.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

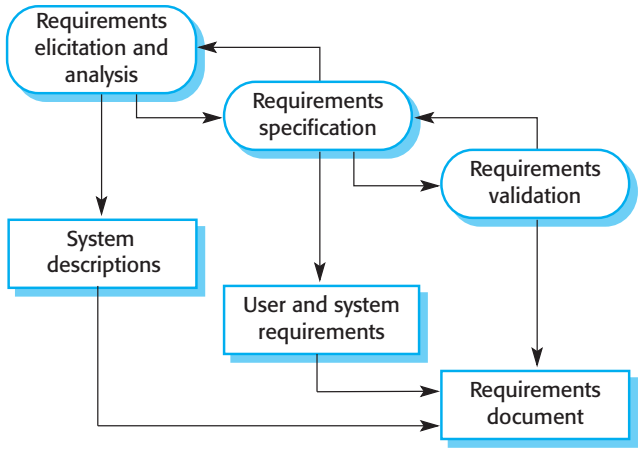


Figure 2.4 The requirements engineering process

There are three main activities in the requirements engineering process:

1. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
2. *Requirements specification* Requirements specification is the activity of translating the information gathered during requirements analysis into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.
3. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved.

In agile methods, requirements specification is not a separate activity but is seen as part of system development. Requirements are informally specified for each increment of the system just before that increment is developed. Requirements are specified according to user priorities. The elicitation of requirements comes from users who are part of or work closely with the development team.

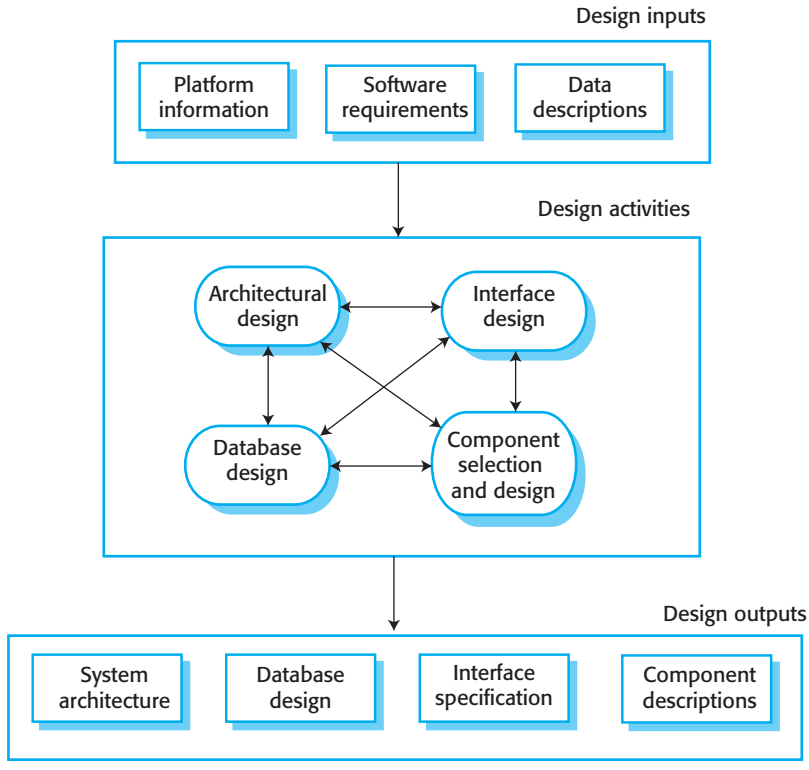


Figure 2.5 A general model of the design process

2.2.2 Software design and implementation

The implementation stage of software development is the process of developing an executable system for delivery to the customer. Sometimes this involves separate activities of software design and programming. However, if an agile approach to development is used, design and implementation are interleaved, with no formal design documents produced during the process. Of course, the software is still designed, but the design is recorded informally on whiteboards and programmer's notebooks.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design in stages. They add detail as they develop their design, with constant backtracking to modify earlier designs.

Figure 2.5 is an abstract model of the design process showing the inputs to the design process, process activities, and the process outputs. The design process activities are both interleaved and interdependent. New information about the design is constantly being generated, and this affects previous design decisions. Design rework is therefore inevitable.

Most software interfaces with other software systems. These other systems include the operating system, database, middleware, and other application systems. These make up the "software platform,' the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with its environment. If the system is to process existing data, then the description of that data may be included in the platform specification. Otherwise, the data description must be an input to the design process so that the system data organization can be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require an additional stage of timing design but may not include a database, so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. *Architectural design,* where you identify the overall structure of the system, the principal components (sometimes called subsystems or modules), their relationships, and how they are distributed.
2. *Database design,* where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.
3. *Interface design,* where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component may be used by other components without them having to know how it is implemented. Once interface specifications are agreed, the components can be separately designed and developed.
4. *Component selection and design,* where you search for reusable components and, if no suitable components are available, design new software components. The design at this stage may be a simple component description with the implementation details left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model expressed in the UML. The design model may then be used to automatically generate an implementation.

These activities lead to the design outputs, which are also shown in Figure 2.5. For critical systems, the outputs of the design process are detailed design documents setting out precise and accurate descriptions of the system. If a model-driven approach is used (Chapter 5), the design outputs are design diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

The development of a program to implement a system follows naturally from system design. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to

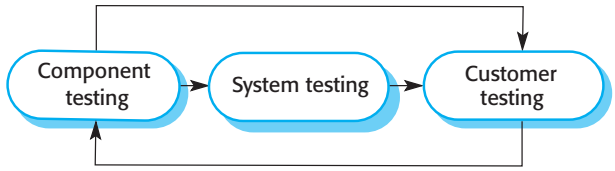


Figure 2.6 Stages of testing

define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is an individual activity, and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process and then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects (bugs) that must be removed from the program. Finding and fixing program defects is called *debugging*. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program and then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, are usually used to support the debugging process.

2.2.3 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system both conforms to its specification and meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. However, most V & V time and effort is spent on program testing.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are individually tested, then the integrated system is tested. For custom software, customer testing involves testing the system with real customer data. For products that are sold as applications, customer testing is sometimes called beta testing where selected users try out and comment on the software.

The stages in the testing process are:

1. *Component testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes or may be coherent groupings of these entities. Test automation tools, such as JUnit for Java, that can rerun tests when new versions of the component are created, are commonly used (Koskela 2013).
2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multistage process where components are integrated to form subsystems that are individually tested before these subsystems are integrated to form the final system.
3. *Customer testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested by the system customer (or potential customer) rather than with simulated test data. For custom-built software, customer testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Customer testing may also reveal requirements problems where the system's facilities do not really meet the users' needs or the system performance is unacceptable. For products, customer testing shows how well the software product meets the customer's needs.

Ideally, component defects are discovered early in the testing process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged, and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

Normally, component testing is simply part of the normal development process. Programmers make up their own test data and incrementally test the code as it is developed. The programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In test-driven development, which is a normal part of agile processes, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works

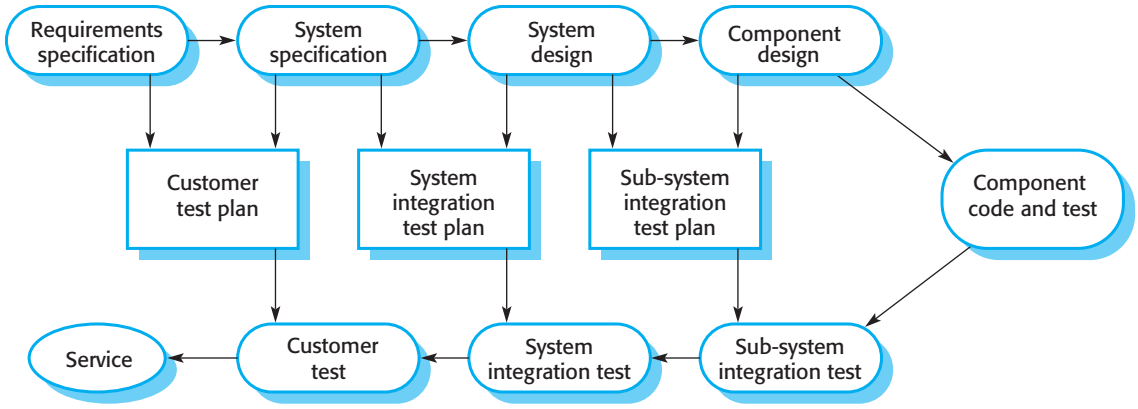


Figure 2.7 Testing phases in a plan-driven software process

from these test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V). The V-model shows the software validation activities that correspond to each stage of the waterfall process model.

When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the product developers. After this feedback, the software product may be modified and released for further beta testing or general sale.

2.2.4 Software evolution

The flexibility of software is one of the main reasons why more and more software is being incorporated into large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. They think that software maintenance is less interesting and challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Very few software systems are completely new systems, and it makes much more

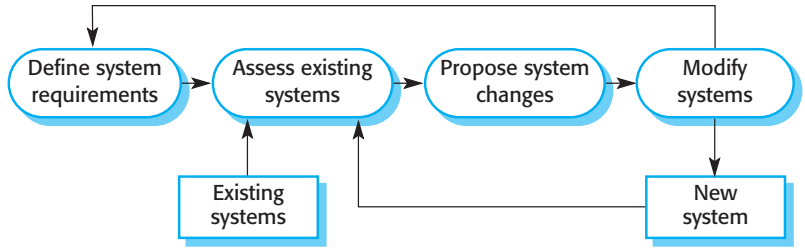


Figure 2.8 Software system evolution

sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as businesses respond to external pressures, competition, and changed management priorities. As new technologies become available, new approaches to design and implementation become possible. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and retest the system.

Two related approaches may be used to reduce the costs of rework:

1. *Change anticipation,* where the software process includes activities that can anticipate or predict possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.

2. *Change tolerance,* where the process and software are designed so that changes can be easily made to the system. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements:

1. *System prototyping,* where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This is a method of change anticipation as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.
2. *Incremental delivery,* where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely, improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3 (Agile methods).

2.3.1 Prototyping

A prototype is an early version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore software solutions and in the development of a user interface for the system.

System prototypes allow potential users to see how well the system supports their work. They may get new ideas for requirements and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the system requirements. A feature described in a specification may seem to be clear and useful. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification can then be modified to reflect the changed understanding of the requirements.

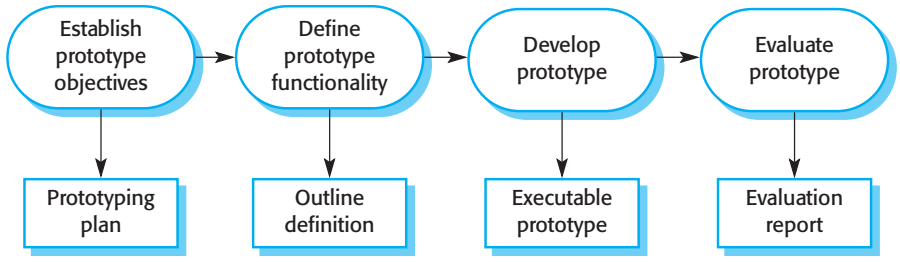


Figure 2.9 Prototype development

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Rapid prototyping with end-user involvement is the only sensible way to develop user interfaces. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements and design.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the application to managers. The same prototype usually cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training, and the prototype objectives should be used to derive a plan for evaluation. Potential users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions. A general problem with prototyping is that users may not use the prototype in the same way as they use the final system. Prototype testers may not be typical of system users. There may not be enough time to train users during prototype evaluation. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

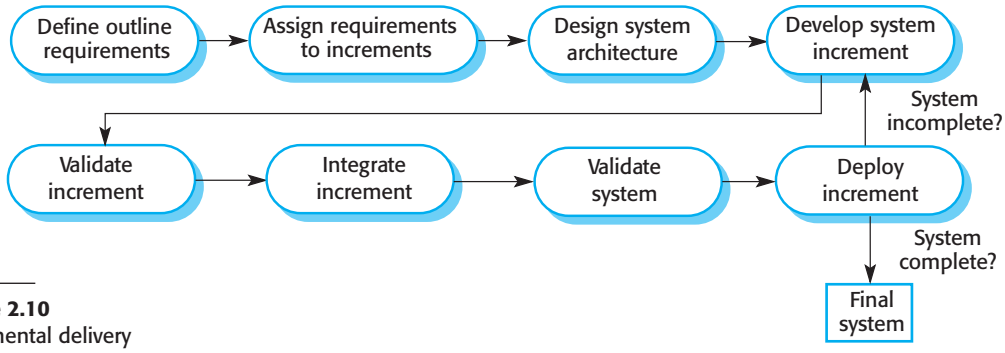


Figure 2.10
Incremental delivery

2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in their working environment. In an incremental delivery process, customers define which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a subset of the system functionality. The allocation of services to increments depends on the service priority, with the highest priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, it is installed in the customer's normal working environment. They can experiment with the system, and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system, so there is no relearning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements, so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.

4. As the highest priority services are delivered first and later increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery. In practice, it only works in situations where a brand-new system is being introduced and the system evaluators are given time to experiment with the new system. Key problems with this approach are:

1. Iterative delivery is problematic when the new system is intended to replace an existing system. Users need all of the functionality of the old system and are usually unwilling to experiment with an incomplete new system. It is often impractical to use the old and the new systems alongside each other as they are likely to have different databases and user interfaces.
2. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

For some types of systems, incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development, and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These large systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a system prototype may be developed and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

2.4 Process improvement

Nowadays, there is a constant demand from industry for cheaper, better software, which has to be delivered to ever-tighter deadlines. Consequently, many software companies have turned to software process improvement as a way of enhancing the

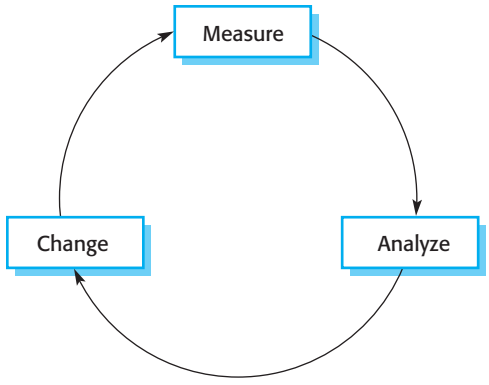


Figure 2.11 The process improvement cycle

quality of their software, reducing costs, or accelerating their development processes. Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. I cover general issues of process measurement and process improvement in detail in web Chapter 26.

Two quite different approaches to process improvement and change are used:

1. The process maturity approach, which has focused on improving process and project management and introducing good software engineering practice into an organization. The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes. The primary goals of this approach are improved product quality and process predictability.
2. The agile approach, which has focused on iterative development and the reduction of overheads in the software process. The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements. The improvement philosophy here is that the best processes are those with the lowest overheads and agile approaches can achieve this. I describe agile approaches in Chapter 3.

People who are enthusiastic about and committed to each of these approaches are generally skeptical of the benefits of the other. The process maturity approach is rooted in plan-driven development and usually requires increased "overhead," in the sense that activities are introduced that are not directly relevant to program development. Agile approaches focus on the code being developed and deliberately minimize formality and documentation.

The general process improvement process underlying the process maturity approach is a cyclical process, as shown in Figure 2.11. The stages in this process are:

1. *Process measurement* You measure one or more attributes of the software process or product. These measurements form a baseline that helps you decide if

process improvements have been effective. As you introduce improvements, you re-measure the same attributes, which will hopefully have improved in some way.

2. *Process analysis* The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed during this stage. The analysis may be focused by considering process characteristics such as rapidity and robustness.

3. *Process change* Process changes are proposed to address some of the identified process weaknesses. These are introduced, and the cycle resumes to collect data about the effectiveness of the changes.

Without concrete data on a process or the software developed using that process, it is impossible to assess the value of process improvement. However, companies starting the process improvement process are unlikely to have process data available as an improvement baseline. Therefore, as part of the first cycle of changes, you may have to collect data about the software process and to measure software product characteristics.

Process improvement is a long-term activity, so each of the stages in the improvement process may last several months. It is also a continuous activity as, whatever new processes are introduced, the business environment will change and the new processes will themselves have to evolve to take these changes into account.

The notion of process maturity was introduced in the late 1980s when the Software Engineering Institute (SEI) proposed their model of process capability maturity (Humphrey 1988). The maturity of a software company's processes reflects the process management, measurement, and use of good software engineering practices in the company. This idea was introduced so that the U.S. Department of Defense could assess the software engineering capability of defense contractors, with a view to limiting contracts to those contractors who had reached a required level of process maturity. Five levels of process maturity were proposed. as shown in Figure 2.12. These have evolved and developed over the last 25 years (Chrissis, Konrad, and Shrum 2011), but the fundamental ideas in Humphrey's model are still the basis of software process maturity assessment.

The levels in the process maturity model are:

1. *Initial* The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.

2. *Managed* At this level, the goals associated with the process area are met, and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.

3. *Defined* This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.

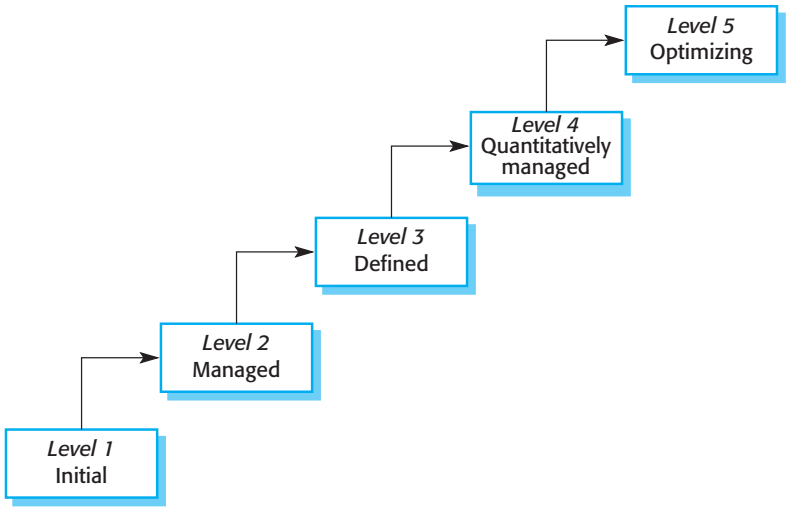


Figure 2.12 Capability maturity levels

4. *Quantitatively managed* At this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses. That is, collected process and product measurements must be used in process management.

5. *Optimizing* At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed and the processes adapted to changing business needs.

The work on process maturity levels has had a major impact on the software industry. It focused attention on the software engineering processes and practices that were used and led to significant improvements in software engineering capability. However, there is too much overhead in formal process improvement for small companies, and maturity estimation with agile processes is difficult. Consequently, only large software companies now use this maturity-focused approach to software process improvement.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reusable component configuration and integration.

- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous, and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- Process improvement is the process of improving existing software processes to improve software quality, lower development costs, or reduce development time. It is a cyclic process involving process measurement, analysis, and change.

FURTHER READING

"Process Models in Software Engineering." This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering,* ed. J. J. Marciniak, John Wiley & Sons, 2001) http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf

Software Process Improvement: Results and Experience from the Field. This book is a collection of papers focusing on process improvement case studies in several small and medium-sized Norwegian companies. It also includes a good introduction to the general issues of process improvement. (Conradi, R., Dybå, T., Sjøberg, D., and Ulsund, T. (eds.), Springer, 2006).

"Software Development Life Cycle Models and Methodologies." This blog post is a succinct summary of several software process models that have been proposed and used. It discusses the advantages and disadvantages of each of these models (M. Sami, 2012). http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-engineering/

EXERCISES

2.1. Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:

A system to control antilock braking in a car

A virtual reality system to support software maintenance

A university accounting system that replaces an existing system

An interactive travel planning system that helps users plan journeys with the lowest environmental impact

2.2. Incremental software development could be very effectively used for customers who do not have a clear idea about the systems needed for their operations. Discuss.

2.3. Consider the integration and configuration process model shown in Figure 2.3. Explain why it is essential to repeat the requirements engineering activity in the process.

2.4. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.

2.5. Using an example, explain why the design activities of architectural design, database design, interface design, and component design are interdependent.

2.6. Explain why software testing should always be an incremental, staged activity. Are programmers the best people to test the programs that they have developed?

2.7. Imagine that a government wants a software program that helps to keep track of the utilization of the country's vast mineral resources. Although the requirements put forward by the government were not very clear, a software company was tasked with the development of a prototype. The government found the prototype impressive, and asked it be extended to be the actual system that would be used. Discuss the pros and cons of taking this approach.

2.8. You have developed a prototype of a software system and your manager is very impressed by it. She proposes that it should be put into use as a production system, with new features added as required. This avoids the expense of system development and makes the system immediately useful. Write a short report for your manager explaining why prototype systems should not normally be used as production systems.

2.9. Suggest two advantages and two disadvantages of the approach to process assessment and improvement that is embodied in the SEI's Capability Maturity framework.

2.10. Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

REFERENCES

Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.

———. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, 21 (5), 61–72. doi:10.1145/12944.12948

Boehm, B. W., and R. Turner. 2004. "Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods." In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd ed.* Boston: Addison-Wesley.

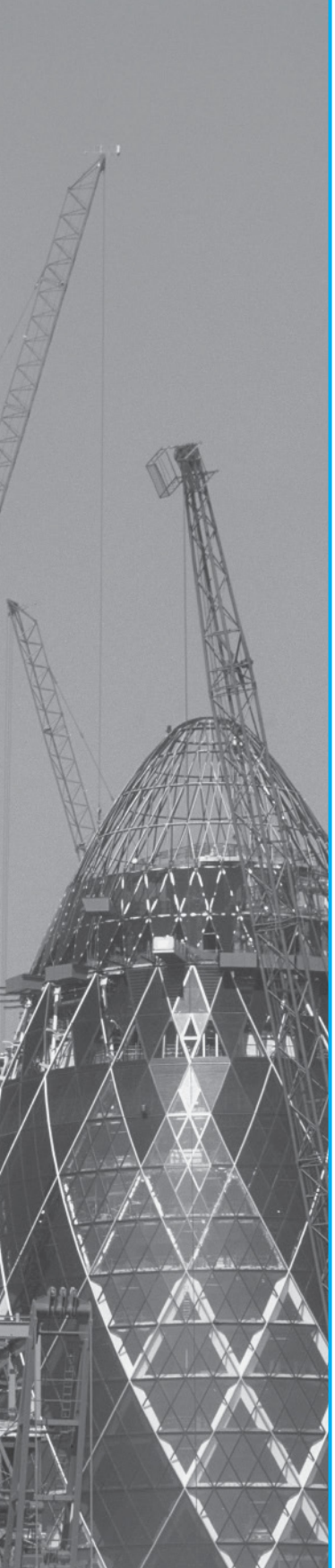
Humphrey, W. S. 1988. "Characterizing the Software Process: A Maturity Framework." *IEEE Software* 5 (2): 73–79. doi:10.1109/2.59.

Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.

Krutchen, P. 2003. *The Rational Unified Process—An Introduction, 3rd ed.* Reading, MA: Addison-Wesley.

Royce, W. W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques." In *IEEE WESTCON*, 1–9. Los Angeles, CA.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.



3 Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know about important agile development practices such as user stories, refactoring, pair programming and test-first development;
- understand the Scrum approach to agile project management;
- understand the issues of scaling agile development methods and combining agile approaches with plan-driven approaches in the development of large software systems.

Contents

3.1 Agile methods
3.2 Agile development techniques
3.3 Agile project management
3.4 Scaling agile methods

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions and the emergence of competing products and services. Software is part of almost all business operations, so new software has to be developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid software development and delivery is therefore the most critical requirement for most business systems. In fact, businesses may be willing to trade off software quality and compromise on requirements if they can deploy essential new software quickly.

Because these businesses are operating in a changing environment, it is practically impossible to derive a complete set of stable software requirements. Requirements change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, external factors drive requirements change.

Plan-driven software development processes that completely specify the requirements and then design, build, and test a system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually a lengthy one, and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, this plan-driven approach is the right one. However, in a fast-moving business environment, it can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid software development and processes that can handle changing requirements has been recognized for many years (Larman and Basili 2003). However, faster software development really took off in the late 1990s with the development of the idea of "agile methods" such as Extreme Programming (Beck 1999), Scrum (Schwaber and Beedle 2001), and DSDM (Stapleton 2003).

Rapid software development became known as agile development or agile methods. These agile methods are designed to produce useful software quickly. All of the agile methods that have been proposed share a number of common characteristics:

1. The processes of specification, design and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document is an outline definition of the most important characteristics of the system.

2. The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment.

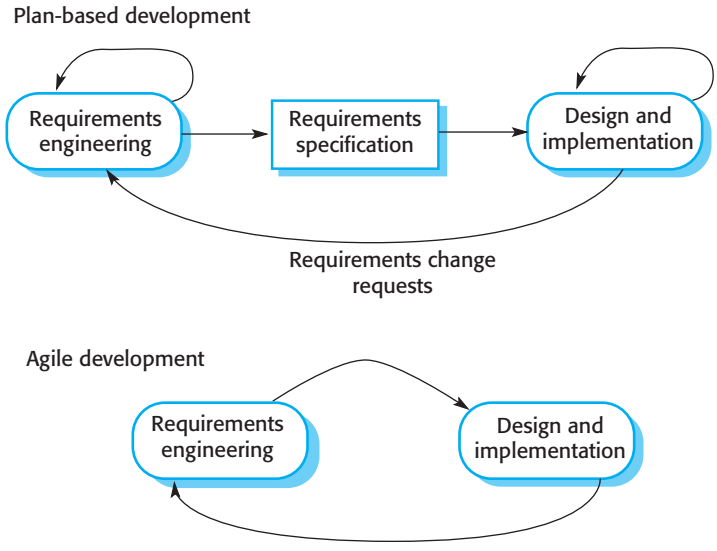


Figure 3.1 Plan-driven and agile development

They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. Extensive tool support is used to support the development process. Tools that may be used include automated testing tools, tools to support configuration management, and system integration and tools to automate user interface production.

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity.

Figure 3.1 shows the essential distinctions between plan-driven and agile approaches to system specification. In a plan-driven software development process, iteration occurs within activities, with formal documents used to communicate between stages of the process. For example, the requirements will evolve, and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together rather than separately.

In practice, as I explain in Section 3.4.1, plan-driven processes are often used along with agile programming practices, and agile methods may incorporate some planned

activities apart from programming and testing. It is perfectly feasible, in a plan-driven process, to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused, and it may produce some design documentation. Agile developers may decide that an iteration should not produce new code but rather should produce system models and documentation.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, use of analysis and design methods supported by software tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This plan-driven approach was developed for software developed by large teams, working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. Plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design have to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led to the development of agile methods in the late 1990s. These methods allowed the development team to focus on the software itself rather than on its design and documentation. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto (http://agilemanifesto.org) issued by the leading developers of these methods. This manifesto states:

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more[†].

All agile methods suggest that software should be developed and delivered incrementally. These methods are based on different agile processes but they share a set of principles, based on the agile manifesto, and so they have much in common. I have listed these principles in Figure 3.2.

Agile methods have been particularly successful for two kinds of system development.

1. Product development where a software company is developing a small or medium-sized product for sale. Virtually all software products and apps are now developed using an agile approach.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external stakeholders and regulations that affect the software.

Agile methods work well in these situations because it is possible to have continuous communications between the product manager or system customer and the development team. The software itself is a stand-alone system rather than tightly integrated with other systems being developed at the same time. Consequently, there is no need to coordinate parallel development streams. Small and medium-sized

[†]http://agilemanifesto.org/

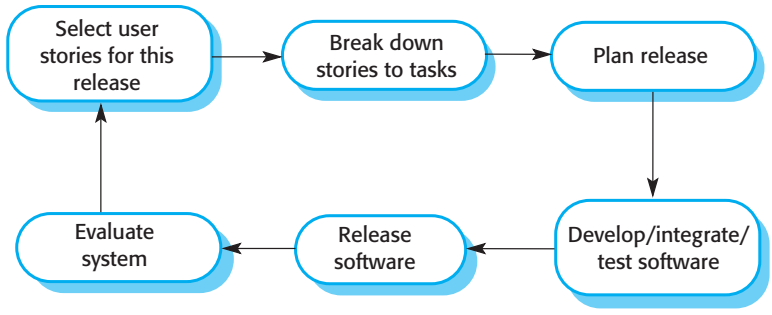


Figure 3.3 The XP release cycle

systems can be developed by co-located teams, so informal communications among team members work well.

3.2 Agile development techniques

The ideas underlying agile methods were developed around the same time by a number of different people in the 1990s. However, perhaps the most significant approach to changing software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck (Beck 1998) because the approach was developed by pushing recognized good practice, such as iterative development, to "extreme" levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated, and tested in a day. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

In XP, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system.

Extreme programming was controversial as it introduced a number of agile practices that were quite different from the development practice of that time. These practices are summarized in Figure 3.4 and reflect the principles of the agile manifesto:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.

5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In practice, the application of Extreme Programming as originally proposed has proved to be more difficult than anticipated. It cannot be readily integrated with the management practices and culture of most businesses. Therefore, companies adopting agile methods pick and choose those XP practices that are most appropriate for their way of working. Sometimes these are incorporated into their own development processes but, more commonly, they are used in conjunction with a management-focused agile method such as Scrum (Rubin 2013).

Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose; If she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A "prescribing medication" story

I am not convinced that XP on its own is a practical agile method for most companies, but its most significant contribution is probably the set of agile development practices that it introduced to the community. I discuss the most important of these practices in this section.

3.2.1 User stories

Software requirements always change. To handle these changes, agile methods do not have a separate requirements engineering activity. Rather, they integrate requirements elicitation with development. To make this easier, the idea of "user stories" was developed where a user story is a scenario of use that might be experienced by a system user.

As far as possible, the system customer works closely with the development team and discusses these scenarios with other team members. Together, they develop a "story card" that briefly describes a story that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the Mentcare system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

User stories may be used in planning system iterations. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be

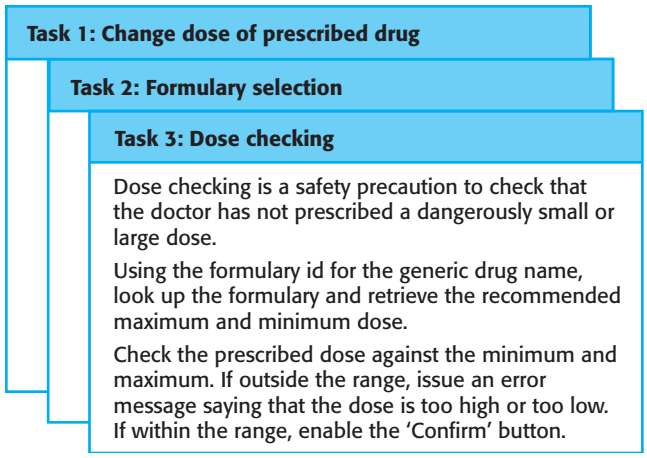


Figure 3.6 Examples of task cards for prescribing medication

used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

The idea of user stories is a powerful one—people find it much easier to relate to these stories than to a conventional requirements document or use cases. User stories can be helpful in getting users involved in suggesting requirements during an initial predevelopment requirements elicitation activity. I discuss this in more detail in Chapter 4.

The principal problem with user stories is completeness. It is difficult to judge if enough user stories have been developed to cover all of the essential requirements of a system. It is also difficult to judge if a single story gives a true picture of an activity. Experienced users are often so familiar with their work that they leave things out when describing it.

3.2.2 Refactoring

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. Often the changes anticipated never materialize, or completely different change requests may actually be made.

Of course, in practice, changes will always have to be made to the code being developed. To make these changes easier, the developers of XP suggested that the code being developed should be constantly refactored. Refactoring (Fowler et al. 1999) means that the programming team look for possible improvements to the software and implements

them immediately. When team members see code that can be improved, they make these improvements even in situations where there is no immediate need for them.

A fundamental problem of incremental development is that local changes tend to degrade the software structure. Consequently, further changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Refactoring improves the software structure and readability and so avoids the structural deterioration that naturally occurs when software is changed.

Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of similar code sections, with calls to methods defined in a program library. Program development environments usually include tools for refactoring. These simplify the process of finding dependencies between code sections and making global code modifications.

In principle, when refactoring is part of the development process, the software should always be easy to understand and change as new requirements are proposed. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require that the architecture of the system be modified.

3.2.3 Test-first development

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

Extreme Programming developed a new approach to program testing to address the difficulties of testing without a specification. Testing is automated and is central to the development process, and development cannot proceed until all tests have been successfully executed. The key features of testing in XP are:

1. test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

XP's test-first philosophy has now evolved into more general test-driven development techniques (Jeffries and Melnik 2007). I believe that test-driven development is one of the most important innovations in software engineering. Instead of writing code and then writing tests for that code, you write the tests before you write the code. This

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

means that you can run the test as the code is being written and discover problems during development. I discuss test-driven development in more depth in Chapter 8.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. Test-first development requires there to be a clear relationship between system requirements and the code implementing the corresponding requirements. In XP, this relationship is clear because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of "test-lag." This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

XP's test-first approach assumes that user stories have been developed, and these have been broken down into a set of task cards, as shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I explain in Chapter 8, acceptance testing is the process whereby the system is tested using customer data to check that it meets the customer's real needs.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be stand-alone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Tahchiev et al. 2010) is a widely used example of an automated testing framework for Java programs.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually result in a large number of tests being written and executed. However, there are problems in ensuring that test coverage is complete:

1. Programmers prefer programming to testing, and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the "display logic" and workflow between screens.

It is difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so will remain untested. Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests are written after development, then undetected bugs may be delivered in the system release.

3.2.4 Pair programming

Another innovative practice that was introduced in XP is that programmers work in pairs to develop the software. The programming pair sits at the same computer to develop the software. However, the same pair do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

Pair programming has a number of advantages.

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's idea of egoless programming (Weinberg 1971) where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (Chapter 24) are effective in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Pair programming is a less formal process that probably doesn't find as many errors as code inspections. However, it is cheaper and easier to organize than formal program inspections.
3. It encourages refactoring to improve the software structure. The problem with asking programmers to refactor in a normal development environment is that effort

involved is expended for long-term benefit. An developer who spends time refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. Many companies that have adopted agile methods are suspicious of pair programming and do not use it. Other companies mix pair and individual programming with an experienced programmer working with a less experienced colleague when they have problems.

Formal studies of the value of pair programming have had mixed results. Using student volunteers, Williams and her collaborators (Williams et al. 2000) found that productivity with pair programming seems to be comparable to that of two people working independently. The reasons suggested are that pairs discuss the software before development and so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers did not replicate these results (Arisholm et al. 2007). They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits, but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.3 Agile project management

In any software business, managers need to know what is going on and whether or not a project is likely to meet its objectives and deliver the software on time with the proposed budget. Plan-driven approaches to software development evolved to meet this need. As I discussed in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach requires a manager to have a stable view of everything that has to be developed and the development processes.

The informal planning and project control that was proposed by the early adherents of agile methods clashed with this business requirement for visibility. Teams were self-organizing, did not produce documentation, and planned development in very short cycles. While this can and does work for small companies developing software products, it is inappropriate for larger companies who need to know what is going on in their organization.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be "potentially shippable," which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of "to do" items that the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2 to 4 weeks long.
Velocity	An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Figure 3.8 Scrum terminology

the team. To address this issue, the Scrum agile method was developed (Schwaber and Beedle 2001; Rubin 2013) to provide a framework for organizing agile projects and, to some extent at least, provide external visibility of what is going on. The developers of Scrum wished to make clear that Scrum was not a method for project management in the conventional sense, so they deliberately invented new terminology, such as ScrumMaster, which replaced names such as project manager. Figure 3.8 summarizes Scrum terminology and what it means.

Scrum is an agile method insofar as it follows the principles from the agile manifesto, which I showed in Figure 3.2. However, it focuses on providing a framework for agile project organization, and it does not mandate the use of specific development

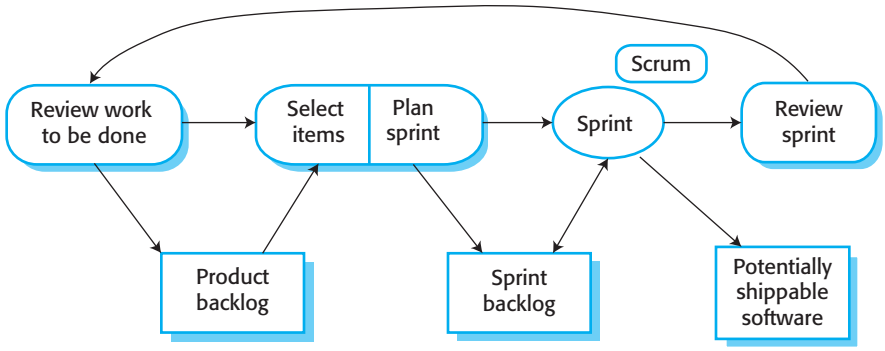


Figure 3.9 The Scrum sprint cycle

practices such as pair programming and test-first development. This means that it can be more easily integrated with existing practice in a company. Consequently, as agile methods have become a mainstream approach to software development, Scrum has emerged as the most widely used method.

The Scrum process or sprint cycle is shown in Figure 3.9. The input to the process is the product backlog. Each process iteration produces a product increment that could be delivered to customers.

The starting point for the Scrum sprint cycle is the product backlog—the list of items such as product features, requirements, and engineering improvement that have to be worked on by the Scrum team. The initial version of the product backlog may be derived from a requirements document, a list of user stories, or other description of the software to be developed.

While the majority of entries in the product backlog are concerned with the implementation of system features, other activities may also be included. Sometimes, when planning an iteration, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. There may also be backlog items to design the system architecture or to develop system documentation.

The product backlog may be specified at varying levels of detail, and it is the responsibility of the Product Owner to ensure that the level of detail in the specification is appropriate for the work to be done. For example, a backlog item could be a complete user story such as that shown in Figure 3.5, or it could simply be an instruction such as "Refactor user interface code" that leaves it up to the team to decide on the refactoring to be done.

Each sprint cycle lasts a fixed length of time, which is usually between 2 and 4 weeks. At the beginning of each cycle, the Product Owner prioritizes the items on the product backlog to define which are the most important items to be developed in that cycle. Sprints are never extended to take account of unfinished work. Items are returned to the product backlog if these cannot be completed within the allocated time for the sprint.

The whole team is then involved in selecting which of the highest priority items they believe can be completed. They then estimate the time required to complete these items. To make these estimates, they use the velocity attained in previous

sprints, that is, how much of the backlog could be covered in a single sprint. This leads to the creation of a sprint backlog—the work to be done during that sprint. The team self-organizes to decide who will work on what, and the sprint begins.

During the sprint, the team holds short daily meetings (Scrums) to review progress and, where necessary, to re-prioritize work. During the Scrum, all team members share information, describe their progress since the last meeting, bring up problems that have arisen, and state what is planned for the following day. Thus, everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them. Everyone participates in this short-term planning; there is no top-down direction from the ScrumMaster.

The daily interactions among Scrum teams may be coordinated using a Scrum board. This is an office whiteboard that includes information and post-it notes about the Sprint backlog, work done, unavailability of staff, and so on. This is a shared resource for the whole team, and anyone can change or move items on the board. It means that any team member can, at a glance, see what others are doing and what work remains to be done.

At the end of each sprint, there is a review meeting, which involves the whole team. This meeting has two purposes. First, it is a means of process improvement. The team reviews the way they have worked and reflects on how things could have been done better. Second, it provides input on the product and the product state for the product backlog review that precedes the next sprint.

While the ScrumMaster is not formally a project manager, in practice ScrumMasters take this role in many organizations that have a conventional management structure. They report on progress to senior management and are involved in longer-term planning and project budgeting. They may be involved in project administration (agreeing on holidays for staff, liaising with HR, etc.) and hardware and software purchases.

In various Scrum success stories (Schatz and Abdelshafi 2005; Mulder and van Vliet 2008; Bellouiti 2009), the things that users like about the Scrum method are:

1. The product is broken down into a set of manageable and understandable chunks that stakeholders can relate to.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything, and consequently team communication and morale are improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works. They are not faced with last-minute surprises when a team announces that software will not be delivered as expected.
5. Trust between customers and developers is established, and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams, with team members located in different places around the world. This allows companies to take advantage

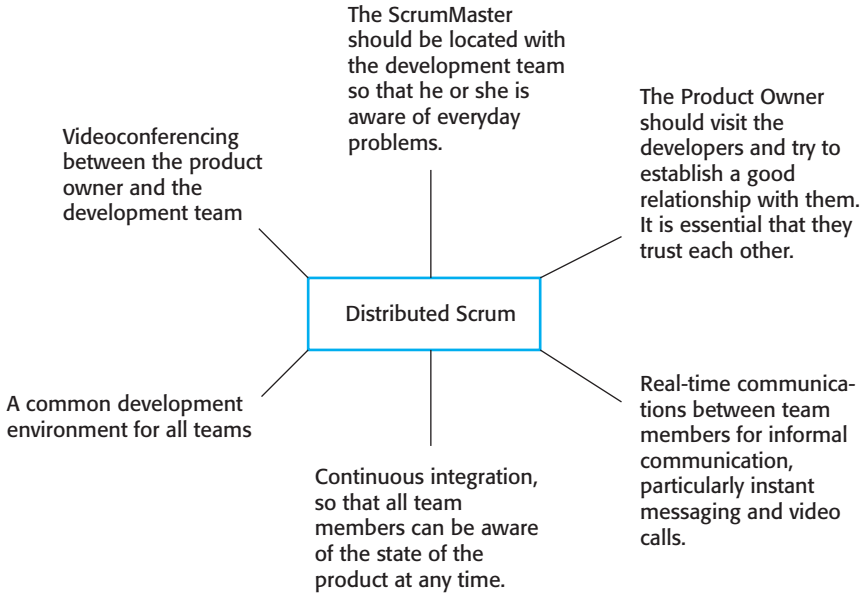


Figure 3.10 Distributed Scrum

of lower cost staff in other countries, makes access to specialist skills possible, and allows for 24-hour development, with work going on in different time zones.

Consequently, there have been developments of Scrum for distributed development environments and multi-team working. Typically, for offshore development, the product owner is in a different country from the development team, which may also be distributed. Figure 3.10 shows the requirements for Distributed Scrum (Deemer 2011).

3.4 Scaling agile methods

Agile methods were developed for use by small programming teams that could work together in the same room and communicate informally. They were originally used by for the development of small and medium-sized systems and software products. Small companies, without formal processes or bureaucracy, were enthusiastic initial adopters of these methods.

Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to both larger systems and larger companies. Consequently, over the past few years, a lot of work has been put into evolving agile methods for both large software systems and for use in large companies.

Scaling agile methods has closely related facets:

1. Scaling up these methods to handle the development of large systems that are too big to be developed by a single small team.
2. Scaling out these methods from specialized development teams to more widespread use in a large company that has many years of software development experience.

Of course, scaling up and scaling out are closely related. Contracts to develop large software systems are usually awarded to large organizations, with multiple teams working on the development project. These large companies have often experimented with agile methods in smaller projects, so they face the problems of scaling up and scaling out at the same time.

There are many anecdotes about the effectiveness of agile methods, and it has been suggested that these can lead to orders of magnitude improvements in productivity and comparable reductions in defects. Ambler (Ambler 2010), an influential agile method developer, suggests that these productivity improvements are exaggerated for large systems and organizations. He suggests that an organization moving to agile methods can expect to see productivity improvement across the organization of about 15% over 3 years, with similar reductions in the number of product defects.

3.4.1 Practical problems with agile methods

In some areas, particularly in the development of software products and apps, agile development has been incredibly successful. It is by far the best approach to use for this type of system. However, agile methods may not be suitable for other types of software development, such as embedded systems engineering or the development of large and complex systems.

For large, long-lifetime systems that are developed by a software company for an external client, using an agile approach presents a number of problems.

1. The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
2. Agile methods are most appropriate for new software development rather than for software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
3. Agile methods are designed for small co-located teams, yet much software development now involves worldwide distributed teams.

Contractual issues can be a major problem when agile methods are used. When the system customer uses an outside organization for system development, a contract for the software development is drawn up between them. The software requirements document is usually part of that contract between the customer and the supplier. Because the interleaved development of requirements and code is fundamental to agile methods, there is no definitive statement of requirements that can be included in the contract.

Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. As long as all goes well, this benefits both the customer and the developer. However, if problems arise, then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

As I explain in Chapter 9, a huge amount of software engineering effort goes into the maintenance and evolution of existing software systems. Agile practices, such as incremental delivery, design for change, and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process that supports continual change. If agile methods are used for software product development, new releases of the product or app simply involve continuing the agile approach.

However, where maintenance involves a custom system that must be changed in response to new business requirements, there is no clear consensus on the suitability of agile methods for software maintenance (Bird 2011; Kilner 2012). Three types of problems can arise:

- lack of product documentation
- keeping customers involved
- development team continuity

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is rarely updated and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. The lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance is that the most important document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. The use of agile methods may therefore make subsequent system maintenance more difficult and expensive. This is a particular problem if development team continuity cannot be maintained.

A key challenge in using an agile approach to maintenance is keeping customers involved in the process. While a customer may be able to justify the full-time involvement of a representative during system development, this is less likely during maintenance where changes are not continuous. Customer representatives are likely to lose interest in the system. Therefore, it is likely that alternative mechanisms, such as change proposals, discussed in Chapter 25, will have to be adapted to fit in with an agile approach.

Another potential problem that may arise is maintaining continuity of the development team. Agile methods rely on team members understanding aspects of the system without having to consult documentation. If an agile development team is broken up, then this implicit knowledge is lost and it is difficult for new team members to build up the same understanding of the system and its components. Many programmers prefer to work on new development to software maintenance, and so they are unwilling to continue to work on a software system after the first release has been delivered. Therefore, even when the intention is to keep the development team together, people leave if they are assigned maintenance tasks.

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know product features several months in advance to prepare an effective marketing campaign.
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People, not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods and therefore may not interact well with other team members.

Figure 3.11 Agile principles and organizational practice

3.4.2 Agile and plan-driven methods

A fundamental requirement of scaling agile methods is to integrate them with plan-driven approaches. Small startup companies can work with informal and short-term planning, but larger companies have to have longer-term plans and budgets for investment, staffing, and business development. Their software development must support these plans, so longer-term software planning is essential.

Early adopters of agile methods in the first decade of the 21st century were enthusiasts and deeply committed to the agile manifesto. They deliberately rejected the plan-driven approach to software engineering and were reluctant to change the initial vision of agile methods in any way. However, as organizations saw the value and benefits of an agile approach, they adapted these methods to suit their own culture and ways of working. They had to do this because the principles underlying agile methods are sometimes difficult to realize in practice (Figure 3.11).

To address these problems, most large "agile" software development projects combine practices from plan-driven and agile approaches. Some are mostly agile, and others are mostly plan-driven but with some agile practices. To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human and organizational questions. These relate to the system being developed, the development team, and the organizations that are developing and procuring the system (Figure 3.12).

Agile methods were developed and refined in projects to develop small to medium-sized business systems and software products, where the software developer controls the specification of the system. Other types of system have attributes such as size, complexity, real-time response, and external regulation that mean a "pure" agile approach is

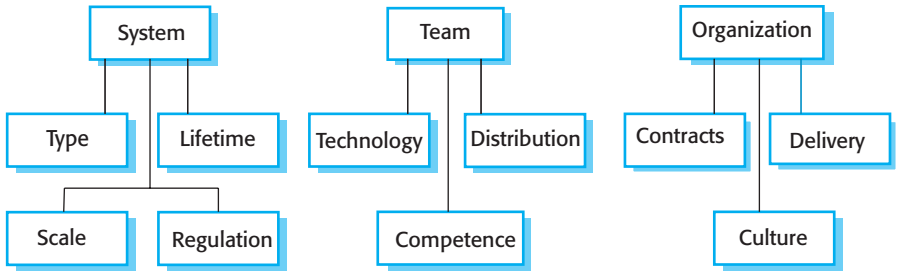


Figure 3.12 Factors influencing the choice of plan-based or agile development

unlikely to work. There needs to be some up-front planning, design, and documentation in the systems engineering process. Some of the key issues are as follows:

1. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a relatively small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams, so a plan-driven approach may have to be used.
2. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
3. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and is not of much use for long-term system maintenance.
4. Is the system subject to external regulation? If a system has to be approved by an external regulator (e.g., the Federal Aviation Administration approves software that is critical to the operation of an aircraft), then you will probably be required to produce detailed documentation as part of the system safety case.

Agile methods place a great deal of responsibility on the development team to cooperate and communicate during the development of the system. They rely on individual engineering skills and software support for the development process. However, in reality, not everyone is a highly skilled engineer, people do not communicate effectively, and it is not always possible for teams to work together. Some planning may be required to make the most effective use of the people available. Key issues are:

1. How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code. If you have a team with relatively low skill levels, you may need to use the best people to develop the design, with others responsible for programming.

2. How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.
3. What technologies are available to support system development? Agile methods often rely on good tools to keep track of an evolving design. If you are developing a system using an IDE that does not have good tools for program visualization and analysis, then more design documentation may be required.

Television and films have created a popular vision of software companies as informal organizations run by young men (mostly) who provide a fashionable working environment, with a minimum of bureaucracy and organizational procedures. This is far from the truth. Most software is developed in large companies that have established their own working practices and procedures. Management in these companies may be uncomfortable with the lack of documentation and the informal decision making in agile methods. Key issues are:

1. Is it important to have a very detailed specification and design before moving to implementation, perhaps for contractual reasons? If so, you probably need to use a plan-driven approach for requirements engineering but may use agile development practices during system implementation.
2. Is an incremental delivery strategy, where you deliver the software to customers or other system stakeholders and get rapid feedback from them, realistic? Will customer representatives be available, and are they willing to participate in the development team?
3. Are there cultural issues that may affect system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering. This usually requires extensive design documentation rather than the informal knowledge used in agile processes.

In reality, the issue of whether a project can be labeled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. Software developers should be pragmatic and should choose those methods that are most effective for the type of system being developed, whether or not these are labeled agile or plan-driven.

3.4.3 Agile methods for large systems

Agile methods have to evolve to be used for large-scale software development. The fundamental reason for this is that large-scale software systems are much more complex and difficult to understand and manage than small-scale systems or software products. Six principal factors (Figure 3.13) contribute to this complexity:

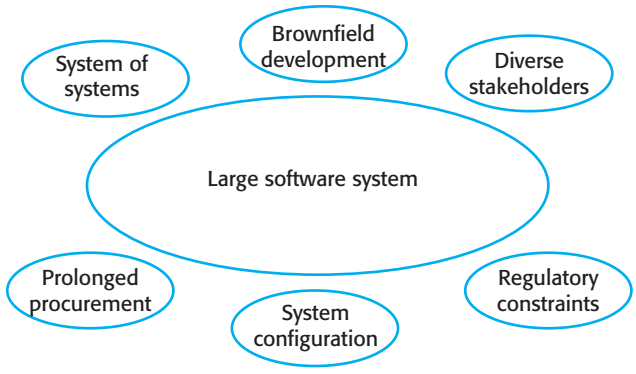


Figure 3.13 Large project characteristics

1. Large systems are usually systems of systems—collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.

2. Large systems are brownfield systems (Hopkins and Jenkins 2008); that is, they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.

3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.

4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, and so on. Customers may have specific compliance requirements that may have to be followed, and these may require process documentation to be completed.

5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.

6. Large systems usually have a diverse set of stakeholders with different perspectives and objectives. For example, nurses and administrators may be the end-users of a medical system, but senior medical staff, hospital managers, and others, are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

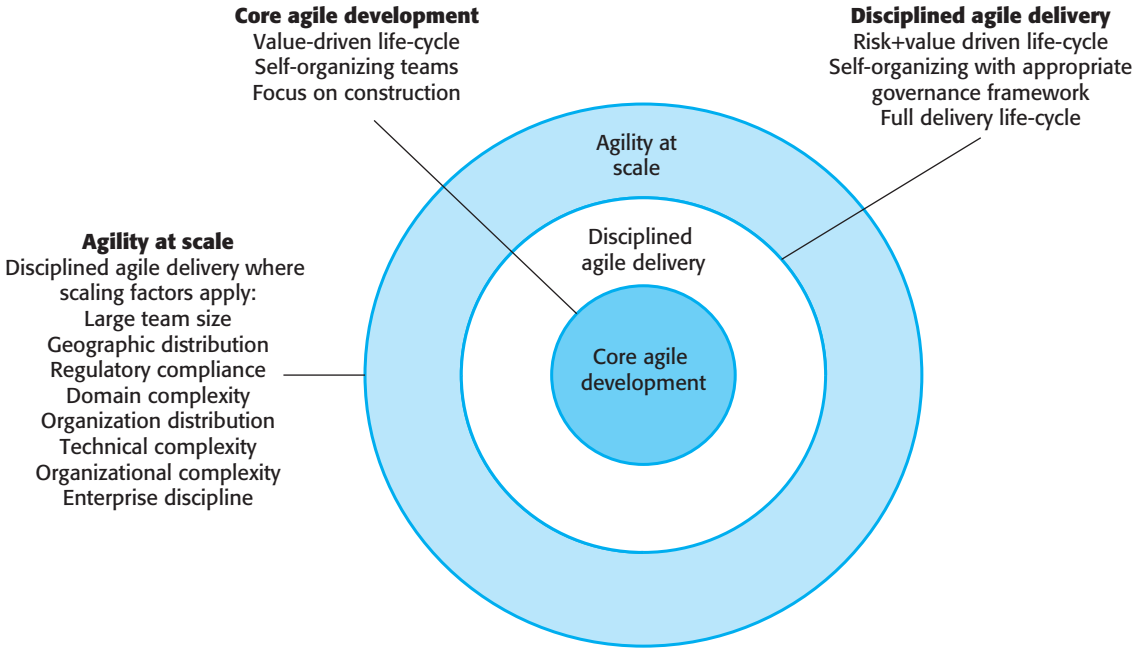


Figure 3.14 IBM's Agility at Scale model (© IBM 2010)

Dean Leffingwell, who has a great deal of experience in scaling agile methods, has developed the Scaled Agile Framework (Leffingwell 2007, 2011) to support large-scale, multi-team software development. He reports how this method has been used successfully in a number of large companies. IBM has also developed a framework for the large-scale use of agile methods called the Agile Scaling Model (ASM). Figure 3.14, taken from Ambler's white paper that discusses ASM (Ambler 2010), shows an overview of this model.

The ASM recognizes that scaling is a staged process where development teams move from the core agile practices discussed here to what is called Disciplined Agile Delivery. Essentially, this stage involves adapting these practices to a disciplined organizational setting and recognizing that teams cannot simply focus on development but must also take into account other stages of the software engineering process, such as requirements and architectural design.

The final scaling stage in ASM is to move to Agility at Scale where the complexity that is inherent in large projects is recognized. This involves taking account of factors such as distributed development, complex legacy environments, and regulatory compliance requirements. The practices used for disciplined agile delivery may have to be modified on a project-by-project basis to take these into account and, sometimes, additional plan-based practices added to the process.

No single model is appropriate for all large-scale agile products as the type of product, the customer requirements, and the people available are all different. However, approaches to scaling agile methods have a number of things in common:

1. A completely incremental approach to requirements engineering is impossible. Some early work on initial software requirements is essential. You need this work to identify the different parts of the system that may be developed by different teams and, often, to be part of the contract for the system development. However, these requirements should not normally be specified in detail; details are best developed incrementally.
2. There cannot be a single product owner or customer representative. Different people have to be involved for different parts of the system, and they have to continuously communicate and negotiate throughout the development process.
3. It is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed, and there has to be documentation produced to describe critical aspects of the system, such as database schemas and the work breakdown across teams.
4. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and videoconferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as email, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
5. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. Configuration management tools that support multi-team software development are essential.

Scrum has been adapted for large-scale development. In essence, the Scrum team model described in Section 3.3 is maintained, but multiple Scrum teams are set up. The key characteristics of multi-team Scrum are:

1. *Role replication* Each team has a Product Owner for its work component and ScrumMaster. There may be a chief Product Owner and ScrumMaster for the entire project.
2. *Product architects* Each team chooses a product architect, and these architects collaborate to design and evolve the overall system architecture.
3. *Release alignment* The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
4. *Scrum of Scrums* There is a daily Scrum of Scrums where representatives from each team meet to discuss progress, identify problems, and plan the work to be done that day. Individual team Scrums may be staggered in time so that representatives from other teams can attend if necessary.

3.4.4 Agile methods across organizations

Small software companies that develop software products have been among the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards, and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to "scale out" these methods across the organization.

It can be difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow, and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software tools (e.g., requirements management tools), and the use of these tools is mandated for all projects.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made, and this conflicts with the notion of refactoring. When refactoring is part of an agile process, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with test-first approaches used in agile development methods.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. Rather than trying to force agile methods onto unwilling developers, companies have found that the best way to introduce agile is bit by bit, starting with an enthusiastic group of developers. A successful agile project can act as a starting point, with the project team spreading agile practice across the organization. Once the notion of agile is widely known, explicit actions can then be taken to spread it across the organization.

KEY POINTS

- Agile methods are iterative development methods that focus on reducing process overheads and documentation and on incremental software delivery. They involve customer representatives directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system. In practice, a mix of agile and plan-based techniques may be used.
- Agile development practices include requirements expressed as user stories, pair programming, refactoring, continuous integration, and test-first development.
- Scrum is an agile method that provides a framework for organizing agile projects. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest priority tasks for a sprint.
- To scale agile methods, some plan-based practices have to be integrated with agile practice. These include up-front requirements, multiple customer representatives, more documentation, common tooling across project teams, and the alignment of releases across teams.

FURTHER READING

"Get Ready for Agile Methods, With Care." A thoughtful critique of agile methods that discusses their strengths and weaknesses, written by a vastly experienced software engineer. Still very relevant, although almost 15 years old. (B. Boehm, *IEEE Computer*, January 2002) http://dx.doi.org/10.1109/2.976920

Extreme Programming Explained. This was the first book on XP and is still, perhaps, the most readable. It explains the approach from the perspective of one of its inventors, and his enthusiasm comes through very clearly in the book. (K. Beck and C. Andres, Addison-Wesley, 2004) *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. This is a comprehensive and readable description of the 2011 development of the Scrum method (K.S. Rubin, Addison-Wesley, 2013).

"Agility at Scale: Economic Governance, Measured Improvement and Disciplined Delivery." This paper discusses IBM's approach to scale agile methods, where they have a systematic approach to integrating plan-based and agile development. It is an excellent and thoughtful discussion of the key issues in scaling agile (A.W. Brown, S.W. Ambler, and W. Royce, *Proc. 35th Int. Conf. on Software Engineering*, 2013) http://dx.doi.org/10.1145/12944.12948

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/agile-methods/

EXERCISES

3.1. At the end of their study program, students in a software engineering course are typically expected to complete a major project. Explain how the agile methodology may be very useful for the students to use in this case.

3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.

3.3. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.

3.4. In test-first development, tests are written before the code. Explain how the test suite may compromise the quality of the software system being developed.

3.5. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.

3.6. Compare and contrast the Scrum approach to project management with conventional plan-based approaches as discussed in Chapter 23. Your comparison should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.

3.7. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using Scrum. Explain how you could use technology to support Scrum in a distributed environment to make this possible. What problems are you likely to encounter using this approach?

3.8. Why is it necessary to introduce some methods and documentation from plan-based approaches when scaling agile methods to larger projects that are developed by distributed development teams?

3.9. Explain why agile methods may not work well in organizations that have teams with a wide range of skills and abilities and well-established processes.

3.10. One of the problems of having a user closely involved with a software development team is that they "go native." That is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem, and discuss the advantages and disadvantages of each approach.

REFERENCES

Ambler, S. W. 2010. "Scaling Agile: A Executive Guide." http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling_agile_an_executive_guide10/

Arisholm, E., H. Gallis, T. Dyba, and D. I. K. Sjoberg. 2007. "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise." *IEEE Trans. on Software Eng.* 33 (2): 65–86. doi:10.1109/TSE.2007.17.

Beck, K. 1998. "Chrysler Goes to 'Extremes.'" *Distributed Computing* (10): 24–28.

———. 1999. “Embracing Change with Extreme Programming.” *IEEE Computer* 32 (10): 70–78. doi:10.1109/2.796139.

Bellouiti, S. 2009. “How Scrum Helped Our A-Team.” http://www.scrumalliance.org/community/articles/2009/2009-june/how-scrum-helped-our team

Bird, J. 2011. “You Can't Be Agile in Maintenance.” http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html

Deemer, P. 2011. “The Distributed Scrum Primer.” http://www.goodagile.com/distributedscrumprimer/.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.

Jeffries, R., and G. Melnik. 2007. “TDD: The Art of Fearless Programming.” *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.

Kilner, S. 2012. “Can Agile Methods Work for Software Maintenance.” http://www.vlegaci.com/can-agile-methods-work-for-software-maintenance-part-1/

Larman, C., and V. R. Basili. 2003. “Iterative and Incremental Development: A Brief History.” *IEEE Computer* 36 (6): 47–56. doi:10.1109/MC.2003.1204375.

Leffingwell, D. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.

Leffingwell, D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Boston: Addison-Wesley.

Mulder, M., and M. van Vliet. 2008. “Case Study: Distributed Scrum Project for Dutch Railways.” *InfoQ*. http://www.infoq.com/articles/dutch-railway-scrum

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.

Schatz, B., and I. Abdelshafi. 2005. “Primavera Gets Agile: A Successful Transition to Agile Development.” *IEEE Software* 22 (3): 36–42. doi:10.1109/MS.2005.74.

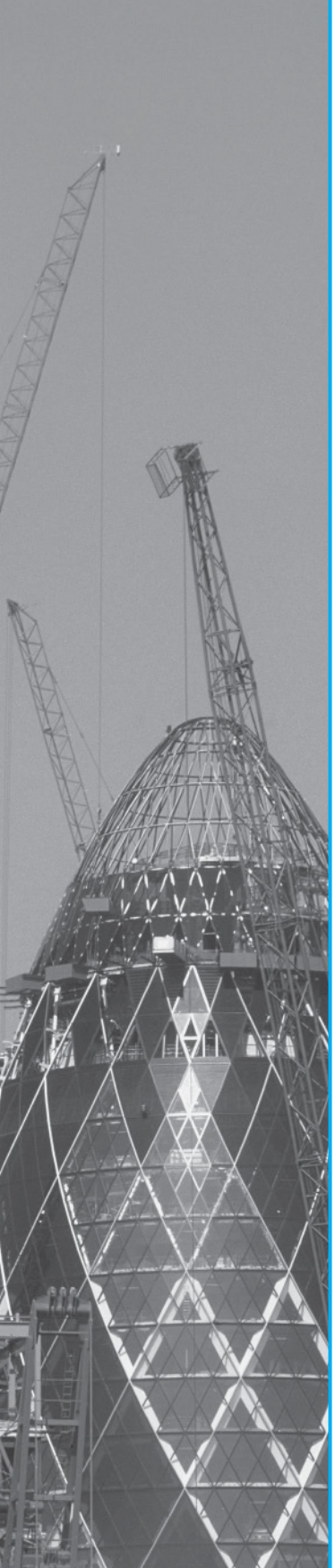
Schwaber, K., and M. Beedle. 2001. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.

Stapleton, J. 2003. *DSDM: Business Focused Development, 2nd ed*. Harlow, UK: Pearson Education.

Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2/e*. Greenwich, CT: Manning Publications.

Weinberg, G. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand.

Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. 2000. “Strengthening the Case for Pair Programming.” *IEEE Software* 17 (4): 19–25. doi:10.1109/52.854064.



4 Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to explain the processes involved in discovering and documenting these requirements. When you have read the chapter, you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand the main requirements engineering activities of elicitation, analysis, and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

4.1 Functional and non-functional requirements
4.2 Requirements engineering processes
4.3 Requirements elicitation
4.4 Requirements specification
4.5 Requirements validation
4.6 Requirements change

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term *requirement* is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis 1993) explains why these differences exist:

> *If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system*[†].

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different kinds of requirement are needed to communicate information about a system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from the mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite

[†]Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

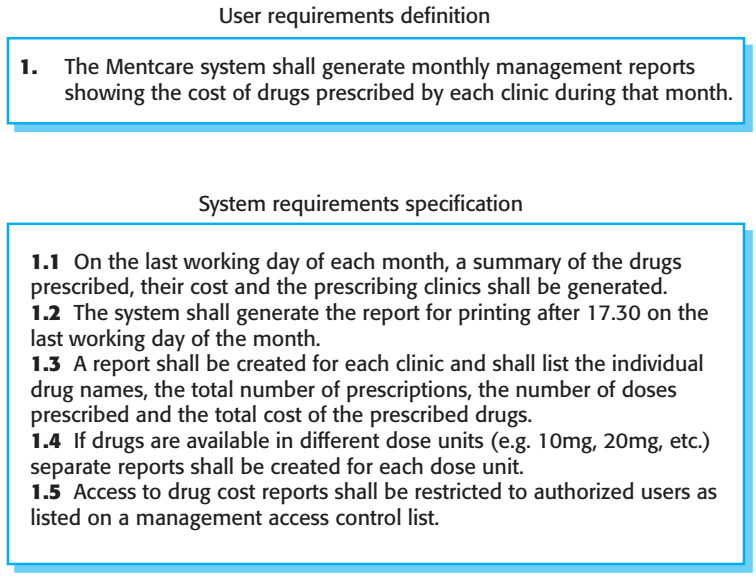


Figure 4.1 User and system requirements

general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 4.2 shows the types of readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

The different types of document readers shown in Figure 4.2 are examples of system stakeholders. As well as users, many other people have some kind of interest in the system. System stakeholders include anyone who is affected by the system in some way and so anyone who has a legitimate interest in it. Stakeholders range from end-users of a system through managers to external stakeholders such as regulators,

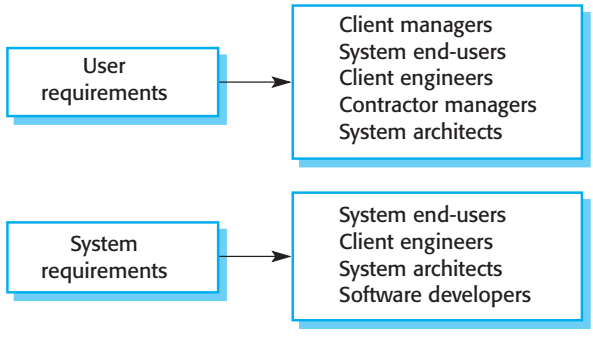


Figure 4.2 Readers of different types of requirements specification



Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: (1) Does the system contribute to the overall objectives of the organization? (2) Can the system be implemented within schedule and budget using current technology? and (3) Can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

http://software-engineering-book.com/web/feasibility-study/

who certify the acceptability of the system. For example, system stakeholders for the Mentcare system include:

1. Patients whose information is recorded in the system and relatives of these patients.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Health care managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Requirements engineering is usually presented as the first stage of the software engineering process. However, some understanding of the system requirements may have to be developed before a decision is made to go ahead with the procurement or development of a system. This early-stage RE establishes a high-level view of what the system might do and the benefits that it might provide. These may then be considered in a feasibility study, which tries to assess whether or not the system is technically and financially feasible. The results of that study help management decide whether or not to go ahead with the procurement or development of the system.

In this chapter, I present a "traditional" view of requirements rather than requirements in agile processes, which I discussed in Chapter 3. For the majority of large systems, it is still the case that there is a clearly identifiable requirements engineering phase before implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, subsequent changes are made to the requirements, and user requirements may be expanded into

more detailed system requirements. Sometimes an agile approach of concurrently eliciting the requirements as the system is developed may be used to add detail and to refine the user requirements.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services.

In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered effectively.

4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional



Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. This means that these engineers may not know whether or not a domain requirement has been missed out or conflicts with other requirements.

http://software-engineering-book.com/web/domain-requirements/

requirements for the Mentcare system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These user requirements define specific functionality that should be included in the system. The requirements show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off-the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Therefore, an information requirement for the Mentcare system might specify what information is to be included in the list of patients expected for appointments that day.

Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first Mentcare system requirement in the above list states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, regardless of the clinic.

A medical staff member specifying a search requirement may expect "search" to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement so that it is easier to implement. Their search function may require the user to choose a clinic and then carry out the search of the patients who attended that clinic. This involves more user input and so takes longer to complete the search.

Ideally, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services and information required by the user should be defined. Consistency means that requirements should not be contradictory.

In practice, it is only possible to achieve requirements consistency and completeness for very small software systems. One reason is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that large systems have many stakeholders, with different backgrounds and expectations. Stakeholders are likely to have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are originally specified, and the inconsistent requirements may only be discovered after deeper analysis or during system development.

4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. These non-functional requirements usually specify or constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. Alternatively, they may define constraints on the system implementation, such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

While it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), this is often more difficult with non-functional requirements. The implementation of these requirements may be spread throughout the system, for two reasons:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met in an embedded system, you may have to organize the system to minimize communications between components.

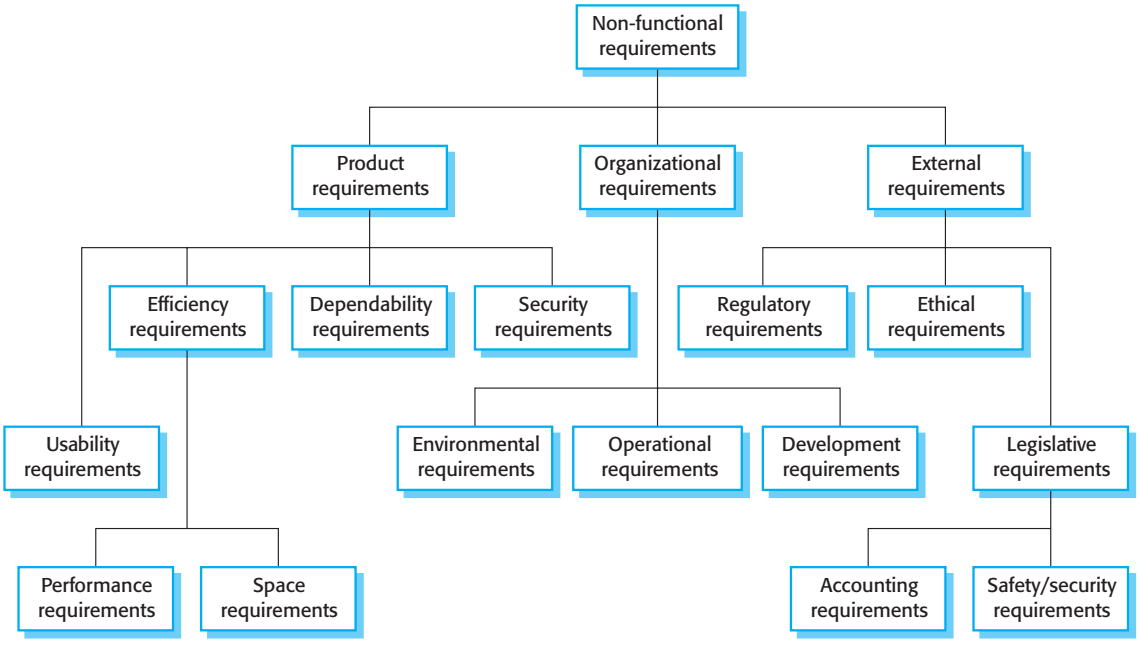


Figure 4.3 Types of non-functional requirements

2. An individual non-functional requirement, such as a security requirement, may generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented. In addition, it may also generate requirements that constrain existing requirements; for example, it may limit access to information in the system.

Nonfunctional requirements arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or external sources:

1. *Product requirements* These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; security requirements; and usability requirements.

2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of possible non-functional requirements for the Mentcare system

programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.

3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

Figure 4.4 shows examples of product, organizational, and external requirements that could be included in the Mentcare system specification. The product requirement is an availability requirement that defines when the system has to be available and the allowed downtime each day. It says nothing about the functionality of the Mentcare system and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in health care systems, and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that stakeholders propose requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

I have rewritten this to show how the goal could be expressed as a "testable" non-functional requirement. It is impossible to objectively verify the system goal, but in the following description you can at least include software instrumentation to count the errors made by users when they are testing the system.

> *Medical staff shall be able to use all the system functions after two hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no simple metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the reliability (for example) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high, and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the identification requirement in Figure 4.4 requires a card reader to be installed with each computer that connects to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' tablets or smartphones. These are not normally

equipped with card readers so, in these circumstances, some alternative identification method may have to be supported.

It is difficult to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should, ideally, highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these dependability requirements in Part 2, which describes ways of specifying reliability, safety, and security requirements.

4.2 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering involves three key activities. These are discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a standard form (specification); and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.4. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.6 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the non-functional requirements and more detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so that the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; and modifications are made to the system's hardware, software, and organizational environment. Changes have to be managed to understand the impact on other requirements and the cost and system implications of making the change. I discuss this process of requirements management in Section 4.6.

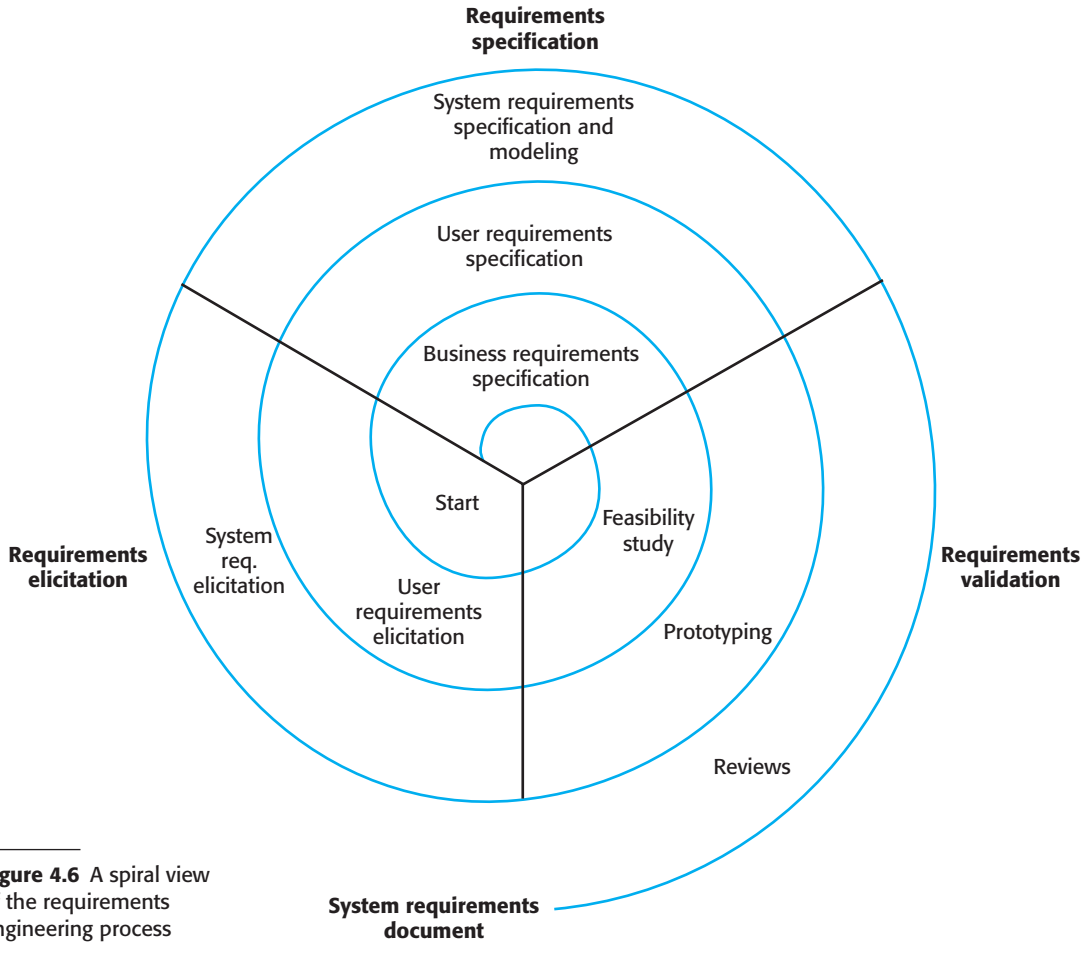


Figure 4.6 A spiral view of the requirements engineering process

4.3 Requirements elicitation

The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work. During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.

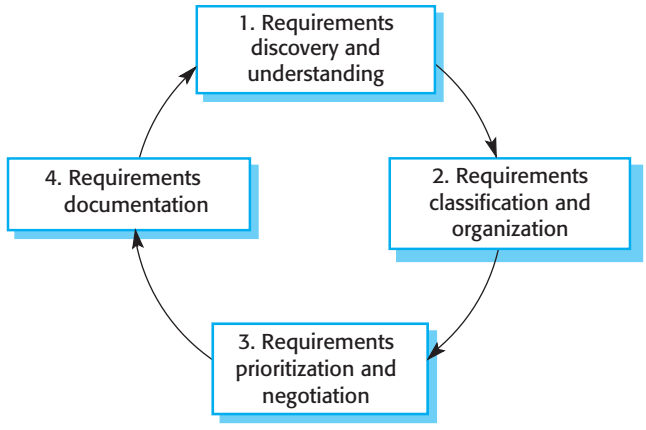


Figure 4.7 The requirements elicitation and analysis process

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders, with diverse requirements, may express their requirements in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

A process model of the elicitation and analysis process is shown in Figure 4.7. Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed, and the standards used.

The process activities are:

1. *Requirements discovery and understanding* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.
3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts



Viewpoints

A viewpoint is a way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, or others. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.software-engineering-book.com/web/viewpoints/

through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements documentation* The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained informally on whiteboards, wikis, or other shared spaces.

Figure 4.7 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document has been produced.

To simplify the analysis of requirements, it is helpful to organize and group the stakeholder information. One way of doing so is to consider each stakeholder group to be a viewpoint and to collect all requirements from that group into the viewpoint. You may also include viewpoints to represent domain requirements and constraints from other systems. Alternatively, you can use a model of the system architecture to identify subsystems and to associate requirements with each subsystem.

Inevitably, different stakeholders have different views on the importance and priority of requirements, and sometimes these views are conflicting. If some stakeholders feel that their views have not been properly considered, then they may deliberately attempt to undermine the RE process. Therefore, it is important that you organize regular stakeholder meetings. Stakeholders should have the opportunity to express their concerns and agree on requirements compromises.

At the requirements documentation stage, it is important that you use simple language and diagrams to describe the requirements. This makes it possible for stakeholders to understand and comment on these requirements. To make information sharing easier, it is best to use a shared document (e.g., on Google Docs or Office 365) or a wiki that is accessible to all interested stakeholders.

4.3.1 Requirements elicitation techniques

Requirements elicitation involves meeting with stakeholders of different kinds to discover information about the proposed system. You may supplement this information

with knowledge of existing systems and their usage and information from documents of various kinds. You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system.

There are two fundamental approaches to requirements elicitation:

1. Interviewing, where you talk to people about what they do.
2. Observation or ethnography, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements, which are then the basis for further discussions.

4.3.1.1 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions, but these usually lead to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work, and so they are usually happy to get involved in interviews. However, unless you have a system prototype to demonstrate, you should not expect stakeholders to suggest specific and detailed requirements. Everyone finds it difficult to visualize what a system might be like. You need to analyze the information collected and to generate the requirements from this.

Eliciting domain knowledge through interviews can be difficult, for two reasons:

1. All application specialists use jargon specific to their area of work. It is impossible for them to discuss domain requirements without using this terminology. They normally use words in a precise and subtle way that requirements engineers may misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures rarely match the reality of decision making in an organization, but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

To be an effective interviewer, you should bear two things in mind:

1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then you should be willing to change your mind about the system.
2. You should prompt the interviewee to get discussions going by using a springboard question or a requirements proposal, or by working together on a prototype system. Saying to people "tell me what you want" is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews is used along with other information about the system from documentation describing business processes or existing systems, user observations, and developer experience. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information, and so it should be used in conjunction with other requirements elicitation techniques.

4.3.1.2 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational environment, and software system requirements may be generated or constrained by that environment. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how social and organizational factors affect the practical operation of the system. It is therefore very important that, during the requirements engineering process, you try to understand the social and organizational issues that affect the use of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes. An analyst immerses himself or herself in the working environment where

the system will be used. The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but that are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a workgroup may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (Suchman 1983) pioneered the use of ethnography to study office work. She found that actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (Crabtree 2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville 2000) and documenting patterns of interaction in cooperative systems (Martin and Sommerville 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.

2. Requirements derived from cooperation and awareness of other people's activities. For example, air traffic controllers (ATCs) may use an awareness of other controlles' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with the development of a system prototype (Figure 4.8). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al. 1993).

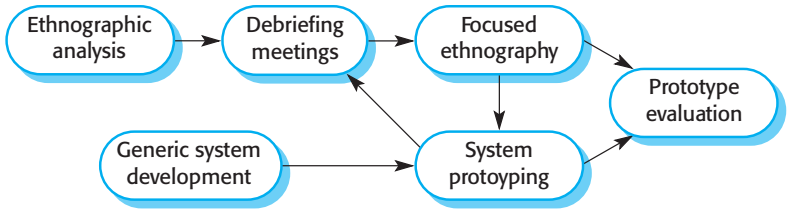


Figure 4.8 Ethnography and prototyping for requirements analysis

Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not effective for discovering broader organizational or domain requirements or for suggestion innovations. You therefore have to use ethnography as one of a number of techniques for requirements elicitation.

4.3.2 Stories and scenarios

People find it easier to relate to real-life examples than abstract descriptions. They are not good at telling you the system requirements. However, they may be able to describe how they handle particular situations or imagine things that they might do in a new way of working. Stories and scenarios are ways of capturing this kind of information. You can then use these when interviewing groups of stakeholders to discuss the system with other stakeholders and to develop more specific system requirements.

Stories and scenarios are essentially the same thing. They are a description of how the system can be used for some particular task. They describe what people do, what information they use and produce, and what systems they may use in this process. The difference is in the ways that descriptions are structured and in the level of detail presented. Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs. I find stories to be effective in setting out the "big picture." Parts of stories can then be developed in more detail and represented as scenarios.

Figure 4.9 is an example of a story that I developed to understand the requirements for the iLearn digital learning environment that I introduced in Chapter 1. This story describes a situation in a primary (elementary) school where the teacher is using the environment to support student projects on the fishing industry. You can see this is a very high-level description. Its purpose is to facilitate discussion of how the iLearn system might be used and to act as a starting point for eliciting the requirements for that system.

Photo sharing in the classroom

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Figure 4.9 A user story for the iLearn system

The advantage of stories is that everyone can easily relate to them. We found this approach to be particularly useful to get information from a wider community than we could realistically interview. We made the stories available on a wiki and invited teachers and students from across the country to comment on them.

These high-level stories do not go into detail about a system, but they can be developed into more specific scenarios. Scenarios are descriptions of example user interaction sessions. I think that it is best to present scenarios in a structured way rather than as narrative text. User stories used in agile methods such as Extreme Programming, are actually narrative scenarios rather than general stories to help elicit requirements.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expect when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how resulting problems can be handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario ends.

As an example of a scenario, Figure 4.10 describes what happens when a student uploads photos to the KidsTakePics system, as explained in Figure 4.9. The key difference between this system and other systems is that a teacher moderates the uploaded photos to check that they are suitable for sharing.

You can see this is a much more detailed description than the story in Figure 4.9, and so it can be used to propose requirements for the iLearn system. Like stories, scenarios can be used to facilitate discussions with stakeholders who sometimes may have different ways of achieving the same result.

Uploading photos to KidsTakePics

Initial assumption: A user or a group of users have one or more digital photographs to be uploaded to the picture-sharing site. These photos are saved on either a tablet or a laptop computer. They have successfully logged on to KidsTakePics.

Normal: The user chooses to upload photos and is prompted to select the photos to be uploaded on the computer and to select the project name under which the photos will be stored. Users should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator, asking them to check new content, and generates an on-screen message to the user that this checking has been done.

What can go wrong: No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wishes to re-upload the photos with the same name, rename the photos, or cancel the upload. If users choose to re-upload the photos, the originals are overwritten. If they choose to rename the photos, a new name is automatically generated by adding a number to the existing filename.

Other activities: The moderator may be logged on to the system and may approve photos as they are uploaded.

System state on completion: User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.

Figure 4.10 Scenario for uploading photos in KidsTakePics

4.4 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is almost impossible to achieve. Stakeholders interpret the requirements in different ways, and there are often inherent conflicts and inconsistencies in the requirements.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language, but other notations based on forms, graphical, or mathematical system models can also be used. Figure 4.11 summarizes possible notations for writing system requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Figure 4.11 Notations for writing system requirements

System requirements are expanded versions of the user requirements that software engineers use as the starting point for the system design. They add detail and explain how the system should provide the user requirements. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should only describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is neither possible nor desirable to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different subsystems that make up the system. We did this when we were defining the requirements for the iLearn system, where we proposed the architecture shown in Figure 1.8.
2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements, such as N-version programming to achieve reliability, discussed in Chapter 11, may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified should be used.

4.4.1 Natural language specification

Natural language has been used to write requirements for software since the 1950s. It is expressive, intuitive, and universal. It is also potentially vague and ambiguous, and its interpretation depends on the background of the reader. As a result, there

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow, so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.12 Example requirements for the insulin pump software system

have been many proposals for alternative ways to write requirements. However, none of these proposals has been widely adopted, and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow these simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using "shall." Desirable requirements are not essential and are written using "should."
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical, software engineering language. It is easy for words such as "architecture" and "module" to be misunderstood. Wherever possible, you should avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable.

Figure 4.12 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. Other requirements for this embedded system are defined in the insulin pump requirements document, which can be downloaded from the book's web pages.

4.4.2 Structured specifications

Structured natural language is a way of writing system requirements where requirements are written in a standard way rather than as free-form text. This approach maintains most of the expressiveness and understandability of natural language but



Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence, and structuring natural language requirements can be difficult.

http://software-engineering-book.com/web/natural-language/

ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson 2013), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, and supporting materials. This is similar to the approach used in the example of a structured specification shown in Figure 4.13.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.13.

When a standard format is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and the origin of these inputs.
3. A description of its outputs and the destination of these outputs.
4. Information about the information needed for the computation or other entities in the system that are required (the "requires" part).
5. A description of the action to be taken.
6. If a functional approach is used, a precondition setting out what must be true before the function is called, and a postcondition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced, and requirements are organized

Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered. (see Figure 4.14)
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Precondition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Postcondition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Figure 4.13 The structured specification of a requirement for an insulin pump

more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.14 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

Figure 4.14 The tabular specification of computation in an insulin pump

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1) < (r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing r2 > r1 & ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

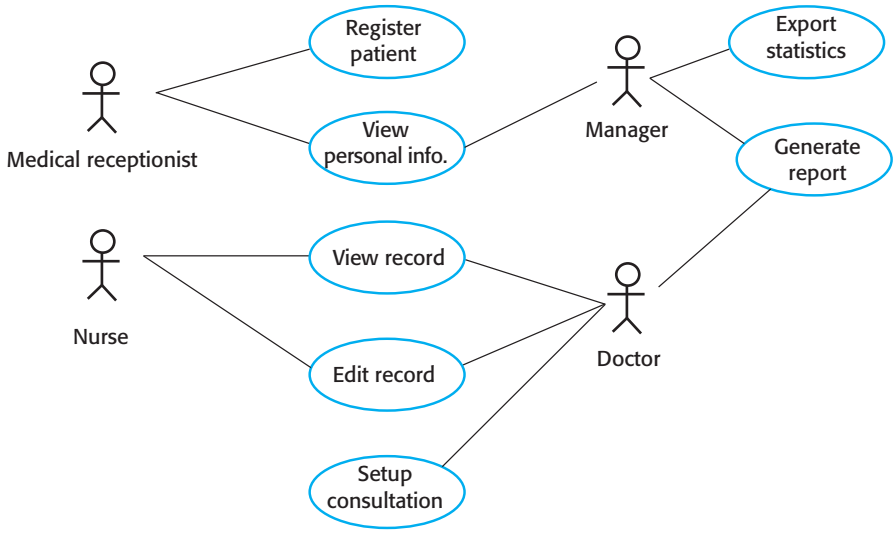


Figure 4.15 Use cases for the Mentcare system

4.4.3 Use cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. They were first introduced in the Objectory method (Jacobsen et al. 1993) and have now become a fundamental feature of the Unified Modeling Language (UML). In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. You then add additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as the UML sequence or state charts (see Chapter 5).

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the Mentcare system.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

> *Setup consultation allows two or more doctors, working in different offices, to view the same patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown menu of doctors who are online. The patient record is then displayed on their screens, but only the initiating doctor can edit the record. In addition, a text chat window is created*

to help coordinate actions. It is assumed that a phone call for voice communication can be separately arranged.

The UML is a standard for object-oriented modeling, so use cases and use case-based elicitation are used in the requirements engineering process. However, my experience with use cases is that they are too fine-grained to be useful in discussing requirements. Stakeholders don't understand the term *use case*; they don't find the graphical model to be useful, and they are often not interested in a detailed description of each and every system interaction. Consequently, I find use cases to be more helpful in systems design than in requirements engineering. I discuss use cases further in Chapter 5, which shows how they are used alongside other system models to document a system design.

Some people think that each use case is a single, low-level interaction scenario. Others, such as Stevens and Pooley (Stevens and Pooley 2006), suggest that each use case includes a set of related, low-level scenarios. Each of these scenarios is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. In practice, you can use them in either way.

4.4.4 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It may include both the user requirements for a system and a detailed specification of the system requirements. Sometimes the user and system requirements are integrated into a single description. In other cases, the user requirements are described in an introductory chapter in the system requirements specification.

Requirements documents are essential when systems are outsourced for development, when different teams develop different parts of the system, and when a detailed analysis of the requirements is mandatory. In other circumstances, such as software product or business system development, a detailed requirements document may not be needed.

Agile methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, agile approaches often collect user requirements incrementally and write these on cards or whiteboards as short user stories. The user then prioritizes these stories for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.16 shows possible users of the document and how they use it.

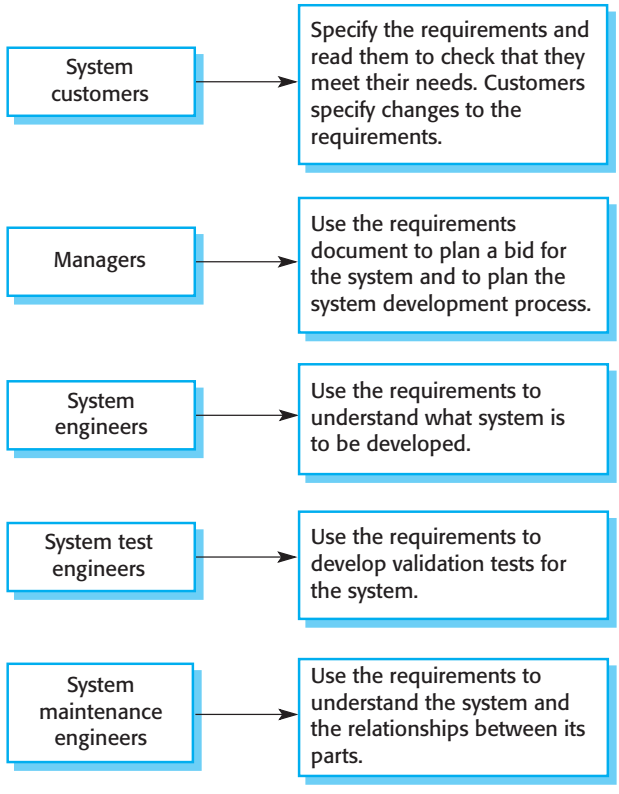


Figure 4.16 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise. It has to describe the requirements for customers, define the requirements in precise detail for developers and testers, as well as include information about future system evolution. Information on anticipated changes helps system designers to avoid restrictive design decisions and maintenance engineers to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need detailed requirements because safety and security have to be analyzed in detail to find possible requirements errors. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be less detailed. Details can be added to the requirements and ambiguities resolved during development of the system.

Figure 4.17 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE 1998). This standard is a generic one that can be adapted to specific uses. In this case, the standard has been extended to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4.17 The structure of a requirements document

Naturally, the information included in a requirements document depends on the type of software being developed and the approach to development that is to be used. A requirements document with a structure like that shown in Figure 4.17 might be produced for a complex engineering system that includes hardware and software developed by different companies. The requirements document is likely to be long and detailed. It is therefore important that a comprehensive table of contents and document index be included so that readers can easily find the information they need.

By contrast, the requirements document for an in-house software product will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, nonfunctional system requirements. The system designers and programmers use their judgment to decide how to meet the outline user requirements for the system.



Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers, and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://software-engineering-book.com/web/requirements-standard/

4.5 Requirements validation

Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. A change to the requirements usually means that the system design and implementation must also be changed. Furthermore, the system must then be retested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.



Requirements reviews

A requirements review is a process in which a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

http://software-engineering-book.com/web/requirements-reviews/

A number of requirements validation techniques can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users.

As a result, you rarely find all requirements problems during the requirements validation process. Inevitably, further requirements changes will be needed to correct omissions and misunderstandings after agreement has been reached on the requirements document.

4.6 Requirements change

The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address "wicked" problems—problems that cannot be completely defined (Rittel and Webber 1973). Because the problem cannot be fully defined, the software requirements are bound to

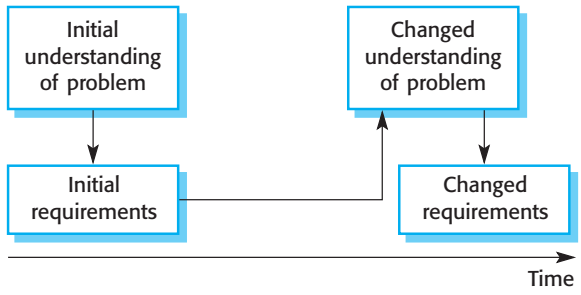


Figure 4.18
Requirements evolution

be incomplete. During the software development process, the stakeholders' understanding of the problem is constantly changing (Figure 4.18). The system requirements must then evolve to reflect this changed problem understanding.

Once a system has been installed and is regularly used, new requirements inevitably emerge. This is partly a consequence of errors and omissions in the original requirements that have to be corrected. However, most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated. It may be necessary to interface the system with other systems. Business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that require system compliance.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements, and, after delivery, new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory. The final system requirements are inevitably a compromise, and some stakeholders have to be given priority. With experience, it is often discovered that the balance of support given to different stakeholders has to be changed and the requirements re-prioritized.

As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You therefore need a formal process for making change proposals and linking these to system requirements. This process of "requirements management" should start as soon as a draft version of the requirements document is available.

Agile development processes have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management



Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

http://software-engineering-book.com/web/changing-requirements/

process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped for the change to be implemented.

The problem with this approach is that users are not necessarily the best people to decide on whether or not a requirements change is cost-effective. In systems with multiple stakeholders, changes will benefit some stakeholders and not others. It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

4.6.1 Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, you have to decide on a number of issues:

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

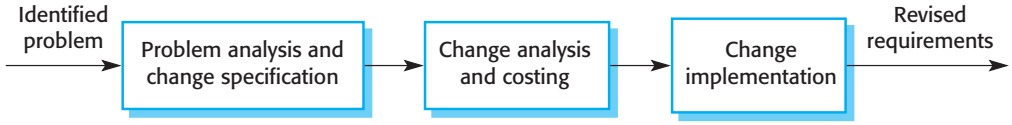


Figure 4.19 Requirements change management

2. *Change management* The process of change management (Figure 4.19) is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.
3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, you do not need to use specialized requirements management tools. Requirements management can be supported using shared web documents, spreadsheets, and databases. However, for larger systems, more specialized tool support, using systems such as DOORS (IBM 2013), makes it much easier to keep track of a large number of changing requirements.

4.6.2 Requirements change management

Requirements change management (Figure 4.19) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.



Requirements traceability

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

http://software-engineering-book.com/web/traceability/

3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the revised requirements.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process includes requirements elicitation, requirements specification, requirements validation, and requirements management.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.

- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

FURTHER READING

"Integrated Requirements Engineering: A Tutorial." This is a tutorial paper that discusses requirements engineering activities and how these can be adapted to fit with modern software engineering practice. (I. Sommerville, *IEEE Software,* 22(1), January–February 2005) http://dx.doi.org/10.1109/MS.2005.13.

"Research Directions in Requirements Engineering." This is a good survey of requirements engineering research that highlights future research challenges in the area to address issues such as scale and agility. (B. H. C. Cheng and J. M. Atlee, *Proc. Conf. on Future of Software Engineering,* IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.17.

Mastering the Requirements Process, 3rd ed. A well-written, easy-to-read book that is based on a particular method (VOLERE) but that also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 2013, Addison-Wesley).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

Requirements document for the insulin pump:

http://software-engineering-book.com/case-studies/insulin-pump/

Mentcare system requirements information:

http://software-engineering-book.com/case-studies/mentcare-system/

EXERCISES

4.1. Identify and briefly describe four types of requirements that may be defined for a computer-based system.

4.2. Discover ambiguities or omissions in the following statement of the requirements for part of a drone system intended for search and recovery:

The drone, a quad chopper, will be very useful in search and recovery operations, especially in remote areas or in extreme weather conditions. It will click high-resolution images. It will fly according to a path preset by a ground operator, but will be able to avoid obstacles on its own, returning to its original path whenever possible. The drone will also be able to identify various objects and match them to the target it is looking for.

4.3. Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in a sensible way.

4.4. Write a set of non-functional requirements for the drone system, setting out its expected safety and response time.

4.5. Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader, then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

The cash-dispensing function in a bank ATM.

In an Internet banking system, a facility that allows customers to transfer funds from one account held with the bank to another account with the same bank.

4.6. Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

4.7. Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.

4.8. To minimize mistakes during a requirements review, an organization decides to allocate two scribes to document the review session. Explain how this can be done.

4.9. When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.

4.10. You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you

should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. However, you also have a responsibility of confidentiality to your previous employer.

REFERENCES

Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.

Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.

IBM. 2013. "Rational Doors Next Generation: Requirements Engineering for Complex Systems." https://jazz.net/products/rational-doors-next-generation/

IEEE. 1998. "IEEE Recommended Practice for Software Requirements Specifications." In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.

Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.

Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.

Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

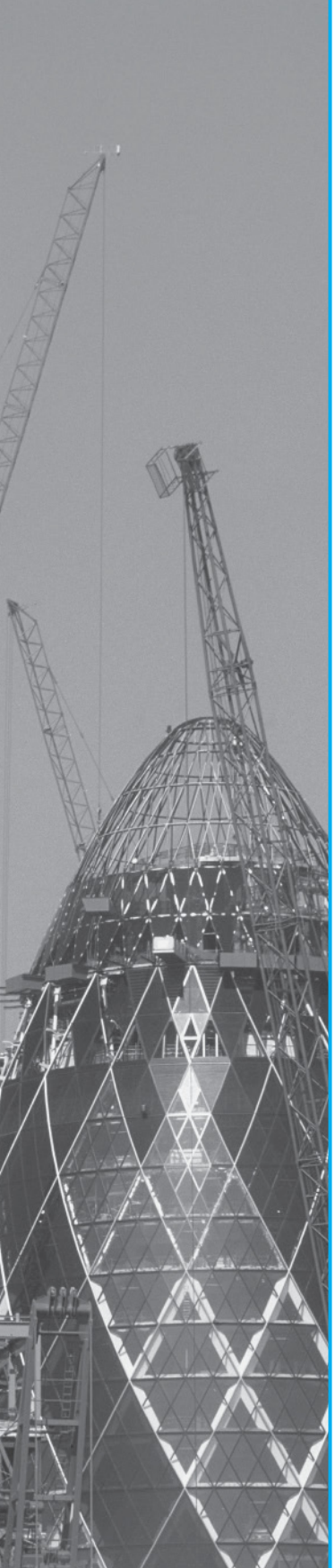
Robertson, S., and J. Robertson. 2013. *Mastering the Requirements Process, 3rd ed.* Boston: Addison-Wesley.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale. 1993. "Integrating Ethnography into the Requirements Engineering Process." In *RE' 93*, 165–173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.

Stevens, P., and R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison-Wesley.

Suchman, L. 1983. "Office Procedures as Practical Action: Models of Work and System Design." *ACM Transactions on Office Information Systems* 1 (3): 320–328. doi:10.1145/357442.357445.

Viller, S., and I. Sommerville. 2000. "Ethnographically Informed Analysis for Software Engineers." *Int. J. of Human-Computer Studies* 53 (1): 169–196. doi:10.1006/ijhc.2000.0370.



5 System modeling

Objectives

The aim of this chapter is to introduce system models that may be developed as part of requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable system is automatically generated from structural and behavioral models.

Contents

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling now usually means representing a system using some kind of graphical notation based on diagram types in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification. I cover graphical modeling using the UML here, and formal modeling is briefly discussed in Chapter 10.

Models are used during the requirements engineering process to help derive the detailed requirements for a system, during the design process to describe the system to engineers implementing the system, and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does, and they can be used to focus a stakeholder discussion on its strengths and weaknesses.
2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. If you use a model-driven engineering process (Brambilla, Cabot, and Wimmer 2012), you can generate a complete or partial system implementation from system models.

It is important to understand that a system model is not a complete representation of system. It purposely leaves out detail to make it easier to understand. A model is an abstraction of the system being studied rather than an alternative representation of that system. A representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies a system design and picks out the most salient characteristics. For example, the PowerPoint slides that accompany this book are an abstraction of the book's key points. However, if the book were translated from English into Italian, this would be an alternative *representation*. The translator's intention would be to maintain all the information as it is presented in English.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.
2. An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
3. A structural perspective, where you model the organization of a system or the structure of the data processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.



The Unified Modeling Language

The Unified Modeling Language (UML) is a set of 13 different diagram types that may be used to model software systems. It emerged from work in the 1990s on object-oriented modeling, where similar object-oriented notations were integrated to create the UML. A major revision (UML 2) was finalized in 2004. The UML is universally accepted as the standard approach for developing models of software systems. Variants, such as SysML, have been proposed for more general system modeling.

http://software-engineering-book.com/web/uml/

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depend on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a way to stimulate and focus discussion about an existing or proposed system. The purpose of the model is to stimulate and focus discussion among the software engineers involved in developing the system. The models may be incomplete (as long as they cover the key points of the discussion), and they may use the modeling notation informally. This is how models are normally used in agile modeling (Ambler and Jeffries 2002).
2. As a way of documenting an existing system. When models are used as documentation, they do not have to be complete, as you may only need to use models to document some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.
3. As a detailed system description that can be used to generate a system implementation. Where models are used as part of a model-based development process, the system models have to be both complete and correct. They are used as a basis for generating the source code of the system, and you therefore have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that may have different meanings.

In this chapter, I use diagrams defined in the Unified Modeling Language (UML) (Rumbaugh, Jacobson, and Booch 2004; Booch, Rumbaugh, and Jacobson 2005), which has become a standard language for object-oriented modeling. The UML has 13 diagram types and so supports the creation of many different types of system model. However, a survey (Erickson and Siau 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system. I therefore concentrate on these five UML diagram types here:

1. *Activity diagrams,* which show the activities involved in a process or in data processing.
2. *Use case diagrams,* which show the interactions between a system and its environment.
3. *Sequence diagrams,* which show interactions between actors and the system and between system components.
4. *Class diagrams,* which show the object classes in the system and the associations between these classes.
5. *State diagrams,* which show how the system reacts to internal and external events.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries, that is, on what is and is not part of the system being developed. This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment. You may decide that automated support for some business processes should be implemented in the software being developed but that other processes should be manual or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

For example, say you are developing the specification for the Mentcare patient information system. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed. In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information. The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information, and if these systems are unavailable, then it may be impossible to use the Mentcare system.

In some situations, the user base for a system is very diverse, and users have a wide range of different system requirements. You may decide not to define

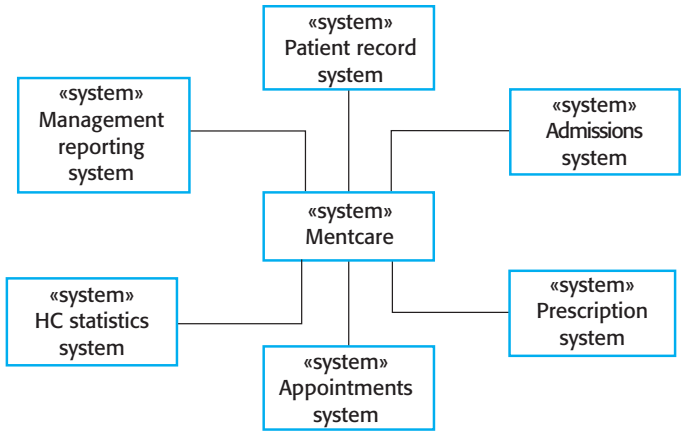


Figure 5.1 The context of the Mentcare system

boundaries explicitly but instead to develop a configurable system that can be adapted to the needs of different users. This was the approach that we adopted in the iLearn systems, introduced in Chapter 1. There, users range from very young children who can't read through to young adults, their teachers, and school administrators. Because these groups need different system boundaries, we specified a configuration system that would allow the boundaries to be specified when the system was deployed.

The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by nontechnical factors. For example, a system boundary may be deliberately positioned so that the complete analysis process can be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; and it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

Figure 5.1 is a context model that shows the Mentcare system and the other systems in its environment. You can see that the Mentcare system is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital admissions, and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of

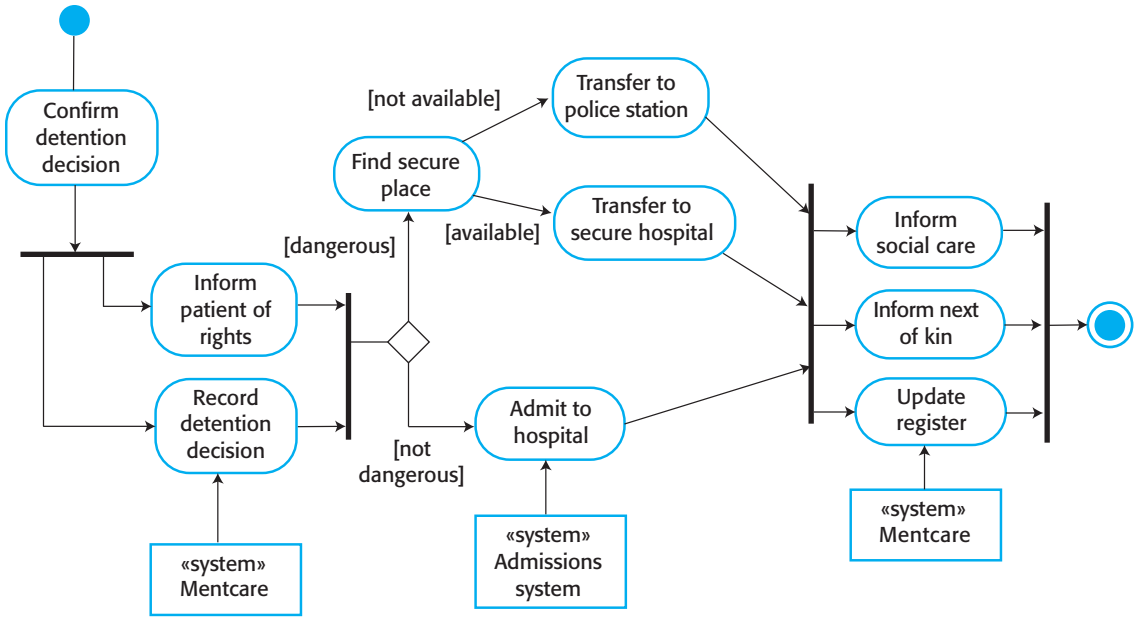


Figure 5.2 A process model of involuntary detention

these relations may affect the requirements and design of the system being defined and so must be taken into account. Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

UML activity diagrams may be used to show the business processes in which systems are used. Figure 5.2 is a UML activity diagram that shows where the Mentcare system is used in an important mental health care process—involuntary detention.

Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered. Such detention is subject to strict legal safeguards—for example, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One critical function of the Mentcare system is to ensure that such safeguards are implemented and that the rights of patients are respected.

UML activity diagrams show the activities in a process and the flow of control from one activity to another. The start of a process is indicated by a filled circle, the end by a filled circle inside another circle. Rectangles with round corners represent activities, that is, the specific subprocesses that must be carried out. You may include objects in activity charts. Figure 5.2 shows the systems that are used to support different subprocesses within the involuntary detection process. I have shown that these are separate systems by using the UML stereotype feature where the type of entity in the box between chevrons is shown.

Arrows represent the flow of work from one activity to another, and a solid bar indicates activity coordination. When the flow from more than one activity leads to a

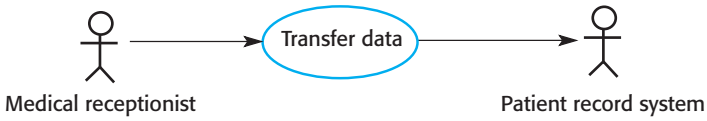


Figure 5.3 Transfer-data use case

solid bar, then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 5.2, the activities to inform social care and the patient's next of kin, as well as to update the detention register, may be concurrent.

Arrows may be annotated with guards (in square brackets) that specify when that flow is followed. In Figure 5.2, you can see guards showing the flows for patients who are dangerous and not dangerous to society. Patients who are dangerous to society must be detained in a secure facility. However, patients who are suicidal and are a danger to themselves may be admitted to an appropriate ward in a hospital, where they can be kept under close supervision.

5.2 Interaction models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs; interaction between the software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling is important as it helps to identify user requirements. Modeling system-to-system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

This section discusses two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external agents (human users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interactions at different levels of detail and so may be used together. For example, the details of the interactions involved in a high-level use case may be documented in a sequence diagram. The UML also includes communication diagrams that can be used to model interactions. I don't describe this diagram type because communication diagrams are simply an alternative representation of sequence diagrams.

5.2.1 Use case modeling

Use case modeling was originally developed by Ivar Jacobsen in the 1990s (Jacobsen et al. 1993), and a UML diagram type to support use case modeling is part of the

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Figure 5.4 Tabular description of the Transfer-data use case

UML. A use case can be taken as a simple description of what a user expects from a system in that interaction. I have discussed use cases for requirements elicitation in Chapter 4. As I said in Chapter 4, I find use case models to be more useful in the early stages of system design rather than in requirements engineering.

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse, with the actors involved in the use case represented as stick figures. Figure 5.3 shows a use case from the Mentcare system that represents the task of uploading data from the Mentcare system to a more general patient record system. This more general system maintains summary data about a patient rather than data about each consultation, which is recorded in the Mentcare system.

Notice that there are two actors in this use case—the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction, but it is also used to represent other external systems and hardware. Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of flow of messages. Obviously, in a use case, messages pass in both directions. However, the arrows in Figure 5.3 are used informally to indicate that the medical receptionist initiates the transaction and data is transferred to the patient record system.

Use case diagrams give a simple overview of an interaction, and you need to add more detail for complete interaction description. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram. You choose the most appropriate format depending on the use case and the level of detail that you think is required in the model. I find a standard tabular format to be the most useful. Figure 5.4 shows a tabular description of the "Transfer data" use case.

Composite use case diagrams show a number of different use cases. Sometimes it is possible to include all possible interactions within a system in a single composite use case diagram. However, this may be impossible because of the number of use cases. In such cases, you may develop several diagrams, each of which shows related use cases. For example, Figure 5.5 shows all of the use cases in the Mentcare system

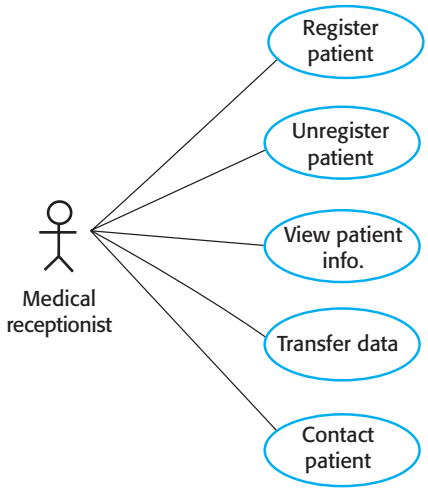


Figure 5.5 Use cases involving the role "Medical receptionist"

in which the actor "Medical Receptionist" is involved. Each of these should be accompanied by a more detailed description.

The UML includes a number of constructs for sharing all or part of a use case in other use case diagrams. While these constructs can sometimes be helpful for system designers, my experience is that many people, especially end-users, find them difficult to understand. For this reason, these constructs are not described here.

5.2.2 Sequence diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled. As space does not allow covering all possibilities here, the focus will be on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Figure 5.6 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Annotated arrows indicate interactions between objects. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. This example also shows the notation used to denote alternatives. A box named alt is used with the

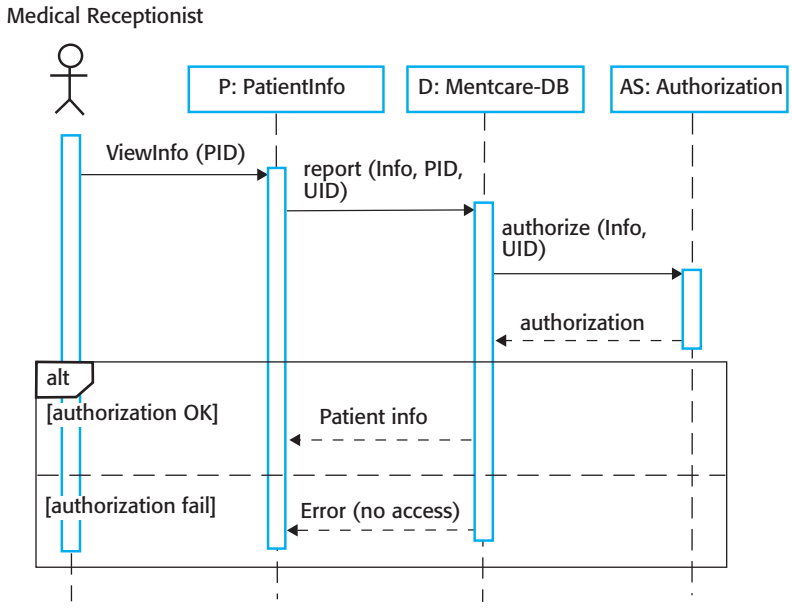


Figure 5.6 Sequence diagram for View patient information

conditions indicated in square brackets, with alternative interaction options separated by a dotted line.

You can read Figure 5.6 as follows:

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID to identify the required information. P is a user interface object, which is displayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking. (At this stage, it is not important where the receptionist's UID comes from.)
3. The database checks with an authorization system that the receptionist is authorized for this action.
4. If authorized, the patient information is returned and is displayed on a form on the user's screen. If authorization fails, then an error message is returned. The box denoted by "alt" in the top-left corner is a choice box indicating that one of the contained interactions will be executed. The condition that selects the choice is shown in square brackets.

Figure 5.7 is a further example of a sequence diagram from the same system that illustrates two additional features. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is

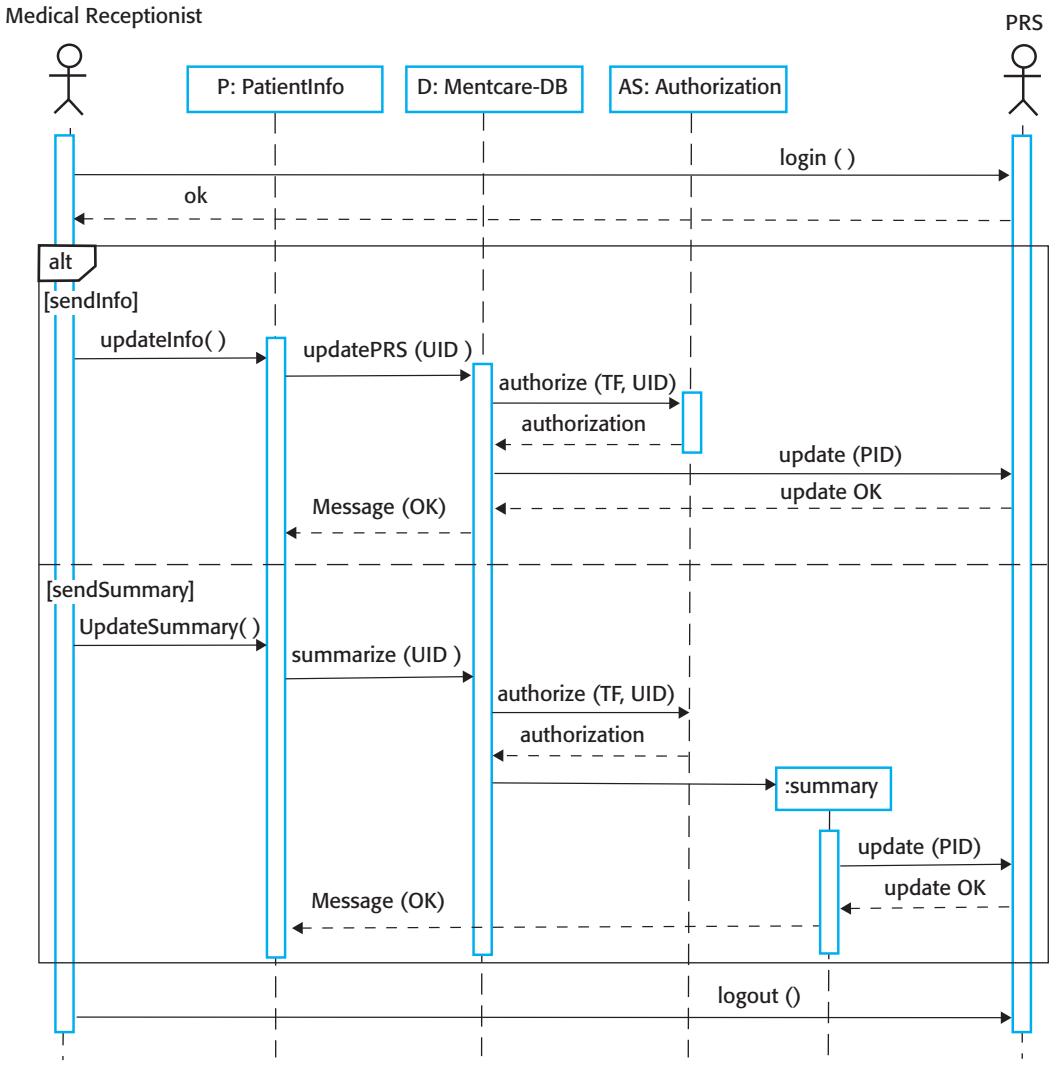


Figure 5.7 Sequence diagram for Transfer Data

to be uploaded to a national PRS (patient records system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. Two options are available (as shown in the "alt" box). These allow the direct transfer of updated patient information from the Mentcare database to the PRS and the transfer of summary health data from the Mentcare database to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.

4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database, and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message and the user logs off.

Unless you are using sequence diagrams for code generation or detailed documentation, you don't have to include every interaction in these diagrams. If you develop system models early in the development process to support requirements engineering and high-level design, there will be many interactions that depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object. As this is not important at this stage, you do not need to include it in the sequence diagram.

5.3 Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the organization of the system design, or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. These can be models of the overall system architecture or more detailed models of the objects in the system and their relationships.

In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system. Architectural design is an important topic in software engineering, and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover architectural modeling in Chapters 6 and 17.

5.3.1 Class diagrams

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes indicating that some relationship exists between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a



Figure 5.8 UML Classes and association

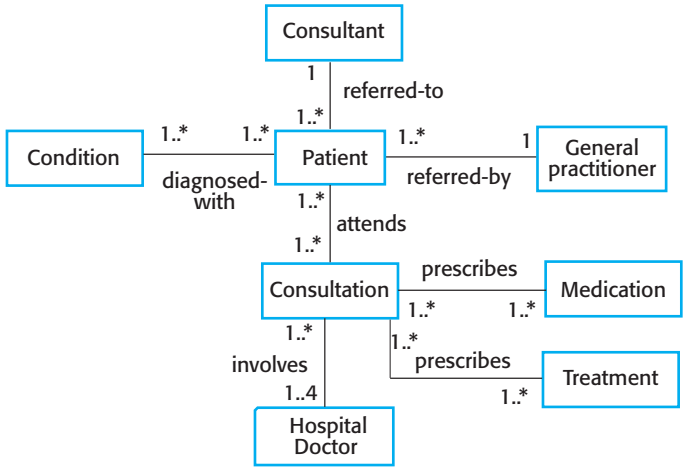


Figure 5.9 Classes and associations in the Mentcare system

prescription, or a doctor. As an implementation is developed, you define implementation objects to represent data that is manipulated by the system. In this section, the focus is on the modeling of real-world objects as part of the requirements or early software design processes. A similar approach is used for data structure modeling.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these diagrams is to write the class name in a box. You can also note the existence of an association by drawing a line between classes. For example, Figure 5.8 is a simple class diagram showing two classes, Patient and Patient Record, with an association between them. At this stage, you do not need to say what the association is.

Figure 5.9 develops the simple class diagram in Figure 5.8 to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists.

Figures 5.8 and 5.9, shows an important feature of class diagrams—the ability to show how many objects are involved in the association. In Figure 5.8 each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record, and each record maintains information about exactly one patient.

As you can see from Figure 5.9, other multiplicities are possible. You can define that an exact number of objects are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association. For example, the (1..*) multiplicity in Figure 5.9 on the relationship between Patient and Condition shows that a patient may suffer from several conditions and that the same condition may be associated with several patients.

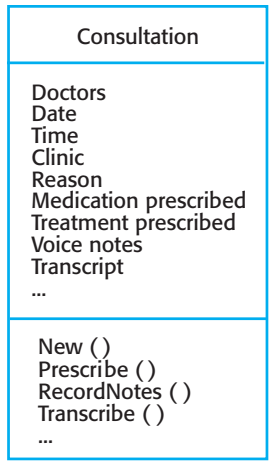


Figure 5.10 A Consultation class

At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities (Hull and King 1987). The UML does not include a diagram type for database modeling, as it models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes (they have no operations), attributes as object class attributes, and relations as named associations between object classes.

When showing the associations between classes, it is best to represent these classes in the simplest possible way, without attributes or operations. To define objects in more detail, you add information about their attributes (the object's characteristics) and operations (the object's functions). For example, a Patient object has the attribute Address, and you may include an operation called ChangeAddress, which is called when a patient indicates that he or she has moved from one address to another.

In the UML, you show attributes and operations by extending the simple rectangle that represents a class. I illustrate this in Figure 5.10 that shows an object representing a consultation between doctor and patient:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This includes the attribute names and, optionally, their types. I don't show the types in Figure 5.10.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle. I show some but not all operations in Figure 5.10.

In the example shown in Figure 5.10, it is assumed that doctors record voice notes that are transcribed later to record details of the consultation. To prescribe medication, the doctor involved must use the Prescribe method to generate an electronic prescription.

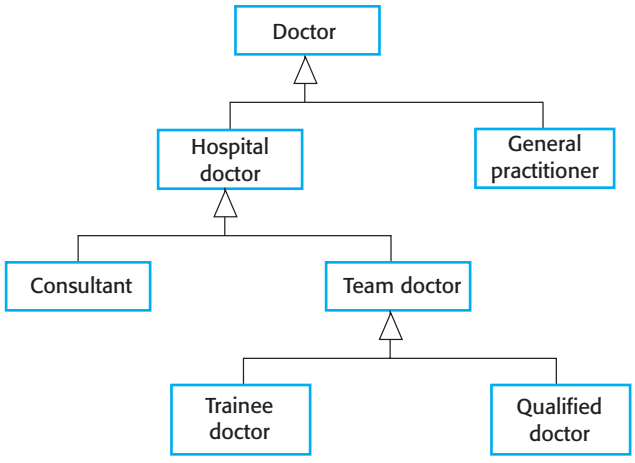


Figure 5.11 A generalization hierarchy

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of everything that we experience, we learn about general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. We then reuse knowledge by classifying things and focus on the differences between them and their class. For example, squirrels and rats are members of the class "rodents," and so share the characteristics of rodents. General statements apply to all class members; for example, all rodents have teeth for gnawing.

When you are modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization and class creation. This means that common information will be maintained in one place only. This is good design practice as it means that, if changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. You can make the changes at the most general level. In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

The UML has a specific type of association to denote generalization, as illustrated in Figure 5.11. The generalization is shown as an arrowhead pointing up to the more general class. This indicates that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor: those who have just graduated from medical school and have to be supervised (Trainee Doctor); those who can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision-making responsibilities.

In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. The lower-level classes are subclasses that inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

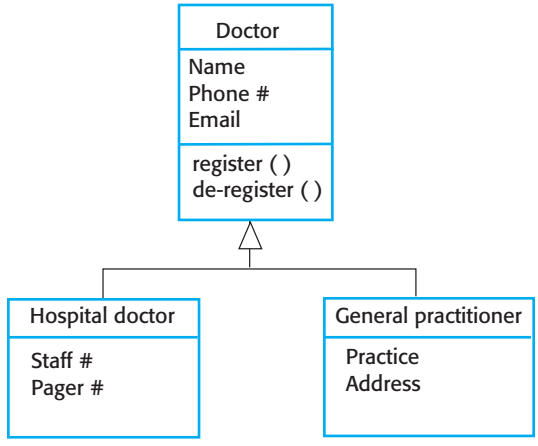


Figure 5.12 A generalization hierarchy with added detail

For example, all doctors have a name and phone number, and all hospital doctors have a staff number and carry a pager. General practitioners don't have these attributes, as they work independently, but they have an individual practice name and address. Figure 5.12 shows part of the generalization hierarchy, which I have extended with class attributes, for the class Doctor. The operations associated with the class Doctor are intended to register and de-register that doctor with the Mentcare system.

5.3.3 Aggregation

Objects in the real world are often made up of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and recommendations for further reading. Sometimes in a system model, you need to illustrate this. The UML provides a special type of association between classes called aggregation, which means that one object (the whole) is composed of other objects (the parts). To define aggregation, a diamond shape is added to the link next to the class that represents the whole.

Figure 5.13 shows that a patient record is an aggregate of Patient and an indefinite number of Consultations. That is, the record maintains personal patient information as well as an individual record for each consultation with a doctor.

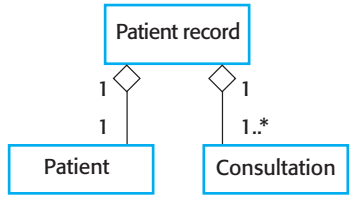


Figure 5.13 The aggregation association



Data flow diagrams

Data-flow diagrams (DFDs) are system models that show a functional perspective where each transformation represents a single function or process. DFDs are used to show how data flows through a sequence of processing steps. For example, a processing step could be the filtering of duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions, where data-flow diagrams are used to document a software design. Activity diagrams in the UML may be used to represent DFDs.

http://software-engineering-book.com/web/dfds/

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. These stimuli may be either data or events:

1. Data becomes available that has to be processed by the system. The availability of the data triggers the processing.
2. An event happens that triggers system processing. Events may have associated data, although this is not always the case.

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill for that customer.

By contrast, real-time systems are usually event-driven, with limited data processing. For example, a landline phone switching system responds to events such as "handset activated" by generating a dial tone, pressing keys on a handset by capturing the phone number, and so on.

5.4.1 Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They can be used during the analysis of requirements as they show end-to-end processing in a system. That is, they show the entire sequence of actions that takes place from an initial input being processed to the corresponding output, which is the system's response.

Data-driven models were among the first graphical software models. In the 1970s, structured design methods used data-flow diagrams (DFDs) as a way to illustrate the

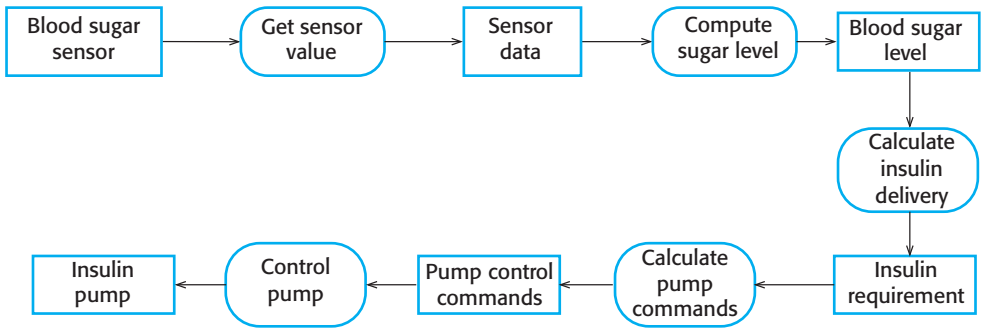


Figure 5.14 An activity model of the insulin pump's operation

processing steps in a system. Data-flow models are useful because tracking and documenting how data associated with a particular process moves through the system help analysts and designers understand what is going on in the process. DFDs are simple and intuitive and so are more accessible to stakeholders than some other types of model. It is usually possible to explain them to potential system users who can then participate in validating the model.

Data-flow diagrams can be represented in the UML using the activity diagram type, described in Section 5.1. Figure 5.14 is a simple activity diagram that shows the chain of processing involved in the insulin pump software. You can see the processing steps, represented as activities (rounded rectangles), and the data flowing between these steps, represented as objects (rectangles).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these diagrams can be used to model interaction, but if you draw these so that messages are only sent from left to right, then they show the sequential data processing in the system. Figure 5.15 illustrates this, using a sequence model of processing an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the operations or activities. In practice, nonexperts seem to find data-flow diagrams more intuitive, but engineers prefer sequence diagrams.

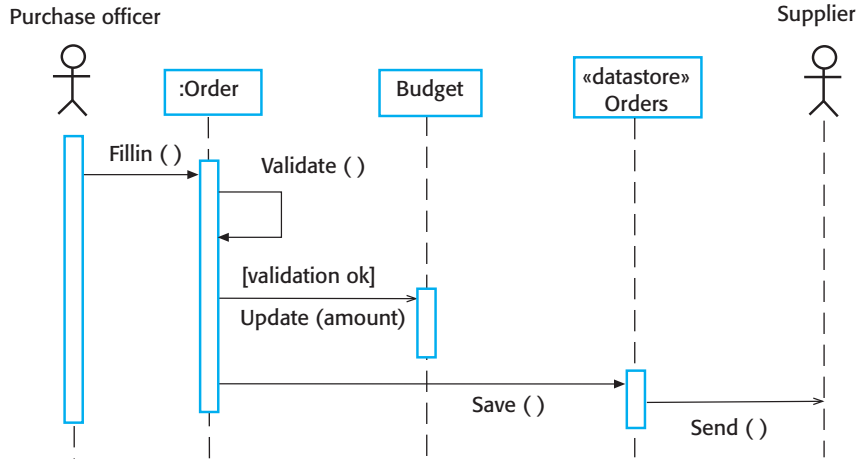


Figure 5.15 Order processing

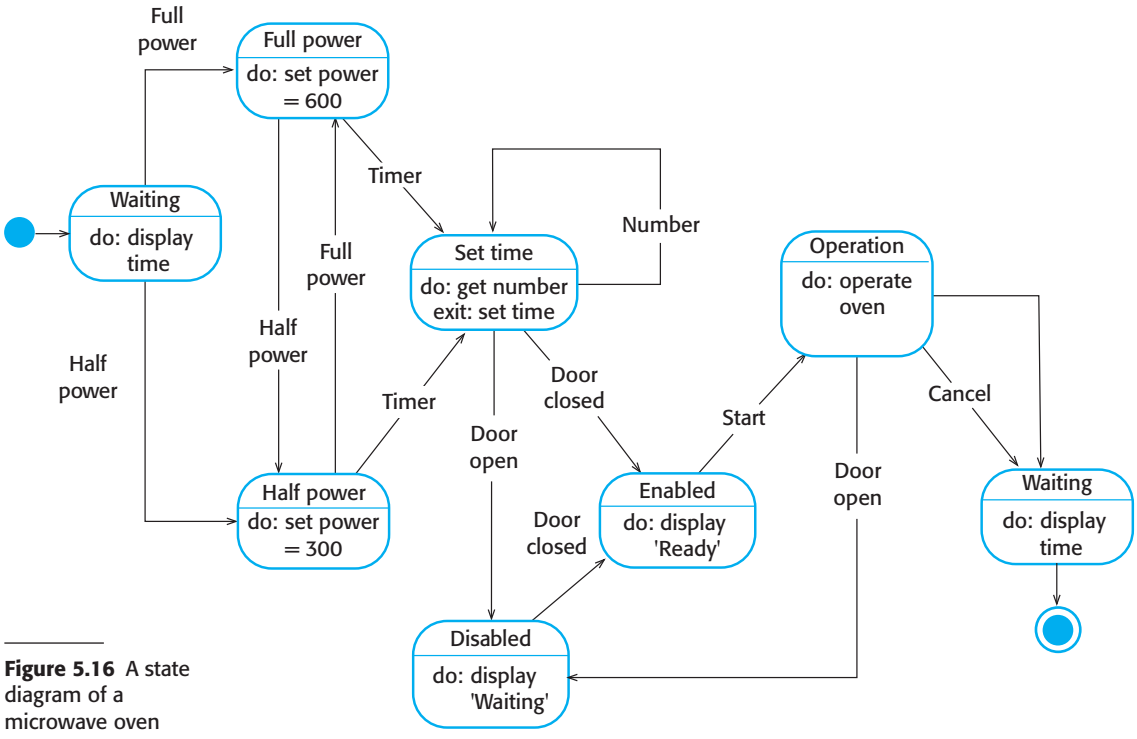


Figure 5.16 A state diagram of a microwave oven

5.4.2 Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received. This view of a system is particularly appropriate for real-time systems. Event-driven modeling is used extensively when designing and documenting real-time systems (Chapter 21).

The UML supports event-based modeling using state diagrams, which are based on Statecharts (Harel 1987). State diagrams show system states and events that cause transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

I use an example of control software for a very simple microwave oven to illustrate event-driven modeling (Figure 5.16). Real microwave ovens are much more complex than this system, but the simplified system is easier to understand. This simple oven has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.

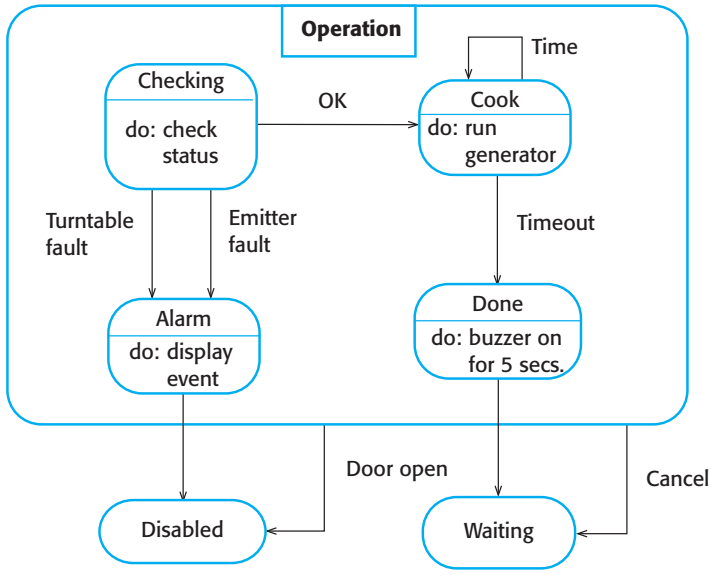


Figure 5.17 A state model of the Operation state

I have assumed that the sequence of actions in using the microwave is as follows:

1. Select the power level (either half power or full power).
2. Input the cooking time using a numeric keypad.
3. Press **Start** and the food is cooked for the given time.

For safety reasons, the oven should not operate when the door is open, and, on completion of cooking, a buzzer is sounded. The oven has a simple display that is used to display various alerts and warning messages.

In UML state diagrams, rounded rectangles represent system states. They may include a brief description (following "do") of the actions taken in that state. The labeled arrows represent stimuli that force a transition from one state to another. You can indicate start and end states using filled circles, as in activity diagrams.

From Figure 5.16, you can see that the system starts in a waiting state and responds initially to either the full-power or the half-power button. Users can change their minds after selecting one of these and may press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation, and cooking takes place for the specified time. This is the end of the cooking cycle, and the system returns to the waiting state.

The problem with state-based modeling is that the number of possible states increases rapidly. For large system models, therefore, you need to hide detail in the models. One way to do this is by using the notion of a "superstate" that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the **Operation** state in Figure 5.16. This is a superstate that can be expanded, as shown in Figure 5.17.

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows "Half power."
Full power	The oven power is set to 600 watts. The display shows "Full power."
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows "Not ready."
Enabled	Oven operation is enabled. Interior oven light is off. Display shows "Ready to cook."
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows "Cooking complete" while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Figure 5.18 States and stimuli for the microwave oven

The Operation state includes a number of substates. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 5.17.

State models of a system provide an overview of event processing, but you normally have to extend this with a more detailed description of the stimuli and the system states. You may use a table to list the states and events that stimulate state transitions along with a description of each state and event. Figure 5.18 shows a tabular description of each state and how the stimuli that force state transitions are generated.

5.4.3 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development whereby models rather than programs are the principal outputs of the development process

(Brambilla, Cabot, and Wimmer 2012). The programs that execute on a hardware/software platform are generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Model-driven engineering was developed from the idea of model-driven architecture (MDA). This was proposed by the Object Management Group (OMG) as a new software development paradigm (Mellor, Scott, and Weise 2004). MDA focuses on the design and implementation stages of software development, whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model-based requirements engineering, software processes for model-based development, and model-based testing are part of MDE but are not considered in MDA.

MDA as an approach to system engineering has been adopted by a number of large companies to support their development processes. This section focuses on the use of MDA for software implementation rather than discuss more general aspects of MDE. The take-up of more general model-driven engineering has been slow, and few companies have adopted this approach throughout their software development life cycle. In his blog, den Haan discusses possible reasons why MDE has not been widely adopted (den Haan 2011).

5.5 Model-driven architecture

Model-driven architecture (Mellor, Scott, and Weise 2004; Stahl and Voelter 2006) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Here, models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

The MDA method recommends that three types of abstract system model should be produced:

1. *A computation independent model (CIM)* CIMs model the important domain abstractions used in a system and so are sometimes called domain models. You may develop several different CIMs, reflecting different views of the system. For example, there may be a security CIM in which you identify important security abstractions such as an asset, and a role and a patient record CIM, in which you describe abstractions such as patients and consultations.

2. *A platform-independent model (PIM)* PIMs model the operation of the system without reference to its implementation. A PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

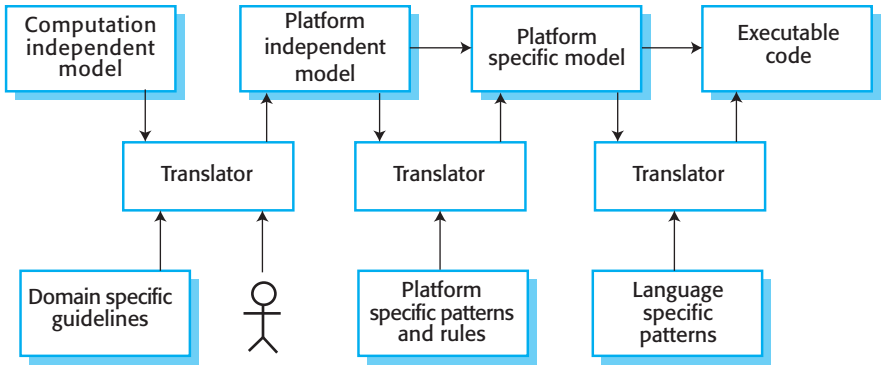


Figure 5.19 MDA transformations

3. *Platform-specific models (PSM)* PSMs are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail. So, the first level PSM could be middleware-specific but database-independent. When a specific database has been chosen, a database-specific PSM can then be generated.

Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to adapt the system to some new platform technology, you write a model translator for that platform. When this is available, all platform-independent models can then be rapidly re-hosted on the new platform.

Fundamental to MDA is the notion that transformations between models can be defined and applied automatically by software tools, as illustrated in Figure 5.19. This diagram also shows a final level of automatic transformation where a transformation is applied to the PSM to generate the executable code that will run on the designated software platform. Therefore, in principle at least, executable software can be generated from a high-level system model.

In practice, completely automated translation of models to code is rarely possible. The translation of high-level CIM to PIM models remains a research problem, and for production systems, human intervention, illustrated using a stick figure in Figure 5.19, is normally required. A particularly difficult problem for automated model transformation is the need to link the concepts used in different CIMS. For example, the concept of a role in a security CIM that includes role-driven access control may have to be mapped onto the concept of a staff member in a hospital CIM. Only a person who understands both security and the hospital environment can make this mapping.

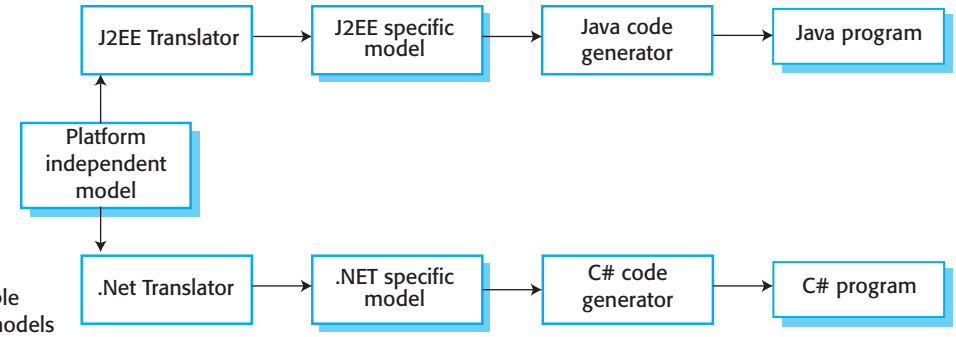


Figure 5.20 Multiple platform-specific models

The translation of platform-independent to platform-specific models is a simpler technical problem. Commercial tools and open-source tools (Koegel 2012) are available that provide translators from PIMS to common platforms such as Java and J2EE. These use an extensive library of platform-specific rules and patterns to convert a PIM to a PSM. There may be several PSMs for each PIM in the system. If a software system is intended to run on different platforms (e.g., J2EE and .NET), then, in principle, you only have to maintain a single PIM. The PSMs for each platform are automatically generated (Figure 5.20).

Although MDA support tools include platform-specific translators, these sometimes only offer partial support for translating PIMS to PSMs. The execution environment for a system is more than the standard execution platform, such as J2EE or Java. It also includes other application systems, specific application libraries that may be created for a company, external services, and user interface libraries.

These vary from one company to another, so off-the-shelf tool support is not available that takes these into account. Therefore, when MDA is introduced into an organization, special-purpose translators may have to be created to make use of the facilities available in the local environment. This is one reason why many companies have been reluctant to take on model-driven approaches to development. They do not want to develop or maintain their own tools or to rely on small software companies, who may go out of business, for tool development. Without these specialist tools, model-based development requires additional manual coding which reduces the cost-effectiveness of this approach.

I believe that there are several other reasons why MDA has not become a mainstream approach to software development.

1. Models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are useful for discussions are the right abstractions for implementation. You may decide to use a completely different implementation approach that is based on the reuse of off-the-shelf application systems.

2. For most complex systems, implementation is not the major problem—requirements engineering, security and dependability, integration with legacy



Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models with clearly defined meanings that can be compiled to executable code. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer 2002).

http://software-engineering-book.com/web/xuml/

systems and testing are all more significant. Consequently, the gains from the use of MDA are limited.

3. The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
4. The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

The success stories for MDA (OMG 2012) have mostly come from companies that are developing systems products, which include both hardware and software. The software in these products has a long lifetime and may have to be modified to reflect changing hardware technologies. The domain of application (automotive, air traffic control, etc.) is often well understood and so can be formalized in a CIM.

Hutchinson and his colleagues (Hutchinson, Rouncefield, and Whittle 2012) report on the industrial use of MDA, and their work confirms that successes in the use of model-driven development have been in systems products. Their assessment suggests that companies have had mixed results when adopting this approach, but the majority of users report that using MDA has increased productivity and reduced maintenance costs. They found that MDA was particularly useful in facilitating reuse, and this led to major productivity improvements.

There is an uneasy relationship between agile methods and model-driven architecture. The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering. Ambler, a pioneer in the development of agile methods, suggests that some aspects of MDA can be used in agile processes (Ambler 2004) but considers automated code generation to be impractical. However, Zhang and Patel report on Motorola's success in using agile development with automated code generation (Zhang and Patel 2011).

KEY POINTS

- A model is an abstract view of a system that deliberately ignores some system details. Complementary system models can be developed to show the system's context, interactions, structure, and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

FURTHER READING

Any of the introductory books on the UML provide more information about the notation than I can cover here. UML has only changed slightly in the last few years, so although some of these books are almost 10 years old, they are still relevant.

Using UML: Software Engineering with Objects and Components, 2nd ed. This book is a short, readable introduction to the use of the UML in system specification and design. I think that it is excellent for learning and understanding the UML notation, although it is less comprehensive than the complete descriptions of UML found in the UML reference manual. (P. Stevens with R. Pooley, Addison-Wesley, 2006)

Model-driven Software Engineering in Practice. This is quite a comprehensive book on model-driven approaches with a focus on model-driven design and implementation. As well as the UML, it also covers the development of domain-specific modeling languages. (M. Brambilla, J. Cabot, and M. Wimmer. Morgan Claypool, 2012)

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

5.1. Scope creep can be defined as a continuous increase in the scope of a project that can significantly increase project cost. Explain how a proper model of the system context can help prevent scope creeps.

5.2. The way in which a system boundary is defined and an appropriate context model is created may have serious implications on the complexity and cost of a project. Give two examples where this may be applicable.

5.3. You have been asked to develop a system that will help with planning large-scale events and parties such as weddings, graduation celebrations, and birthday parties. Using an activity diagram, model the process context for such a system that shows the activities involved in planning a party (booking a venue, organizing invitations, etc.) and the system elements that might be used at each stage.

5.4. For the Mentcare system, propose a set of use cases that illustrates the interactions between a doctor, who sees patients and prescribes medicine and treatments, and the Mentcare system.

5.5. Develop a sequence diagram showing the interactions involved when a student registers for a course in a university. Courses may have limited enrollment, so the registration process must include checks that places are available. Assume that the student accesses an electronic course catalog to find out about available courses.

5.6. Look carefully at how messages and mailboxes are represented in the email system that you use. Model the object classes that might be used in the system implementation to represent a mailbox and an email message.

5.7. Based on your experience with a bank ATM, draw an activity diagram that models the data processing involved when a customer withdraws cash from the machine.

5.8. Draw a sequence diagram for the same system. Explain why you might want to develop both activity and sequence diagrams when modeling the behavior of a system.

5.9. Draw state diagrams of the control software for:

- an automatic washing machine that has different programs for different types of clothes;
- the software for a DVD player;
- the control software for the camera on your mobile phone. Ignore the flash if you have one on your phone.

5.10. In principle, it is possible to generate working programs from a high-level model without manual intervention when using model-driven architectures. Discuss some of the current challenges that stand in the way of the existence of completely automated translation tools.

REFERENCES

Ambler, S. W. 2004. *The Object Primer: Agile Model-Driven Development with UML 2.0, 3rd ed*. Cambridge, UK: Cambridge University Press.

Ambler, S. W., and R. Jeffries. 2002. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.

Booch, G., J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide, 2nd ed*. Boston: Addison-Wesley.

Brambilla, M., J. Cabot, and M. Wimmer. 2012. *Model-Driven Software Engineering in Practice*. San Rafael, CA: Morgan Claypool.

Den Haan, J. 2011. "Why There Is No Future for Model Driven Development." http://www.theenterprisearchitect.eu/archive/2011/01/25/why-there-is-no-future-for-model-driven-development/

Erickson, J,, and K Siau. 2007. "Theoretical and Practical Complexity of Modeling Methods." *Comm. ACM* 50 (8): 46–51. doi:10.1145/1278201.1278205.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

Hull, R., and R King. 1987. "Semantic Database Modeling: Survey, Applications and Research Issues." *ACM Computing Surveys* 19 (3): 201–260. doi:10.1145/45072.45073.

Hutchinson, J., M. Rouncefield, and J. Whittle. 2012. "Model-Driven Engineering Practices in Industry." In *34th Int. Conf. on Software Engineering*, 633–642. doi:10.1145/1985793.1985882.

Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.

Koegel, M. 2012. "EMF Tutorial: What Every Eclipse Developer Should Know about EMF." http://eclipsesource.com/blogs/tutorials/emf-tutorial/

Mellor, S. J., and M. J. Balcer. 2002. *Executable UML*. Boston: Addison-Wesley.

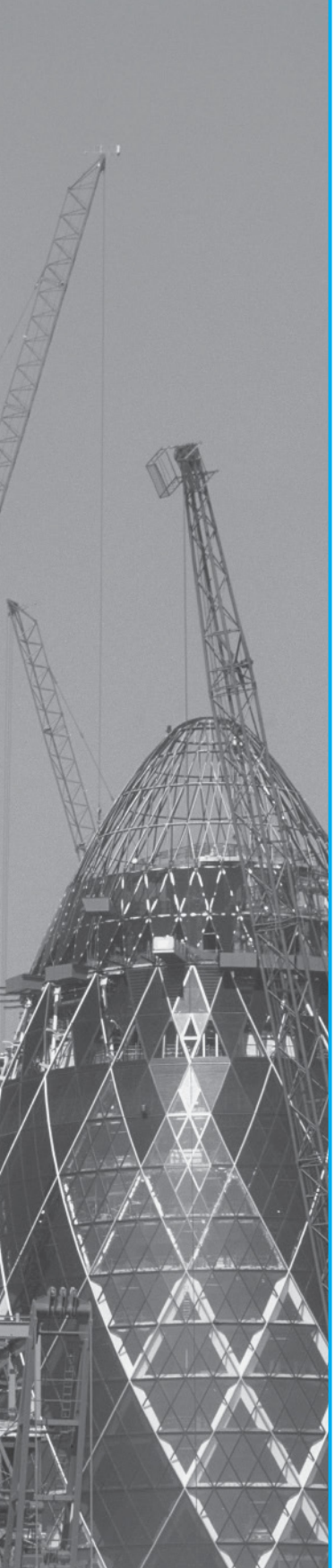
Mellor, S. J., K. Scott, and D. Weise. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley.

OMG. 2012. "Model-Driven Architecture: Success Stories." http://www.omg.org/mda/products_success.htm

Rumbaugh, J., I. Jacobson, and G Booch. 2004. *The Unified Modelling Language Reference Manual, 2nd ed.* Boston: Addison-Wesley.

Stahl, T., and M. Voelter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.

Zhang, Y., and S. Patel. 2011. "Agile Model-Driven Development in Practice." *IEEE Software* 28 (2): 84–91. doi:10.1109/MS.2010.85.



6 Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the software architecture during the architectural design process;
- have been introduced to the idea of Architectural patterns, well-tried ways of organizing software architectures that can be reused in system designs;
- understand how Application-Specific Architectural patterns may be used in transaction processing and language processing systems.

Contents

6.1 Architectural design decisions
6.2 Architectural views
6.3 Architectural patterns
6.4 Application architectures

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system. In the model of the software development process that I described in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of an agile development process should focus on designing an overall system architecture. Incremental development of architectures is not usually successful. Refactoring components in response to changes is usually relatively easy. However, refactoring the system architecture is expensive because you may need to modify most system components to adapt them to the architectural changes.

To help you understand what I mean by system architecture, look at Figure 6.1. This diagram shows an abstract model of the architecture for a packing robot system. This robotic system can pack different kinds of objects. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not

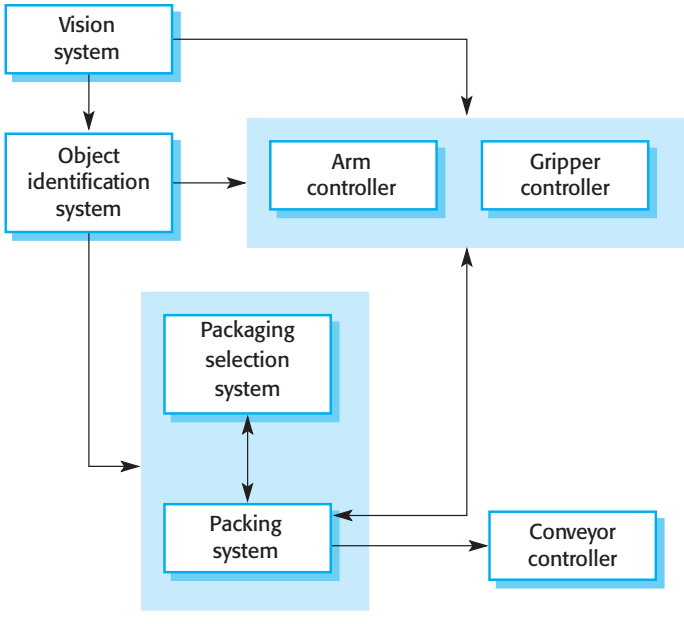


Figure 6.1 The architecture of a packing robot control system

include any design information. This ideal is unrealistic, however, except for very small systems. You need to identify the main architectural components as these reflect the high-level features of the system. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or subsystems. You then use this decomposition to discuss the requirements and more detailed features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large:*

1. *Architecture in the small* is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.
2. *Architecture in the large* is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems may be distributed over different computers, which may be owned and managed by different companies. (I cover architecture in the large in Chapters 17 and 18.)

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000). As Bosch explains, individual components implement the functional system requirements, but the dominant influence on the non-functional system characteristics is the system's architecture. Chen et al. (Chen, Ali Babar, and Nuseibeh 2013) confirmed this in a study of "architecturally significant requirements" where they found that non-functional requirements had the most significant effect on the system's architecture.

Bass et al. (Bass, Clements, and Kazman 2012) suggest that explicitly designing and documenting software architecture has three advantages:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. *Large-scale reuse* An architectural model is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 15, product-line architectures are an approach to reuse where the same architecture is reused across a range of related systems.

System architectures are often modeled informally using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to subcomponents. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's handbook of software architecture (Booch 2014).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. In spite of their widespread use, Bass et al. (Bass, Clements, and Kazman 2012) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

1. *As a way of encouraging discussions about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so that managers can start assigning people to plan the development of these systems.
2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections. The argument for such a model is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are a good way of supporting communications between the people involved in the software design process. They are intuitive, and domain experts and software engineers can relate to them and participate in discussions about the system. Managers find them helpful in planning the project. For many projects, block diagrams are the only architectural description.

Ideally, if the architecture of a system is to be documented in detail, it is better to use a more rigorous notation for architectural description. Various architectural description languages (Bass, Clements, and Kazman 2012) have been developed for this purpose. A more detailed and complete description means that there is less scope for misunderstanding the relationships between the architectural components. However, developing a detailed architectural description is an expensive and time-consuming process. It is practically impossible to know whether or not it is cost-effective, so this approach is not widely used.

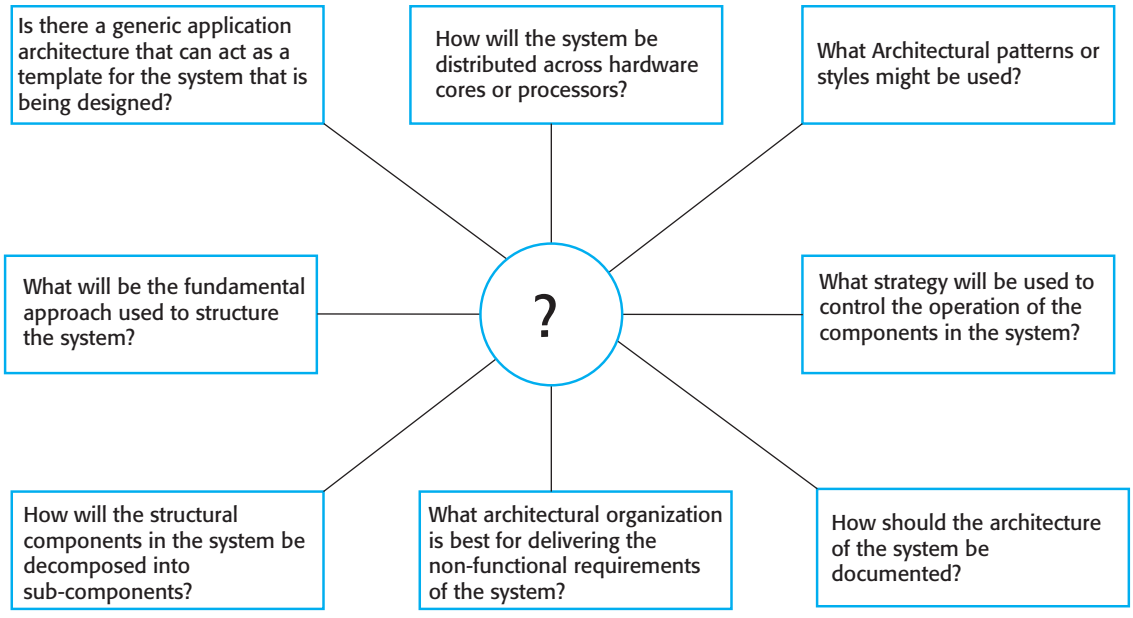


Figure 6.2 Architectural design decisions

6.1 Architectural design decisions

Architectural design is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system. There is no formulaic architectural design process. It depends on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. Consequently, I think it is best to consider architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the fundamental questions shown in Figure 6.2.

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse.

For embedded systems and apps designed for personal computers and mobile devices, you do not have to design a distributed architecture for the system. However, most large systems are distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a

key decision that affects the performance and reliability of the system. This is a major topic in its own right that I cover in Chapter 17.

The architecture of a software system may be based on a particular Architectural pattern or style (these terms have come to mean the same thing). An Architectural pattern is a description of a system organization (Garlan and Shaw 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I cover several frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style covers questions 4 to 6 in the list of fundamental architectural questions shown in Figure 6.2. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on a strategy for decomposing components into subcomponents. Finally, in the control modeling process, you develop a general model of the control relationships between the various parts of the system and make decisions about how the execution of components is controlled.

Because of the close relationship between non-functional system characteristics and software architecture, the choice of architectural style and structure should depend on the non-functional requirements of the system:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, finer-grain components. Using large components reduces the number of component communications, as most of the interactions between related system features take place within a component. You may also consider runtime system organizations that allow the system to be replicated and executed on different processors.

2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and a high level of security validation applied to these layers.

3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are co-located in a single component or in a small number of components. This reduces the costs and problems of safety validation and may make it possible to provide related protection systems that can safely shut down the system in the event of failure.

4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe fault-tolerant system architectures for high-availability systems in Chapter 11.

5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers, and shared data structures should be avoided.

Obviously, there is potential conflict between some of these architectures. For example, using large components improves performance, and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, however, then some compromise must be found. You can sometimes do this by using different Architectural patterns or styles for separate parts of the system. Security is now almost always a critical requirement, and you have to design an architecture that maintains security while also satisfying other non-functional requirements.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic Architectural patterns. Bosch's description (Bosch 2000) of the non-functional characteristics of some Architectural patterns can help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single diagram, as a graphical model can only show one view or perspective of the system. It might show how a system is decomposed into modules, how the runtime processes interact, or the different ways in which system components are distributed across a network. Because all of these are useful at different times, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (Krutchen 1995) in his well-known 4+1 view model of software architecture, suggests that there should

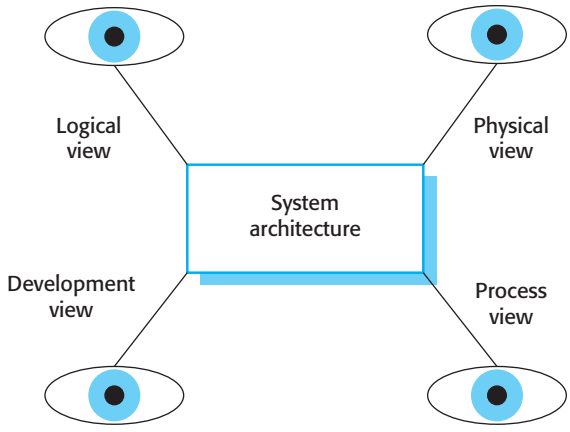


Figure 6.3 Architectural views

be four fundamental architectural views, which can be linked through common use cases or scenarios (Figure 6.3). He suggests the following views:

1. *A logical view,* which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.
2. *A process view,* which shows how, at runtime, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. *A development view,* which shows how the software is decomposed for development; that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. *A physical view,* which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (Hofmeister, Nord, and Soni 2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 15) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views of a system's architecture are almost always developed during the design process. They are used to explain the system architecture to stakeholders and to inform architectural decision making. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but it is rarely necessary to develop a complete description from all perspectives. It may also be possible to associate Architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for describing and documenting software architectures. A survey in 2006 (Lange, Chaudron, and Muskens 2006) showed that, when the UML was used, it was mostly applied in an informal way. The authors of that paper argued that this was a bad thing.

I disagree with this view. The UML was designed for describing object-oriented systems, and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description. I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and that can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers (Bass, Clements, and Kazman 2012) have proposed the use of more specialized architectural description languages (ADLs) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because ADLs are specialist languages, domain and application specialists find it hard to understand and use ADLs. There may be some value in using domain-specific ADLs as part of model-driven development, but I do not think they will become part of mainstream software engineering practice. Informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop these documents. I largely agree with this view, and I think that, except for critical systems, it is not worth developing a detailed architectural description from Krutchen's four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al. 1995). This prompted the development of other types of patterns, such as patterns for organizational design (Coplien and Harrison 2004), usability patterns (Usability Group 1998), patterns of cooperative interaction (Martin and Sommerville 2004), and configuration management patterns (Berczuk and Appleton 2002).

Architectural patterns were proposed in the 1990s under the name "architectural styles" (Shaw and Garlan 1996). A very detailed five-volume series of handbooks on pattern-oriented software architecture was published between 1996 and 2007 (Buschmann et al. 1996; Schmidt et al. 2000; Buschmann, Henney, and Schmidt 2007a, 2007b; Kircher and Jain 2004).

In this section, I introduce Architectural patterns and briefly describe a selection of Architectural patterns that are commonly used. Patterns may be described in a standard way (Figures 6.4 and 6.5) using a mixture of narrative description and diagrams.

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.5.
Example	Figure 6.6 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them.
Disadvantages	May involve additional code and code complexity when the data model and interactions are simple.

Figure 6.4 The Model-View-Controller (MVC) pattern

For more detailed information about patterns and their use, you should refer to the published pattern handbooks.

You can think of an Architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. So, an Architectural pattern should describe a system organization that has been successful in previous systems. It should include information on when it is and is not appropriate to use that pattern, and details on the pattern's strengths and weaknesses.

Figure 6.4 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. The stylized pattern description includes the pattern

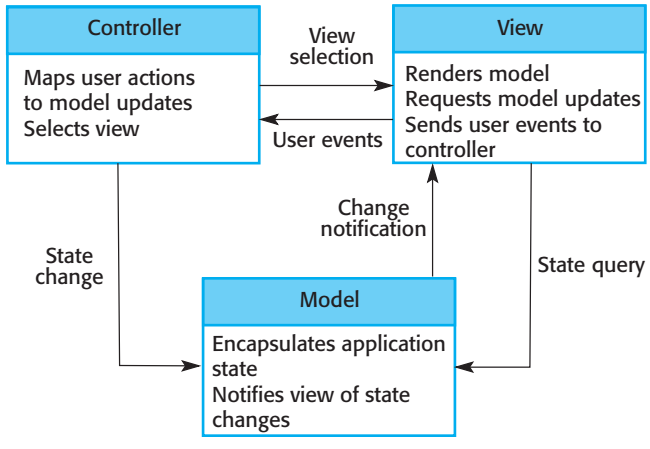


Figure 6.5 The organization of the Model-View-Controller

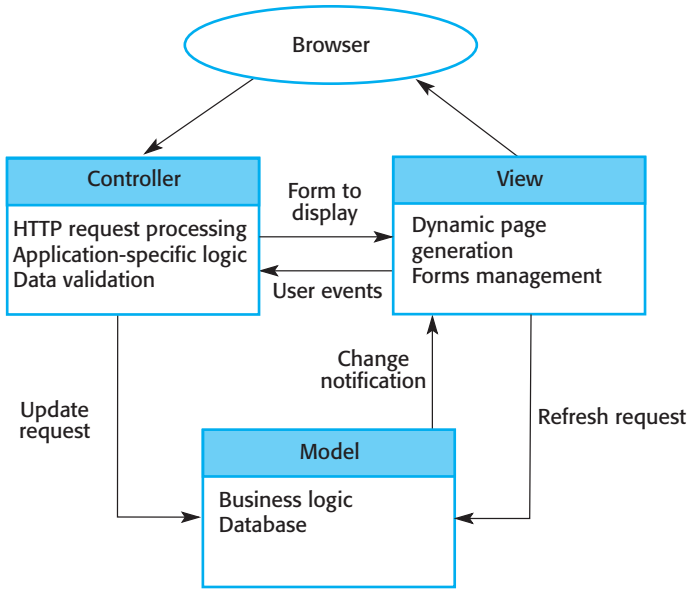


Figure 6.6 Web application architecture using the MVC pattern

name, a brief description, a graphical model, and an example of the type of system where the pattern is used. You should also include information about when the pattern should be used and its advantages and disadvantages.

Graphical models of the architecture associated with the MVC pattern are shown in Figures 6.5 and 6.6. These present the architecture from different views: Figure 6.5 is a conceptual view, and Figure 6.6 shows a runtime system architecture when this pattern is used for interaction management in a web-based system.

In this short space, it is impossible to describe all of the generic patterns that can be used in software development. Instead, I present some selected examples of patterns that are widely used and that capture good architectural design principles.

6.3.1 Layered architecture

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The MVC pattern, shown in Figure 6.4, separates elements of a system, allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The Layered Architecture pattern is another way of achieving separation and independence. This pattern is shown in Figure 6.7. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. If its interface is unchanged, a new layer with extended functionality can replace an existing layer

Name	Layered architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Figure 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Figure 6.7 The Layered Architecture pattern

without changing other parts of the system. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine dependencies, this makes it easier to provide multi-platform implementations of an application system. Only the machine-dependent layers need be reimplemented to take account of the facilities of a different operating system or database.

Figure 6.8 is an example of a layered architecture with four layers. The lowest layer includes system support software—typically, database and operating system support. The next layer is the application layer, which includes the components concerned with the application functionality and utility components used by other application components.

The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities. Of course, the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.

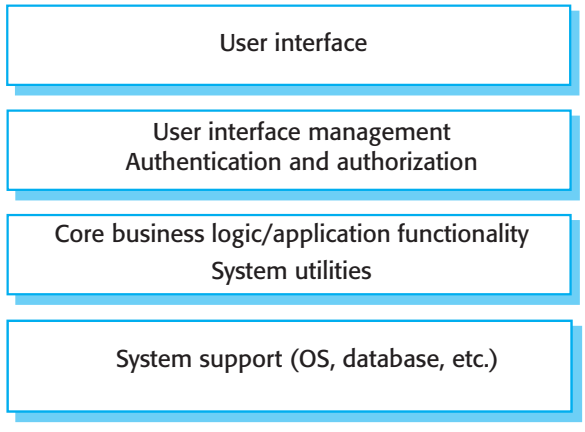


Figure 6.8 A generic layered architecture

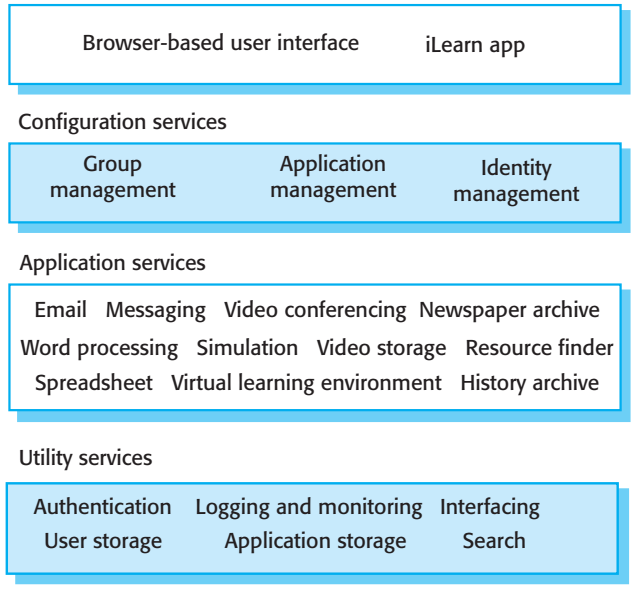


Figure 6.9 The architecture of the iLearn system

Figure 6.9 shows that the iLearn digital learning system, introduced in Chapter 1, has a four-layer architecture that follows this pattern. You can see another example of the Layered Architecture pattern in Figure 6.19 (Section 6.4, which shows the organization of the Mentcare system.

6.3.2 Repository architecture

The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system. My next example, the Repository pattern (Figure 6.10), describes how a set of interacting components can share data.

Figure 6.10 The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.11 is an example of an IDE where the components use a repository of system design information. Each software tool generates information, which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

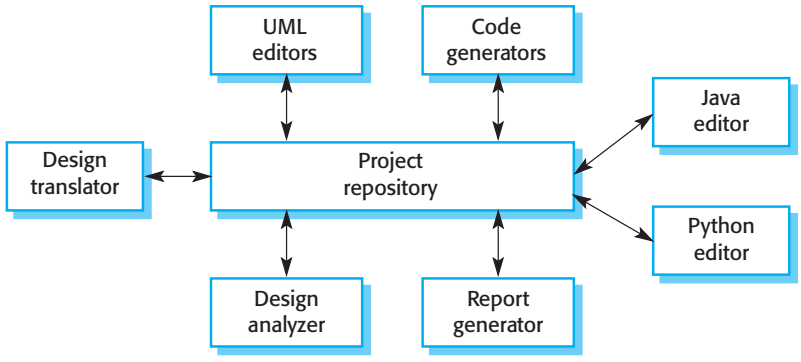


Figure 6.11 A repository architecture for an IDE

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, Computer-Aided Design (CAD) systems, and interactive development environments for software.

Figure 6.11 illustrates a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment (as discussed in Chapter 25) that keeps track of changes to software and allows rollback to earlier versions.

Organizing tools around a repository is an efficient way of sharing large amounts of data. There is no need to transmit data explicitly from one component to another. However, components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool, and it may be difficult or impossible to integrate new components if their data models do not fit the agreed schema. In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.

In the repository architecture shown in Figure 6.11, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for artificial intelligence (AI) systems, uses a "blackboard" model that triggers components when particular data become available. This is appropriate when the data in the repository is unstructured. Decisions about which tool is to be activated can only be made when the data has been analyzed. This model was introduced by Nii (Nii 1986), and Bosch (Bosch 2000) includes a good discussion of how this style relates to system quality attributes.

6.3.3 Client–server architecture

The Repository pattern is concerned with the static structure of a system and does not show its runtime organization. My next example, the Client–Server pattern (Figure 6.12), illustrates a commonly used runtime organization for distributed

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Figure 6.12 The Client–Server pattern

systems. A system that follows the Client–Server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server that offers programming language compilation services. Servers are software components, and several servers may run on the same computer.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Client–server systems are usually implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures, but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request–reply protocol (such as http), where a client makes a request to a server and waits until it receives a reply from that server.

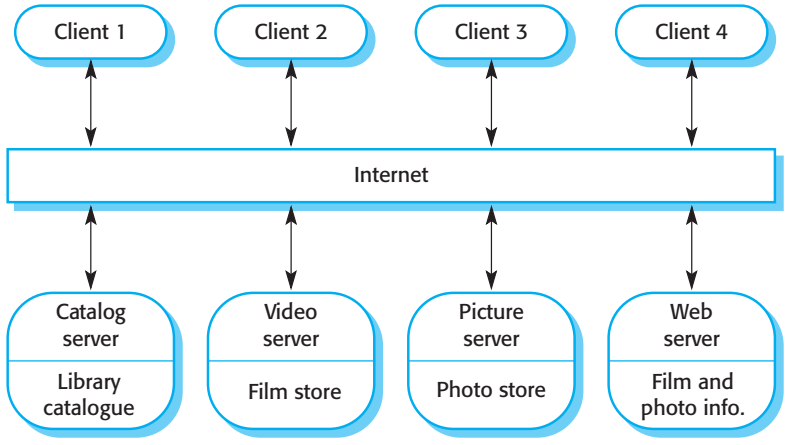


Figure 6.13 A client–server architecture for a film library

Figure 6.13 is an example of a system that is based on the client–server model. This is a multiuser, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

The catalog must be able to deal with a variety of queries and provide links into the web information system that include data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The client program is simply an integrated user interface, constructed using a web browser, to access these services.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system. I cover distributed architectures in Chapter 17, where I explain the client–server model and its variants in more detail.

6.3.4 Pipe and filter architecture

My final example of a general Architectural pattern is the Pipe and Filter pattern (Figure 6.14). This is a model of the runtime organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.15 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

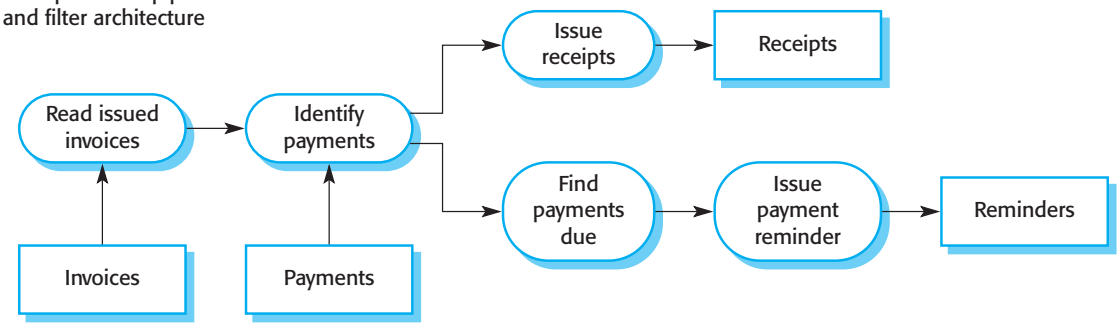
Figure 6.14 The Pipe and Filter pattern

The name "pipe and filter" comes from the original Unix system where it was possible to link processes using "pipes." These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term *filter* is used because a transformation "filters out" the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data-processing systems such as billing systems. The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I cover use of this pattern in embedded systems in Chapter 21.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.15. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

Figure 6.15 An example of the pipe and filter architecture





Architectural patterns for control

There are specific Architectural patterns that reflect commonly used ways of organizing control in a system. These include centralized control, based on one component calling other components, and event-based control, where the system reacts to external events.

http://software-engineering-book.com/web/archpatterns/

those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Pipe and filter systems are best suited to batch processing systems and embedded systems where there is limited user interaction. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. While simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections. It is difficult to implement this as a sequential stream that conforms to the pipe and filter model.

6.4 Application architectures

Application systems are intended to meet a business or an organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector-specific applications. Therefore, as well as general business functions, all phone companies need systems to connect and meter calls, manage their network and issue bills to customers. Consequently, the application systems used by these businesses also have much in common.

These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architecture may be reimplemented when developing new systems. However, for many business systems, application architecture reuse is implicit when generic application systems are configured to create a new application. We see this in the widespread use of Enterprise Resource Planning (ERP) systems and off-the-shelf configurable application systems, such as systems for accounting and stock control. These systems have a standard architecture and components. The components are configured and adapted to create a specific business application.



Application architectures

There are several examples of application architectures on the book's website. These include descriptions of batch data-processing systems, resource allocation systems, and event-based editing systems.

http://software-engineering-book.com/web/apparch/

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements.

As a software designer, you can use models of application architectures in a number of ways:

1. *As a starting point for the architectural design process* If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. You then specialize this for the specific system that is being developed.
2. *As a design checklist* If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.
3. *As a way of organizing the work of the development team* The application architectures identify stable structural features of the system architectures, and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.
4. *As a means of assessing components for reuse* If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.
5. *As a vocabulary for talking about applications* If you are discussing a specific application or trying to compare applications, then you can use the concepts identified in the generic architecture to talk about these applications.

There are many types of application system, and, in some cases, they may seem to be very different. However, superficially dissimilar applications may have much in common and thus share an abstract application architecture. I illustrate this by describing the architectures of two types of application:

1. *Transaction processing applications* Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common types of interactive business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained. This class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

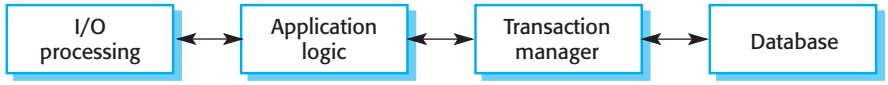


Figure 6.16 The structure of transaction processing applications

2. *Language processing systems* Language processing systems are systems in which the user's intentions are expressed in a formal language, such as a programming language. The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers, which translate high-level language programs into machine code. However, language processing systems are also used to interpret command languages for databases and information systems, and markup languages such as XML.

I have chosen these particular types of system because a large number of web-based business systems are transaction processing systems, and all software development relies on language processing systems.

6.4.1 Transaction processing systems

Transaction processing systems are designed to process user requests for information from a database, or requests to update a database (Lewis, Bernstein, and Kifer 2003). Technically, a database transaction is part of a sequence of operations and is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within a transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as "find the times of flights from London to Paris." If the user transaction does not require the database to be changed, then it may not be necessary to package this as a technical database transaction.

An example of a database transaction is a customer request to withdraw money from a bank account using an ATM. This involves checking the customer account balance to see if sufficient funds are available, modifying the balance by the amount withdrawn and sending commands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

Transaction processing systems are usually interactive systems in which users make asynchronous requests for service. Figure 6.16 illustrates the conceptual architectural structure of transaction processing applications. First, a user makes a request to the system through an I/O processing component. The request is processed by some application-specific logic. A transaction is created and passed to a transaction manager, which is usually embedded in the database management system. After the transaction manager has ensured that the transaction is properly completed, it signals to the application that processing has finished.

Transaction processing systems may be organized as a "pipe and filter" architecture, with system components responsible for input, processing, and output. For

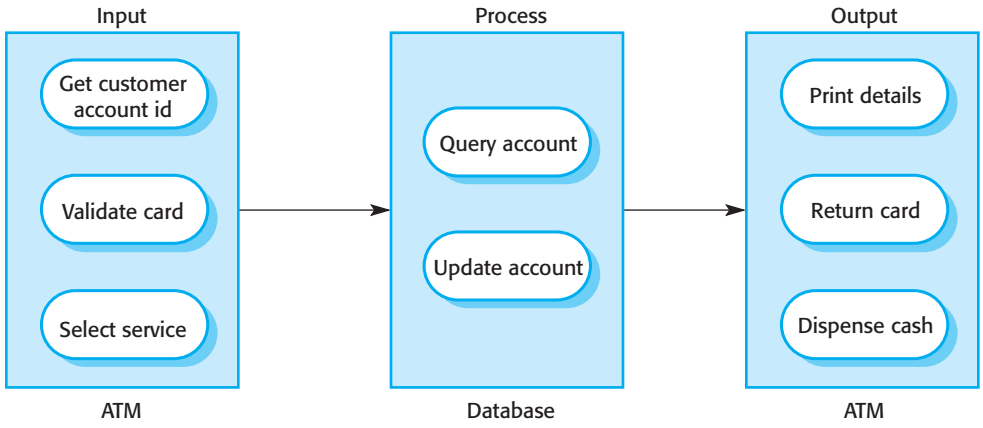


Figure 6.17 The software architecture of an ATM system

example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server. The input and output components are implemented as software in the ATM, and the processing component is part of the bank's database server. Figure 6.17 shows the architecture of this system, illustrating the functions of the input, process, and output components.

6.4.2 Information systems

All systems that involve interaction with a shared database can be considered to be transaction-based information systems. An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital. Information systems are almost always web-based systems, where the user interface is implemented in a web browser.

Figure 6.18 presents a very general model of an information system. The system is modeled using a layered approach (discussed in Section 6.3) where the top layer

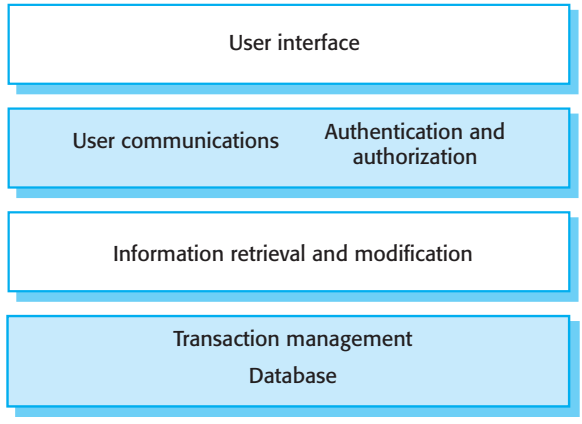


Figure 6.18 Layered information system architecture

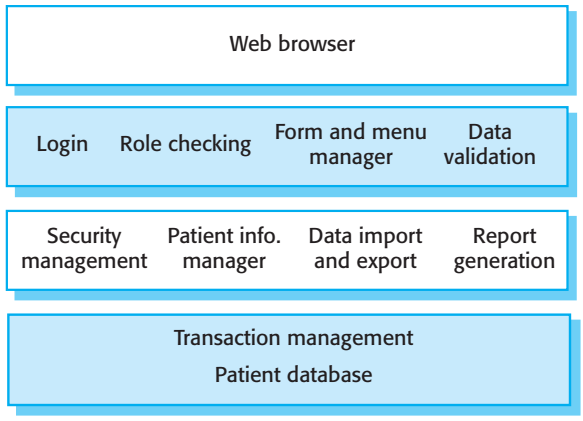


Figure 6.19 The architecture of the Mentcare system

supports the user interface and the bottom layer is the system database. The user communications layer handles all input and output from the user interface, and the information retrieval layer includes application-specific logic for accessing and updating the database. The layers in this model can map directly onto servers in a distributed Internet-based system.

As an example of an instantiation of this layered model, Figure 6.19 shows the architecture of the Mentcare system. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:

1. The top layer is a browser-based user interface.
2. The second layer provides the user interface functionality that is delivered through the web browser. It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
4. Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

Information and resource management systems are sometimes also transaction processing systems. For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer. In an e-commerce

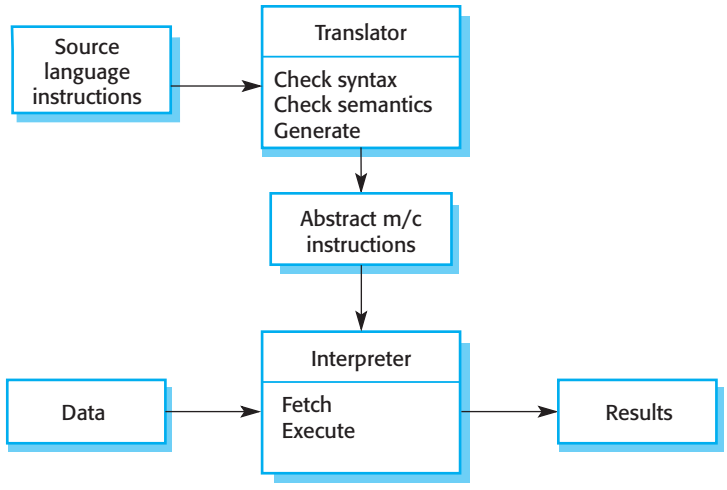


Figure 6.20 The architecture of a language processing system

system, the application-specific layer includes additional functionality supporting a "shopping cart" in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

The organization of servers in these systems usually reflects the four-layer generic model presented in Figure 6.18. These systems are often implemented as distributed systems with a multitier client server/architecture

1. The web server is responsible for all user communications, with the user interface implemented using a web browser;
2. The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
3. The database server moves information to and from the database and handles transaction management.

Using multiple servers allows high throughput and makes it possible to handle thousands of transactions per minute. As demand increases, servers can be added at each level to cope with the extra processing involved.

6.4.3 Language processing systems

Language processing systems translate one language into an alternative representation of that language and, for programming languages, may also execute the resulting code. Compilers translate a programming language into machine code. Other language processing systems may translate an XML data description into commands to query a database or to an alternative XML representation. Natural language processing systems may translate one natural language to another, for example, French to Norwegian.

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.20. The source language instructions define the

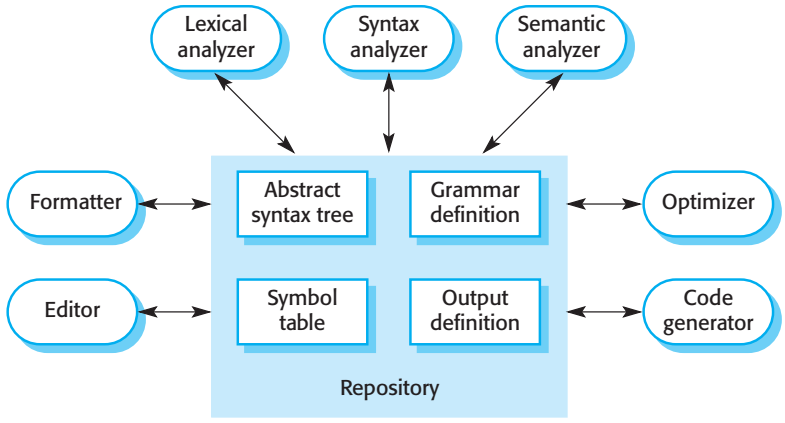


Figure 6.21 A repository architecture for a language processing system

program to be executed, and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

For many compilers, the interpreter is the system hardware that processes machine instructions, and the abstract machine is a real processor. However, for dynamically typed languages, such as Ruby or Python, the interpreter is a software component.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.21) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them into an internal form.
2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
4. A syntax tree, which is an internal structure representing the program being compiled.
5. A semantic analyzer, which uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
6. A code generator, which "walks" the syntax tree and generates abstract machine code.

Other components might also be included that analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.



Reference architectures

Reference architectures capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture, although, in reality, it is very unlikely that any individual application would include all the features shown in a reference architecture. The main purpose of reference architectures is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain.

http://software-engineering-book.com/web/refarch/

In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary. The output of the system is translation of the input text.

Figure 6.21 illustrates how a language processing system can be part of an integrated set of programming support tools. In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository. Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed. A program formatter can create listings of the program that highlight different syntactic elements and are therefore easier to read and understand.

Alternative Architectural patterns may be used in a language processing system (Garlan and Shaw 1993). Compilers can be implemented using a composite of a repository and a pipe and filter model. In a compiler architecture, the symbol table is a repository for shared data. The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown in Figure 6.22, and communicate through the shared symbol table.

This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another. It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger, or a program formatter. In this situation, changes from one component need to be reflected immediately in other components. It is better to organize the system around a repository, as shown in Figure 6.21 if you are implementing a general, language-oriented programming environment.

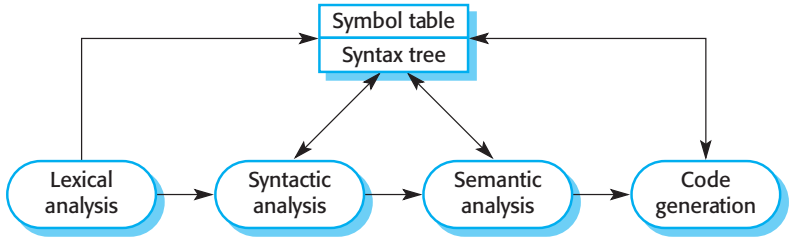


Figure 6.22 A pipe and filter compiler architecture

KEY POINTS

- A software architecture is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used, and the ways in which the architecture should be documented and evaluated.
- Architectures may be documented from several different perspectives or views. Possible views include a conceptual view, a logical view, a process view, a development view, and a physical view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used, and point out its advantages and disadvantages.
- Commonly used Architectural patterns include model-view-controller, layered architecture, repository, client–server, and pipe and filter.
- Generic models of application systems architectures help us understand the operation of applications, compare applications of the same type, validate application system designs, and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users. Information systems and resource management systems are examples of transaction processing systems.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

FURTHER READING

Software Architecture: Perspectives on an Emerging Discipline. This was the first book on software architecture and has a good discussion on different architectural styles that is still relevant. (M. Shaw and D. Garlan, 1996, Prentice-Hall).

"The Golden Age of Software Architecture." This paper surveys the development of software architecture from its beginnings in the 1980s through to its usage in the 21st century. There is not a lot of technical content, but it is an interesting historical overview. (M. Shaw and P. Clements, *IEEE Software,* 21 (2), March–April 2006) http://doi.dx.org/10.1109/MS.2006.58.

Software Architecture in Practice (3rd ed.). This is a practical discussion of software architectures that does not oversell the benefits of architectural design. It provides a clear business rationale, explaining why architectures are important. (L. Bass, P. Clements, and R. Kazman, 2012, Addison-Wesley).

Handbook of Software Architecture. This is a work in progress by Grady Booch, one of the early evangelists for software architecture. He has been documenting the architectures of a range of software systems so that you can see reality rather than academic abstraction. Available on the web and intended to appear as a book. (G. Booch, 2014) http://www.handbookofsoftwarearchitecture.com/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

6.1. When describing a system, explain why you may have to start the design of the system architecture before the requirements specification is complete.

6.2. You have been asked to prepare and deliver a presentation to a nontechnical manager to justify the hiring of a system architect for a new project. Write a list of bullet points setting out the key points in your presentation in which you explain the importance of software architecture.

6.3. Performance and security may pose to be conflicting non-functional requirements when architecting software systems. Make an argument in support of this statement.

6.4. Draw diagrams showing a conceptual view and a process view of the architectures of the following systems:

A ticket machine used by passengers at a railway station.

A computer-controlled video conferencing system that allows video, audio, and computer data to be visible to several participants at the same time.

A robot floor-cleaner that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.

6.5. A software system will be built to allow drones to autonomously herd cattle in farms. These drones can be remotely controlled by human operators. Explain how multiple architectural patterns can fit together to help build this kind of system.

6.6. Suggest an architecture for a system (such as iTunes) that is used to sell and distribute music on the Internet. What Architectural patterns are the basis for your proposed architecture?

6.7. An information system is to be developed to maintain information about assets owned by a utility company such as buildings, vehicles, and equipment. It is intended that this will be

updatable by staff working in the field using mobile devices as new asset information becomes available. The company has several existing asset databases that should be integrated through this system. Design a layered architecture for this asset management system based on the generic information system architecture shown in Figure 6.18.

6.8. Using the generic model of a language processing system presented here, design the architecture of a system that accepts natural language commands and translates these into database queries in a language such as SQL.

6.9. Using the basic model of an information system, as presented in Figure 6.18, suggest the components that might be part of an information system that allows users to view box office events, available tickets and prices, and to eventually buy tickets.

6.10. Should there be a separate profession of 'software architect' whose role is to work independently with a customer to design the software system architecture? A separate software company would then implement the system. What might be the difficulties of establishing such a profession?

REFERENCES

Bass, L., P. Clements, and R. Kazman. 2012. *Software Architecture in Practice (3rd ed.)*. Boston: Addison-Wesley.

Berczuk, S. P., and B. Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley.

Booch, G. 2014. "Handbook of Software Architecture." http://handbookofsoftwarearchitecture.com/

Bosch, J. 2000. *Design and Use of Software Architectures*. Harlow, UK: Addison-Wesley.

Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.

———. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.

Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.

Chen, L., M. Ali Babar, and B. Nuseibeh. 2013. "Characterizing Architecturally Significant Requirements." *IEEE Software* 30 (2): 38–45. doi:10.1109/MS.2012.174.

Coplien, J. O., and N. B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice-Hall.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., and M. Shaw. 1993. "An Introduction to Software Architecture." In *Advances in Software Engineering and Knowledge Engineering*, edited by V. Ambriola and G. Tortora, 2:1–39. London: World Scientific Publishing Co.

Hofmeister, C., R. Nord, and D. Soni. 2000. *Applied Software Architecture*. Boston: Addison-Wesley.

Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Krutchen, P. 1995. "The 4+1 View Model of Software Architecture." *IEEE Software* 12 (6): 42–50. doi:10.1109/52.469759.

Lange, C. F. J., M. R. V. Chaudron, and J. Muskens. 2006. "UML Software Architecture and Design Description." *IEEE Software* 23 (2): 40–46. doi:10.1109/MS.2006.50.

Lewis, P. M., A. J. Bernstein, and M. Kifer. 2003. *Databases and Transaction Processing: An Application-Oriented Approach*. Boston: Addison-Wesley.

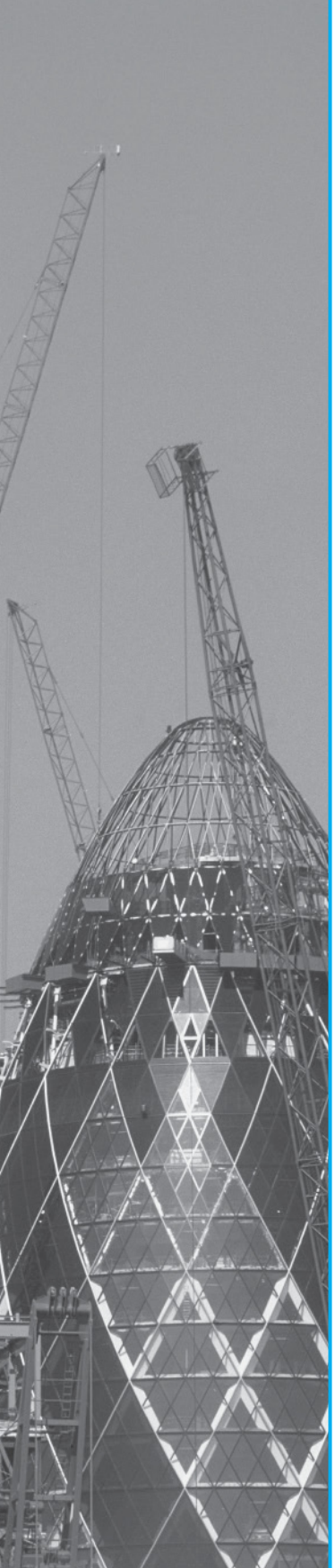
Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.

Nii, H. P. 1986. "Blackboard Systems, Parts 1 and 2." *AI Magazine* 7 (2 and 3): 38–53 and 62–69. http://www.aaai.org/ojs/index.php/aimagazine/article/view/537/473

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall.

Usability Group. 1998. "Usability Patterns". University of Brighton. http://www.it.bton.ac.uk/Research/patterns/home.html



7 Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

7.1 Object-oriented design using the UML
7.2 Design patterns
7.3 Implementation issues
7.4 Open-source development

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. For some simple systems, software engineering means software design and implementation and all other software engineering activities are merged with this process. However, for large systems, software design and implementation is only one of a number of software engineering processes (requirements engineering, verification and validation, etc.).

Software design and implementation activities are invariably interleaved. Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program. Sometimes there is a separate design stage, and this design is modeled and documented. At other times, a design is in the programmer's head or roughly sketched on a whiteboard or sheets of paper. Design is about how to solve a problem, so there is always a design process. However, it isn't always necessary or appropriate to describe the design in detail using the UML or other design description language.

Design and implementation are closely linked, and you should normally take implementation issues into account when developing a design. For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#. It is less useful, I think, if you are developing using a dynamically typed language like Python. There is no point in using the UML if you are implementing your system by configuring an off-the-shelf package. As I discussed in Chapter 3, agile methods usually work from informal sketches of the design and leave design decisions to programmers.

One of the most important implementation decisions that has to be made at an early stage of a software project is whether to build or to buy the application software. For many types of application, it is now possible to buy off-the-shelf application systems that can be adapted and tailored to the users' requirements. For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It is usually cheaper and faster to use this approach rather than developing a new system in a conventional programming language.

When you develop an application system by reusing an off-the-shelf product, the design process focuses on how to configure the system product to meet the application requirements. You don't develop design models of the system, such as models of the system objects and their interactions. I discuss this reuse-based approach to development in Chapter 15.

I assume that most readers of this book have had experience of program design and implementation. This is something that you acquire as you learn to program and master the elements of a programming language like Java or Python. You will have probably learned about good programming practice in the programming languages that you have studied, as well as how to debug programs that you have developed. Therefore, I don't cover programming topics here. Instead, this chapter has two aims:

1. To show how system modeling and architectural design (covered in Chapters 5 and 6) are put into practice in developing an object-oriented software design.

2. To introduce important implementation issues that are not usually covered in programming books. These include software reuse, configuration management and open-source development.

As there are a vast number of different development platforms, the chapter is not biased toward any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than a programming language such as Java or Python.

7.1 Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, you need to:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Sometimes you use notations, such as the UML, precisely to clarify aspects of the design; at other times, notations are used informally to stimulate discussions.

I explain object-oriented software design by developing a design for part of the embedded software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each weather station

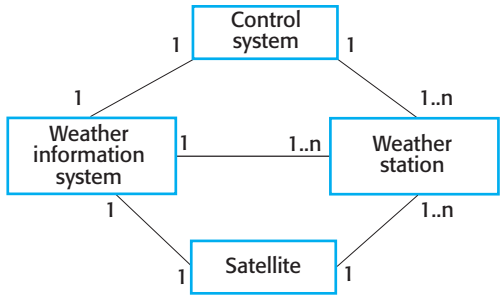


Figure 7.1 System context for the weather station

records local weather information and periodically transfers this to a weather information system, using a satellite link.

7.1.1 System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. As I discussed in Chapter 5, understanding the context also lets you establish the boundaries of the system.

Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations. Figure 7.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model. As I discussed in Chapters 4 and 5, each use case represents



Weather station use cases

Report weather–send weather data to the weather information system

Report status–send status information to the weather information system

Restart–if the weather station is shut down, restart the system

Shutdown–shut down the weather station

Reconfigure–reconfigure the weather station software

Powersave–put the weather station into power-saving mode

Remote control–send control commands to any weather station subsystem

http://software-engineering-book.com/web/ws-use-cases/

an interaction with the system. Each possible interaction is named in an ellipse, and the external entity involved in the interaction is represented by a stick figure.

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands. The stick figure is used in the UML to represent other systems as well as human users.

Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on. As I explain in Chapter 21, embedded systems are often modeled by describing

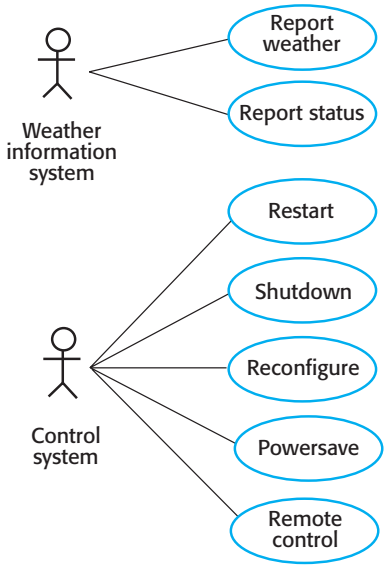


Figure 7.2 Weather station use cases

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Figure 7.3 Use case description—Report weather

how they respond to internal or external stimuli. Therefore, the stimuli and associated responses should be listed in the description. Figure 7.3 shows the description of the Report weather use case from Figure 7.2 that is based on this approach.

7.1.2 Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this knowledge with your general knowledge of the principles of architectural design and with more detailed domain knowledge. You identify the major components that make up the system and their interactions. You may then design the system organization using an architectural pattern such as a layered or client–server model.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate

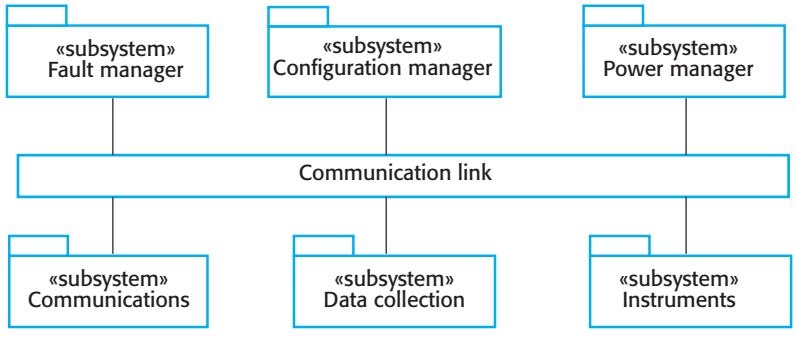


Figure 7.4 High-level architecture of weather station

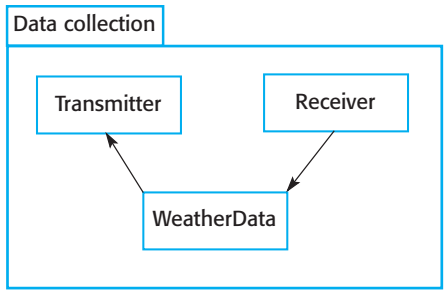


Figure 7.5 Architecture of data collection system

by broadcasting messages on a common infrastructure, shown as Communication link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.

When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Figure 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The Transmitter and Receiver objects are concerned with managing communications, and the WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows the producer–consumer pattern, discussed in Chapter 21.

7.1.3 Object class identification

By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. From the description of the Report weather use case, it is obvious that you will need to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the general object classes in the system.

As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott 1983).
2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations

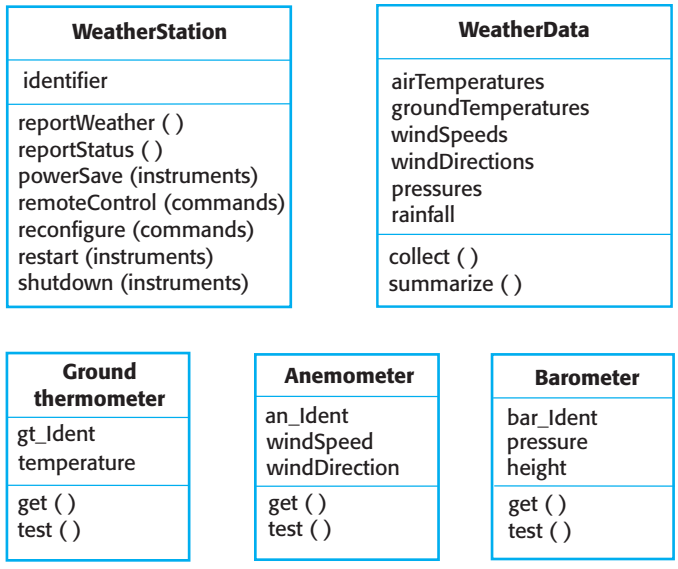


Figure 7.6 Weather station objects

such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 1990).

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham 1989).

In practice, you have to use several knowledge sources to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or analyses of existing systems. As well as the objects representing entities external to the system, you may also have to design "implementation objects" that are used to provide general services such as searching and validity checking.

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations are based on the interactions shown in Figure 7.3. I use a single object class, and it includes all of these interactions. Alternatively, you could design the system interface as several different classes, with one class per interaction.

2. The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.
3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

You use knowledge of the application domain to identify other objects, attributes. and services:

1. Weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments.
2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.
3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these objects might be implemented. Once you have identified the objects, you then refine the object design. You look for common features and then design the inheritance hierarchy for the system. For example, you may identify an Instrument superclass, which defines the common features of all instruments, such as an identifier, and get and test operations. You may also add new attributes and operations to the superclass, such as an attribute that records how often data should be collected.

7.1.4 Design models

Design or system models, as I discussed in Chapter 5, show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

The level of detail that you need in a design model depends on the design process used. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.

However, if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models. The UML supports 13 different types of models, but, as I discussed in Chapter 5, many of these models are not widely used. Minimizing the number of models that are produced reduces the costs of the design and the time required to complete the design process.

When you use the UML to develop a design, you should develop two kinds of design model:

1. *Structural models,* which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. *Dynamic models,* which describe the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.

I think three UML model types are particularly useful for adding detail to use case and architectural models:

1. *Subsystem models,* which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.
2. *Sequence models,* which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. *State machine models,* which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

A subsystem model is a useful static model that shows how a design is organized into logically related groups of objects. I have already shown this type of model in Figure 7.4 to present the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger

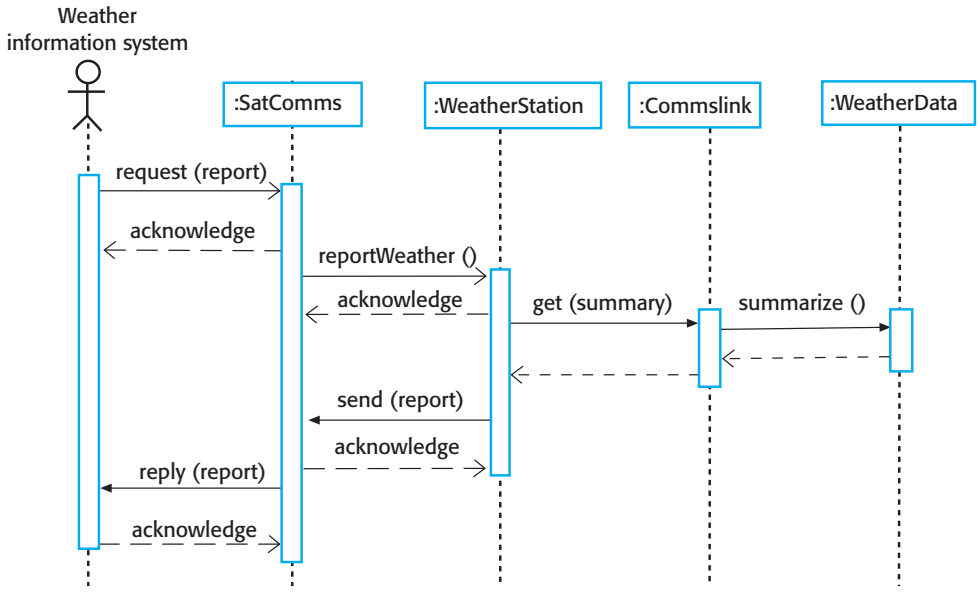


Figure 7.7 Sequence diagram describing data collection

in doing too much modeling. You should not make detailed decisions about the implementation that are really best left until the system is implemented.

Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model, then there should be a sequence model for each use case that you have identified.

Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. Commslink calls the summarize method in the object WeatherData and waits for a reply.

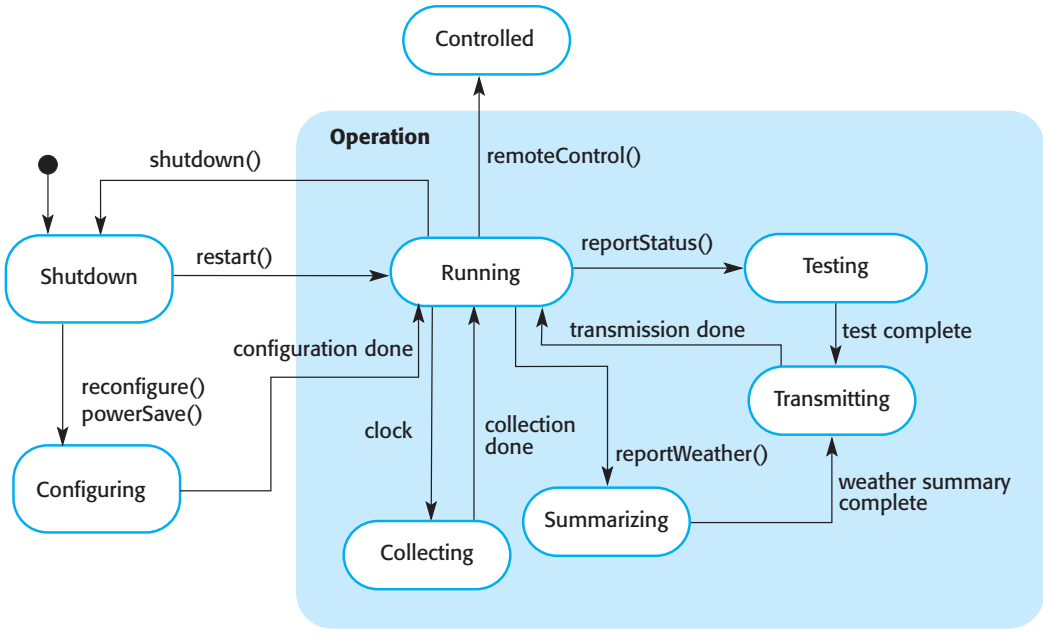


Figure 7.8 Weather station state diagram

5. The weather data summary is computed and returned to **WeatherStation** via the **Commslink** object.

6. **WeatherStation** then calls the **SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

The **SatComms** and **WeatherStation** objects may be implemented as concurrent processes, whose execution can be suspended and resumed. The **SatComms** object instance listens for messages from the external system, decodes these messages, and initiates weather station operations.

Sequence diagrams are used to model the combined behavior of a group of objects, but you may also want to summarize the behavior of an object or a subsystem in response to messages and events. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. As I discuss in Chapter 5, the UML includes state diagrams to describe state machine models.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the system state is **Shutdown**, then it can respond to a **restart**(), a **reconfigure**() or a **powerSave**() message. The unlabeled arrow with the black blob indicates that the **Shutdown** state is the initial state. A **restart**() message causes a transition to normal operation. Both the **powerSave**() and **reconfigure**() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is allowed only if the system has been shut down.

2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.
4. If a signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
5. If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation. However, you don't need a state diagram for all of the objects in the system. Many system objects in a system are simple, and their operation can be easily described without a state model.

7.1.5 Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL). I discuss the use of the OCL in Chapter 16, where I explain how it can be used to describe the semantics of components.

You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data. As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, you should normally expose the attributes in an object model, as this is the clearest way of describing the essential characteristics of the objects.

There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects "implement" interfaces. Equally, a group of objects may all be accessed through a single interface.

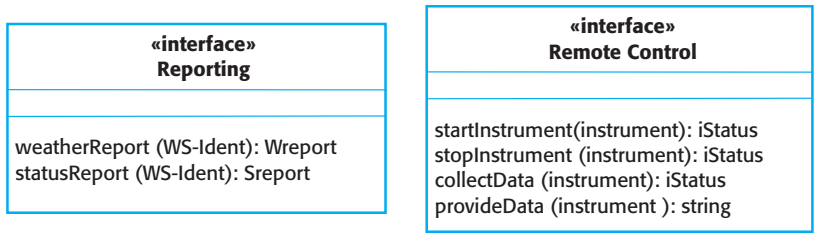


Figure 7.9 Weather station interfaces

Figure 7.9 shows two interfaces that may be defined for the weather station. The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remoteControl method, shown in Figure 7.6.

7.2 Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander 1979), who suggested that there were certain common patterns of building design that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group website (hillside.net/patterns/), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

> *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience*[†].

Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best known design patterns that were originally described by the "Gang of Four" in their patterns book, published in 1995 (Gamma et al. 1995). Other important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b).

Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is

[†]The HIllside Group: hillside.net/patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.

The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 7.10 The Observer pattern

one that is equally applicable to any kind of software design. For instance, you could have configuration patterns for instantiating reusable application systems.

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

To illustrate pattern description, I use the Observer pattern, taken from the Gang of Four's patterns book. This is shown in Figure 7.10. In my description, I use the

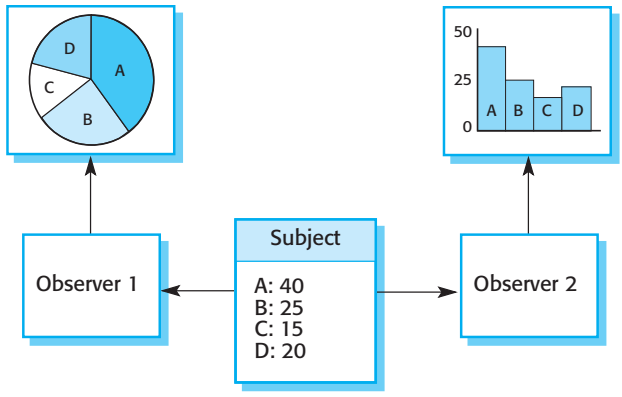


Figure 7.11 Multiple displays

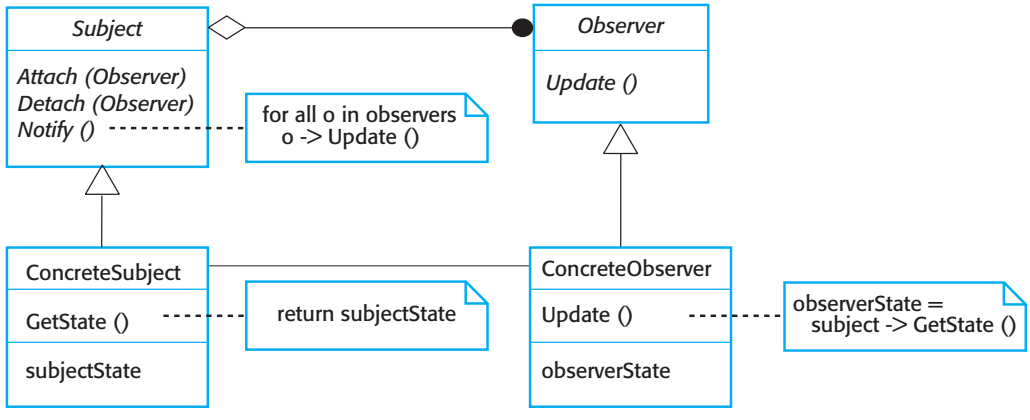
four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two different graphical presentations of the same dataset.

Graphical representations are normally used to illustrate the object classes in patterns and their relationships. These supplement the pattern description and add detail to the solution description. Figure 7.12 is the representation in UML of the Observer pattern.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).
2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

Figure 7.12 A UML model of the Observer pattern



3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
4. Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

Patterns support high-level, concept reuse. When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. When these design decisions conflict with your requirements, reusing the component is either impossible or introduces inefficiencies into your system. Using patterns means that you reuse the ideas but can adapt the implementation to suit the system you are developing.

When you start designing a system, it can be difficult to know, in advance, if you will need a particular pattern. Therefore, using patterns in a design process often involves developing a design, experiencing a problem, and then recognizing that a pattern can be used. This is certainly possible if you focus on the 23 general-purpose patterns documented in the original patterns book. However, if your problem is a different one, you may find it difficult to find an appropriate pattern among the hundreds of different patterns that have been proposed.

Patterns are a great idea, but you need experience of software design to use them effectively. You have to recognize situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

7.3 Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

I assume that most readers of this book will understand programming principles and will have some programming experience. As this chapter is intended to offer a language-independent approach, I haven't focused on issues of good programming practice as language-specific examples need to be used. Instead, I introduce some aspects of implementation that are particularly important to software engineering and that are often not covered in programming texts. These are:

1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

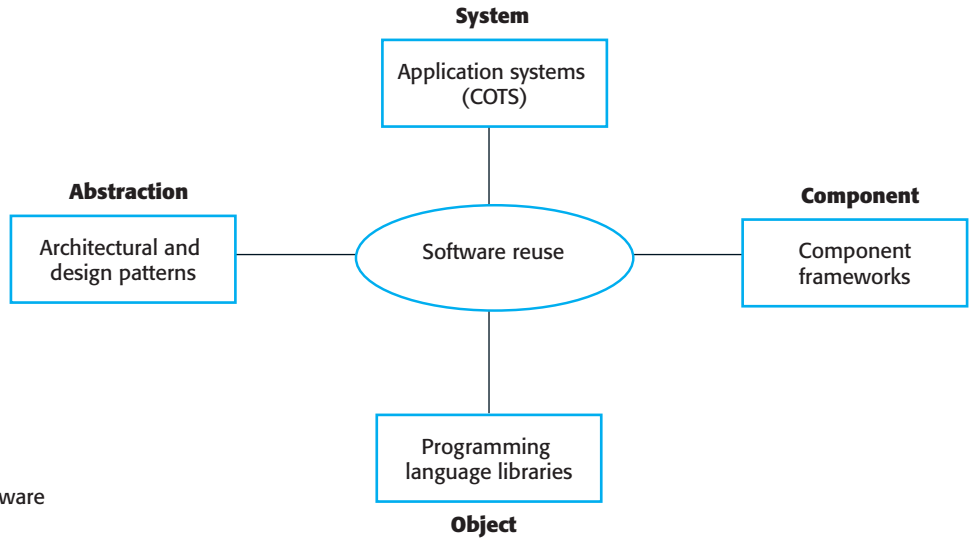


Figure 7.13 Software reuse

2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.

3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

7.3.1 Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels, as shown in Figure 7.13:

1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns (covered in Chapter 6) are ways of representing abstract knowledge for reuse.

2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process email messages in a Java program, you may use objects and methods from a JavaMail library.
3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.
4. *The system level* At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.
2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the

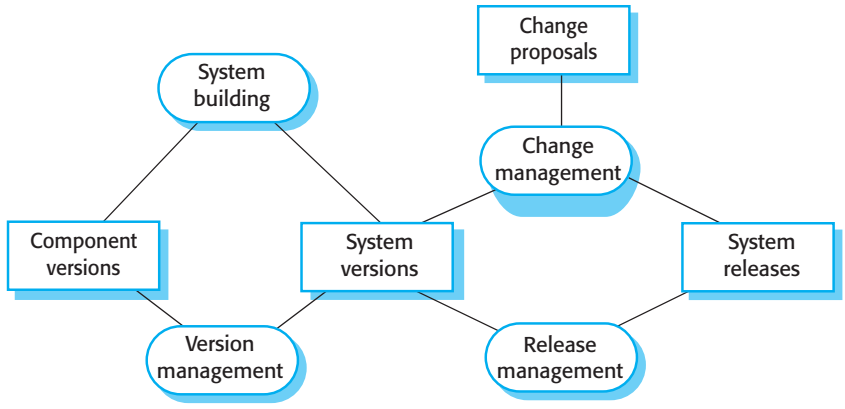


Figure 7.14 Configuration management

possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets. As I discussed in Chapter 2, in a reuse-oriented development process, you search for reusable elements, then modify your requirements and design to make the best use of these.

Because of the importance of reuse in modern software engineering, I devote several chapters in Part 3 of this book to this topic (Chapters 15, 16, and 18).

7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. As shown in Figure 7.14, there are four fundamental configuration management activities:

1. *Version management,* where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.
2. *System integration,* where support is provided to help developers define what versions of components are used to create each version of a system. This

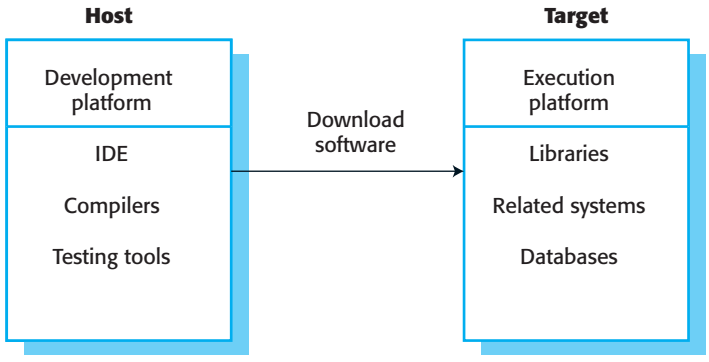


Figure 7.15 Host-target development

description is then used to build a system automatically by compiling and linking the required components.

3. *Problem tracking,* where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse. Version management may be supported using a version management system such as Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) or Git (Loeliger and McCullough 2012), which can support multi-site, multi-team development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed. A comprehensive set of tools built around the Git system is available at Github (http://github.com).

Because of its importance in professional software engineering, I discuss change and configuration management in more detail in Chapter 25.

7.3.3 Host-target development

Most professional software development is based on a host-target model (Figure 7.15). Software is developed on one computer (the host) but runs on a separate machine (the target). More generally, we can talk about a development platform (host) and an execution platform (target). A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures.

If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software. It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system.

A software development platform should provide a range of tools to support software engineering processes. These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 12). Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are now usually installed within an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface. Generally, IDEs are created to support development in a specific programming



UML deployment diagrams

UML deployment diagrams show how software components are physically deployed on processors. That is, the deployment diagram shows the hardware and software in the system and the middleware used to connect the different components in the system. Essentially, you can think of deployment diagrams as a way of defining and documenting the target environment.

http://software-engineering-book.com/web/deployment/

language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org). This environment is based on a plug-in architecture so that it can be specialized for different languages, such as Java, and application domains. Therefore, you can install Eclipse and tailor it for your specific needs by adding plug-ins. For example, you may add a set of plug-ins to support networked systems development in Java (Vogel 2013) or embedded systems engineering using C.

As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. *Component communications* If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators and its real-time operating system. I discuss embedded systems engineering in Chapter 21.

7.4 Open-source development

Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process (Raymond 2001). Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.

Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

It is usually cheap or even free to acquire open-source software. You can normally download open-source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low. The other key benefit of using open-source products is that widely used open-source systems are very reliable. They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system. Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

For a company involved in software development, there are two open-source issues that have to be considered:

1. Should the product that is being developed make use of open-source components?
2. Should an open-source approach be used for its own software development?

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available

open-source systems. Even then, however, it could be quicker and cheaper to modify the open-source system rather than redevelop the functionality that you need.

Many software product companies are now using an open-source approach to development, especially for specialized systems. Their business model is not reliant on selling a software product but rather on selling support for that product. They believe that involving the open-source community will allow software to be developed more cheaply and more quickly and will create a community of users for the software.

Some companies believe that adopting an open-source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open-source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. Making a software system open source does not guarantee community involvement. There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

7.4.1 Open-source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license (St. Laurent 2004). Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source. Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed-source systems.

Most open-source licenses (Chapman 2010) are variants of one of three general models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to

open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open-source software in its development.

If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (Bayersdorfer 2007) suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

The open-source approach is one of several business models for software. In this model, companies release the source of their software and sell add-on services and advice in association with this. They may also sell cloud-based software services—an attractive option for users who do not have the expertise to manage their own open-source system and also specialized versions of their system for particular clients. Open-source is therefore likely to increase in importance as a way of developing and distributing software.

KEY POINTS

- Software design and implementation are interleaved activities. The level of detail in the design depends on the type of system being developed and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models, and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.
- When developing software, you should always consider the possibility of reusing existing software, either as components, services, or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people is cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open-source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

FURTHER READING

Design Patterns: Elements of Reusable Object-oriented Software. This is the original software patterns handbook that introduced software patterns to a wide community. (E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley, 1995).

Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd ed. Larman writes clearly on object-oriented design and also discusses use of the UML; this is a good introduction to using patterns in the design process. Although it is more than 10 years old, it remains the best book on this topic that is available. (C. Larman, Prentice-Hall, 2004).

Producing Open Source Software: How to Run a Successful Free Software Project. This book is a comprehensive guide to the background to open-source software, licensing issues, and the practicalities of running an open-source development project. (K. Fogel, O'Reilly Media Inc., 2008).

Further reading on software reuse is suggested in Chapter 15 and on configuration management in Chapter 25.

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

More information on the weather information system:

http://software-engineering-book.com/case-studies/wilderness-weather-station/

EXERCISES

7.1. Using the tabular notation shown in Figure 7.3, specify the weather station use cases for Report status and Reconfigure. You should make reasonable assumptions about the functionality that is required here.

7.2. Assume that the Mentcare system is being developed using an object-oriented approach. Draw a use case diagram showing at least six possible use cases for this system.

7.3. Using the UML graphical notation for object classes, design the following object classes, identifying attributes and operations. Use your own experience to decide on the attributes and operations that should be associated with these objects.

- a messaging system on a mobile (cell) phone or tablet
- a printer for a personal computer
- a personal music system
- a bank account
- a library catalogue

7.4. A shape can be classified into 2-D and 3-D. Design an inheritance hierarchy that will include different kinds of 2-D and 3-D shapes. Make sure you identify at least five other classes of shapes.

7.5. Develop the design of the weather station to show the interaction between the data collection subsystem and the instruments that collect weather data. Use sequence diagrams to show this interaction.

7.6. Identify possible objects in the following systems and develop an object-oriented design for them. You may make any reasonable assumptions about the systems when deriving the design.

- A group diary and time management system is intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made

that involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange his or her personal diary to make room for the appointment.

- A filling station (gas station) is to be set up for fully automated operation. Drivers swipe their credit card through a reader connected to the pump; the card is verified by communication with a credit company computer, and a fuel limit is established. The driver may then take the fuel required. When fuel delivery is complete and the pump hose is returned to its holster, the driver's credit card account is debited with the cost of the fuel taken. The credit card is returned after debiting. If the card is invalid, the pump returns it before fuel is dispensed.

7.7. Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people are arranging a meeting.

7.8. Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system.

7.9. When code is integrated into a larger system, problems may surface. Explain how configuration management can be useful when handling such problems.

7.10. A small company has developed a specialized software product that it configures specially for each customer. New customers usually have specific requirements to be incorporated into their system, and they pay for these to be developed and integrated with the product. The software company has an opportunity to bid for a new contract, which would more than double its customer base. The new customer wishes to have some involvement in the configuration of the system. Explain why, in these circumstances, it might be a good idea for the company owning the software to make it open source.

REFERENCES

Abbott, R. 1983. "Program Design by Informal English Descriptions." *Comm. ACM* 26 (11): 882–894. doi:10.1145/182.358441.

Alexander, C. 1979. *A Timeless Way of Building*. Oxford, UK: Oxford University Press.

Bayersdorfer, M. 2007. "Managing a Project with Open Source Components." *ACM Interactions* 14 (6): 33–34. doi:10.1145/1300655.1300677.

Beck, K., and W. Cunningham. 1989. "A Laboratory for Teaching Object-Oriented Thinking." In *Proc. OOPSLA' 89 (Conference on Object-Oriented Programming, Systems, Languages and Applications)*, 1–6. ACM Press. doi:10.1145/74878.74879.

Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.

———. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.

Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* New York: John Wiley & Sons.

Chapman, C. 2010. "A Short Guide to Open-Source and Similar Licences." *Smashing Magazine*. http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses./

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA.: Addison-Wesley*.

Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly & Associates.

Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly & Associates.

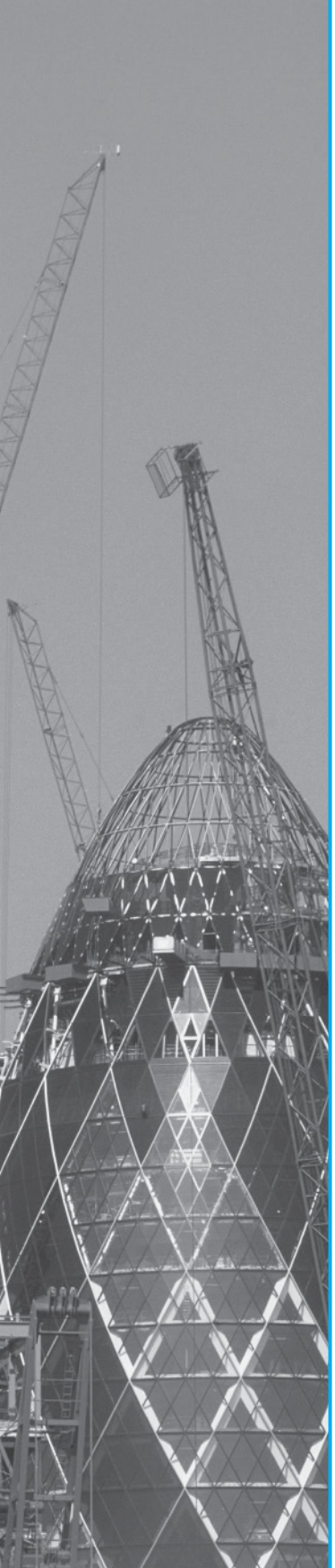
Raymond, E. S. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol. CA: O'Reilly & Associates.

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

St. Laurent, A. 2004. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly & Associates.

Vogel, L. 2013. *Eclipse IDE: A Tutoria*l. Hamburg, Germany: Vogella Gmbh.

Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall.



8 Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know about three distinct types of testing—component testing, system testing, and release testing;
- understand the distinctions between development testing and user testing.

Contents

8.1 Development testing
8.2 Test-driven development
8.3 Release testing
8.4 User testing

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

When you test software, you are trying to do two things:

1. Demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features that will be included in the product release. You may also test combinations of features to check for unwanted interactions between them.

2. Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are caused by defects (bugs) in the software. When you test software to find defects, you are trying to root out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first of these is validation testing, where you expect the system to perform correctly using a set of test cases that reflect the system's expected use. The second is defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used. Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

Figure 8.1 shows the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e. The priority in defect testing is to find those inputs in the set I_e because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside I_e. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test you have overlooked could discover further problems with the system. As Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra 1972):

"Testing can only show the presence of errors, not their absence[†]*"*

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused. Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm 1979):

[†]Dijkstra, E. W. 1972. "The Humble Programmer." Comm. ACM 15 (10): 859–66. doi:10.1145/355604.361591

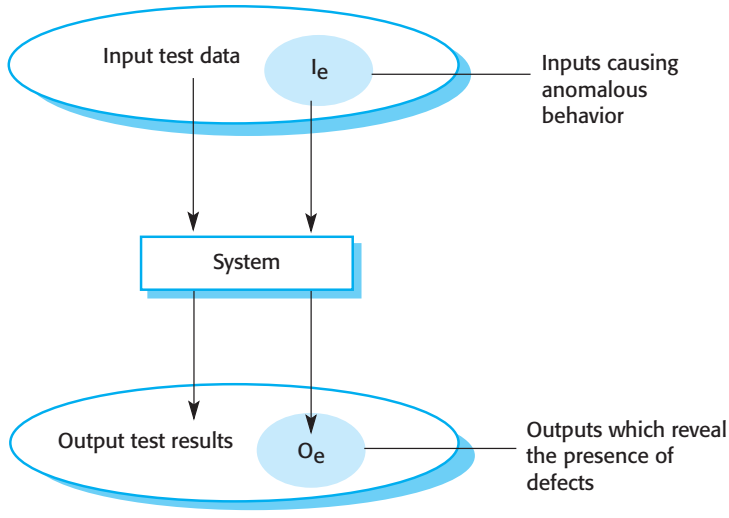


Figure 8.1 An input–output model of program testing

- **Validation:** *Are we building the right product?*
- **Verification:** *Are we building the product right?*

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

Software verification is the process of checking that the software meets its stated functional and non-functional requirements. Validation is a more general process. The aim of software validation is to ensure that the software meets the customer's expectations. It goes beyond checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, statements of requirements do not always reflect the real wishes or needs of system customers and users.

The goal of verification and validation processes is to establish confidence that the software system is "fit for purpose." This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

1. *Software purpose* The more critical the software, the more important it is that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a demonstrator system that prototypes new product ideas.
2. *User expectations* Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users

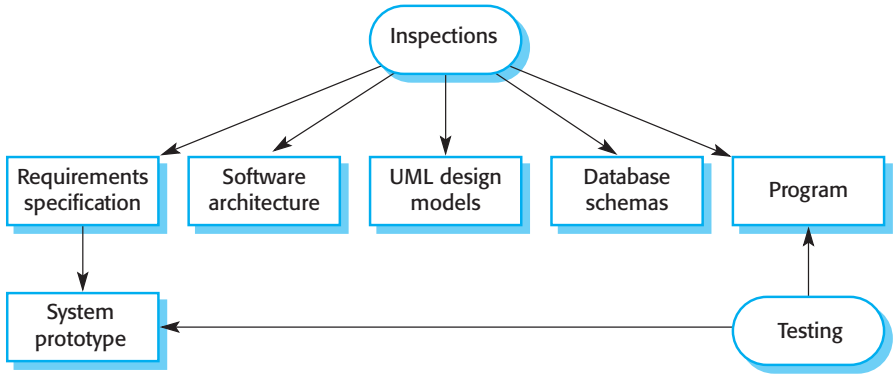


Figure 8.2 Inspections and testing

may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable. Consequently, more thorough testing of later versions of the system may be required.

3. *Marketing environment* When a software company brings a system to market, it must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, the company may decide to release a program before it has been fully tested and debugged because it wants to be the first into the market. If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability.

As well as software testing, the verification and validation process may involve software inspections and reviews. Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. These are "static" V & V techniques in which you don't need to execute the software to verify it. Figure 8.2 shows that software inspections and testing support V & V at different stages in the software process. The arrows indicate the stages in the process where the techniques may be used.

Inspections mostly focus on the source code of a system, but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

Software inspection has three advantages over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection doesn't involve executing the system, you don't have to worry about interactions between errors. Consequently, a single inspection session can discover many errors in a system.

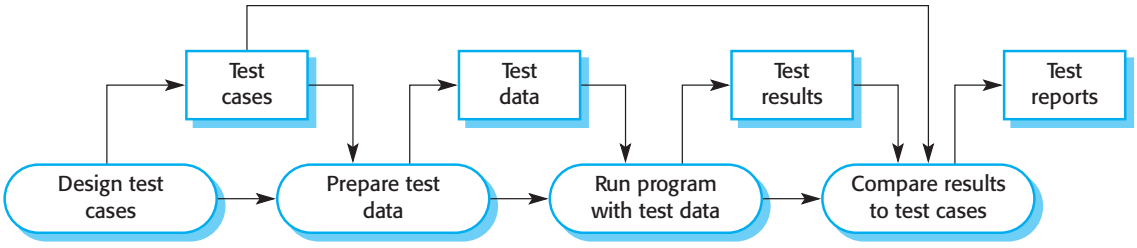


Figure 8.3 A model of the software testing process

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Program inspections are an old idea, and several studies and experiments have shown that inspections are more effective for defect discovery than program testing. Fagan (Fagan 1976) reported that more than 60% of the errors in a program can be detected using informal program inspections. In the Cleanroom process (Prowell et al. 1999), it is claimed that more than 90% of defects can be discovered in program inspections.

However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance. In small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential team members may also be developers of the software.

I discuss reviews and inspections in more detail in Chapter 24 (Quality Management). Static analysis, where the source text of a program is automatically analyzed to discover anomalies, is explained in Chapter 12. In this chapter, I focus on testing and testing processes.

Figure 8.3 is an abstract model of the traditional testing process, as used in plan-driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible. People who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The test results are automatically compared with the predicted results, so there is no need for a person to look for errors and anomalies in the test run.



Test planning

Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available. Usually, a test plan will be created that defines what is to be tested, the predicted testing schedule, and how tests will be recorded. For critical systems, the test plan may also include details of the tests to be run on the software.

http://software-engineering-book.com/web/test-planning/

Typically, a commercial software system has to go through three stages of testing:

1. *Development testing*, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
2. *Release testing*, where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of the system stakeholders.
3. *User testing*, where users or potential users of a system test the system in their own environment. For software products, the "user" may be an internal marketing group that decides if the software can be marketed, released and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process usually involves a mixture of manual and automated testing. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers. In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. This is faster than manual testing, especially when it involves regression testing—re-running previous tests to check that changes to the program have not introduced new bugs.

Unfortunately, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do. It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unanticipated side effects.

8.1 Development testing

Development testing includes all testing activities that are carried out by the team developing the system. The tester of the software is usually the programmer who developed that software. Some development processes use programmer/tester pairs (Cusamano and Selby 1998) where each programmer has an associated tester who



Debugging

Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. When you are debugging a program, you usually use interactive tools that provide extra information about program execution.

http://software-engineering-book.com/web/debugging/

develops tests and assists with the testing process. For critical systems, a more formal process may be used, with a separate testing group within the development team. This group is responsible for developing tests and maintaining detailed records of test results.

There are three stages of development testing:

1. *Unit testing*, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. *Component testing*, where several individual units are integrated to create composite components. Component testing should focus on testing the component interfaces that provide access to the component functions.
3. *System testing*, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where the aim of testing is to discover bugs in the software. It is therefore usually interleaved with debugging—the process of locating problems with the code and changing the program to fix these problems.

8.1.1 Unit testing

Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters. You can use the approaches to test-case design discussed in Section 8.1.2 to design the function or method tests.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should test all operations associated with the object; set and check the value of all attributes associated with the object; and put the object into all possible states. This means that you should simulate all events that cause a state change.

Consider, for example, the weather station object from the example that I discussed in Chapter 7. The attributes and operations of this object are shown in Figure 8.4.

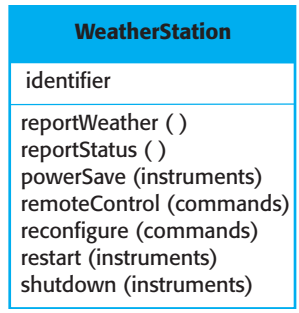


Figure 8.4 The weather station object interface

It has a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks if it has been properly set up. You need to define test cases for all of the methods associated with the object such as **reportWeather** and **reportStatus**. Ideally, you should test methods in isolation, but, in some cases, test sequences are necessary. For example, to test the method that shuts down the weather station instruments (**shutdown**), you need to have executed the **restart** method.

Generalization or inheritance makes object class testing more complicated. You can't simply test an operation in the class where it is defined and assume that it will work as expected in all of the subclasses that inherit the operation. The operation that is inherited may make assumptions about other operations and attributes. These assumptions may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation everywhere that it is used.

To test the states of the weather station, you can use a state model as discussed in Chapter 7 (Figure 7.8). Using this model, you identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, you should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Whenever possible, you should automate unit testing. In automated unit testing, you make use of a test automation framework, such as JUnit (Tahchiev et al. 2010) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some graphical unit interface (GUI), on the success or otherwise of the tests. An entire test suite can often be run in a few seconds, so it is possible to execute all tests every time you make a change to the program.

An automated test has three parts:

1. *A setup part*, where you initialize the system with the test case, namely, the inputs and expected outputs.

2. *A call part,* where you call the object or method to be tested.
3. *An assertion part*, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Sometimes, the object that you are testing has dependencies on other objects that may not have been implemented or whose use slows down the testing process. For example, if an object calls a database, this may involve a slow setup process before it can be used. In such cases, you may decide to use mock objects.

Mock objects are objects with the same interface as the external objects being used that simulate its functionality. For example, a mock object simulating a database may have only a few data items that are organized in an array. They can be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operations or rare events. For example, if your system is intended to take action at certain times of day, your mock object can simply return those times, irrespective of the actual clock time.

8.1.2 Choosing unit test cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

You should therefore design two kinds of test case. The first of these should reflect normal operation of a program and should show that the component works. For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Two strategies that can be effective in helping you choose test cases are:

1. *Partition testing*, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
2. *Guideline-based testing*, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

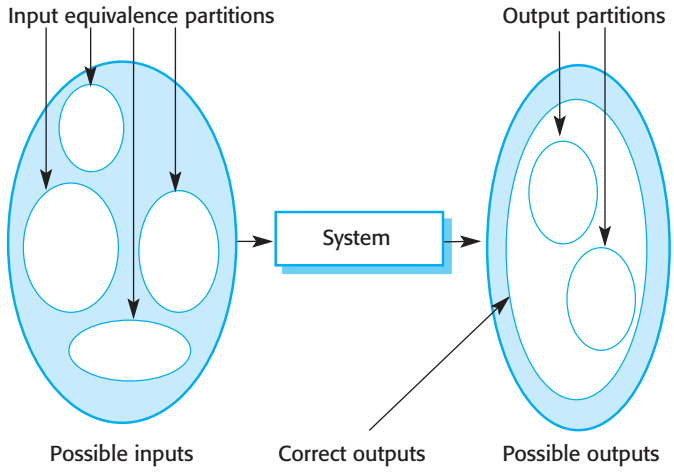


Figure 8.5 Equivalence partitioning

The input data and output results of a program can be thought of as members of sets with common characteristics. Examples of these sets are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a set. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier 1990). One systematic approach to test-case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partition in the same way.

Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur, that is, responses to invalid inputs.

Once you have identified a set of partitions, you choose test cases from each of these partitions. A good rule of thumb for test-case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition. Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) and so are sometimes overlooked by developers. Program failures often occur when processing these atypical values.

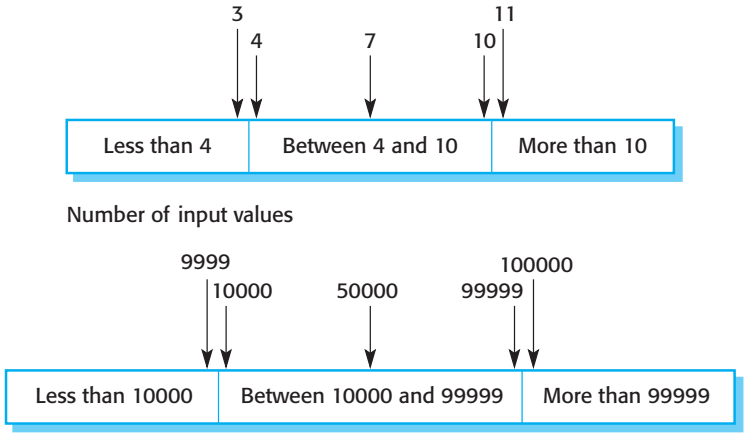


Figure 8.6 Equivalence partitions

You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors. For example, say a program specification states that the program accepts four to eight inputs which are five-digit integers greater than 10,000. You use this information to identify the input partitions and possible test input values. These are shown in Figure 8.6.

When you use the specification of a system to identify equivalence partitions, this is called black-box testing. You don't need any knowledge of how the system works. It is sometimes useful to supplement the black-box tests with "white-box testing," where you look at the code of the program to find other possible tests. For example, your code may include exceptions to handle incorrect inputs. You can use this knowledge to identify "exception partitions"—different ranges where the same exception handling should be applied.

Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions. You can also use testing guidelines to help choose test cases. Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors. For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, if presented with a single-value sequence, a program may not work properly.
2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. Derive tests so that the first, middle, and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.



Path testing

Path testing is a testing strategy that aims to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. All conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used to test the methods associated with objects.

http://software-engineering-book.com/web/path-testing/

Whittaker's book (Whittaker 2009) includes many examples of guidelines that can be used in test-case design. Some of the most general guidelines that he suggests are:

- Choose inputs that force the system to generate all error messages:
- Design inputs that cause input buffers to overflow.
- Repeat the same input or series of inputs numerous times.
- Force invalid outputs to be generated.
- Force computation results to be too large or too small.

As you gain experience with testing, you can develop your own guidelines about how to choose effective test cases. I give more examples of testing guidelines in the next section.

8.1.3 Component testing

Software components are often made up of several interacting objects. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration. You access the functionality of these objects through component interfaces (see Chapter 7). Testing composite components should therefore focus on showing that the component interface or interfaces behave according to its specification. You can assume that unit tests on the individual objects within the component have been completed.

Figure 8.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem. The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components. Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

1. *Parameter interfaces* These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

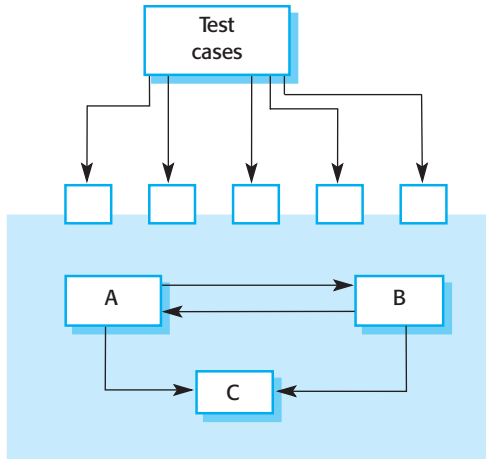


Figure 8.7 Interface testing

2. *Shared memory interfaces* These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other subsystems. This type of interface is used in embedded systems, where sensors create data that is retrieved and processed by other system components.

3. *Procedural interfaces* These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

4. *Message passing interfaces* These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Interface errors are one of the most common forms of error in complex systems (Lutz 1993). These errors fall into three classes:

- *Interface misuse* A calling component calls some other component and makes an error in the use of its interface. This type of error is common in parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.
- *Interface misunderstanding* A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected, which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.
- *Timing errors* These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may

operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure, and so it does not check for queue overflow when an item is entered.

This condition can only be detected during testing by designing a sequence of test cases that force the queue to overflow. The tests should check how calling objects handle that overflow. However, as this is a rare condition, testers may think that this isn't worth checking when writing the test set for the queue object.

A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. Say an object calls another object to receive some service and the calling object assumes that the response is correct. If the called service is faulty in some way, the returned value may be valid but incorrect. The problem is therefore not immediately detectable but only becomes obvious when some later computation, using the returned value, goes wrong.

Some general guidelines for interface testing are:

1. Examine the code to be tested and identify each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.
3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
4. Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.
5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Sometimes it is better to use inspections and reviews rather than testing to look for interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

8.1.4 System testing

System testing during development involves integrating components to create a version of the system and then testing the integrated system. System testing checks that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces. It obviously overlaps with component testing, but there are two important differences:

1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
2. Components developed by different team members or subteams may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

All systems have emergent behavior. This means that some system functionality and characteristics only become obvious when you put the components together. This may be planned emergent behavior, which has to be tested. For example, you may integrate an authentication component with a component that updates the system database. You then have a system feature that restricts information updating to authorized users. Sometimes, however, the emergent behavior is unplanned and unwanted. You have to develop tests that check that the system is only doing what it is supposed to do.

System testing should focus on testing the interactions between the components and objects that make up a system. You may also test reusable components or systems to check that they work as expected when they are integrated with new components. This interaction testing should discover those component bugs that are only revealed when a component is used by other components in the system. Interaction testing also helps find misunderstandings, made by component developers, about other components in the system.

Because of its focus on interactions, use case-based testing is an effective approach to system testing. Several components or objects normally implement each use case in the system. Testing the use case forces these interactions to occur. If you have developed a sequence diagram to model the use case implementation, you can see the objects or components that are involved in the interaction.

In the wilderness weather station example, the system software reports summarized weather data to a remote computeras described in Figure 7.3. Figure 8.8 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests. Therefore issuing a request for a report will result in the execution of the following thread of methods:

SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize

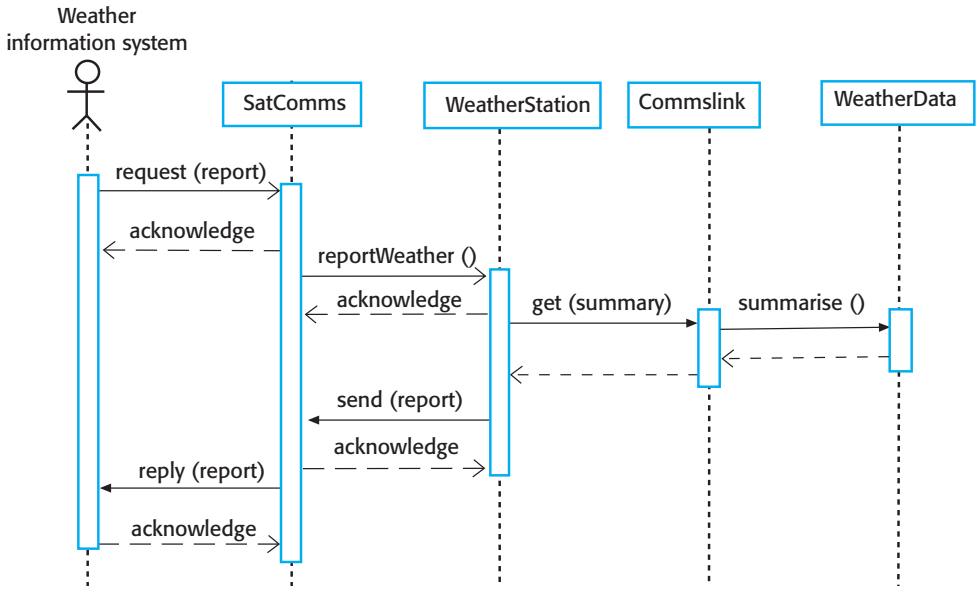


Figure 8.8 Collect weather data sequence chart

The sequence diagram helps you design the specific test cases that you need, as it shows what inputs are required and what outputs are created:

1. An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request. During testing, you should create summarized data that can be used to check that the report is correctly organized.

2. An input request for a report to **WeatherStation** results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of **SatComms** and checking that the **WeatherStation** object correctly produces this summary. This raw data is also used to test the **WeatherData** object.

Of course, I have simplified the sequence diagram in Figure 8.8 so that it does not show exceptions. A complete use case/scenario test must take these exceptions into account and ensure that they are correctly handled.

For most systems, it is difficult to know how much system testing is essential and when you should stop testing. Exhaustive testing, where every possible program execution sequence is tested, is impossible. Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset. These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once. Alternatively, they may be based on experience of system usage and focus on testing the features of the operational system. For example:



Incremental integration and testing

System testing involves integrating different components, then testing the integrated system that you have created. You should always use an incremental approach to integration and testing where you integrate a component, test the system, integrate another component, test again, and so on. If problems occur, they are probably due to interactions with the most recently integrated component.

Incremental integration and testing is fundamental to agile methods, where regression tests are run every time a new increment is integrated.

http://software-engineering-book.com/web/integration/

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

It is clear from experience with major software products such as word processors or spreadsheets that similar guidelines are normally used during product testing. When features of the software are used in isolation, they normally work. Problems arise, as Whittaker explains (Whittaker 2009), when combinations of less commonly used features have not been tested together. He gives the example of how, in a commonly used word processor, using footnotes with multicolumn layout causes incorrect layout of the text.

Automated system testing is usually more difficult than automated unit or component testing. Automated unit testing relies on predicting the outputs and then encoding these predictions in a program. The prediction is then compared with the result. However, the point of implementing a system may be to generate outputs that are large or cannot be easily predicted. You may be able to examine an output and check its credibility without necessarily being able to create it in advance.

8.2 Test-driven development

Test-driven development (TDD) is an approach to program development in which you interleave testing and code development (Beck 2002; Jeffries and Melnik 2007). You develop the code incrementally, along with a set of tests for that increment. You don't start working on the next increment until the code that you have developed passes all of its tests. Test-driven development was introduced as part of the XP agile development method. However, it has now gained mainstream acceptance and may be used in both agile and plan-based processes.

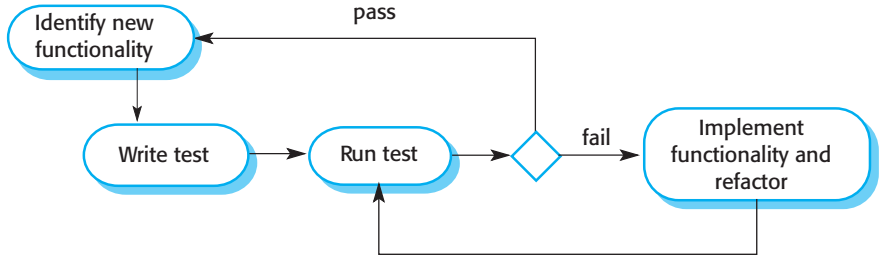


Figure 8.9 Test-driven development

The fundamental TDD process is shown in Figure 8.9. The steps in the process are as follows:

1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. You write a test for this functionality and implement it as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

An automated testing environment, such as the JUnit environment that supports Java program testing (Tahchiev et al. 2010) is essential for TDD. As the code is developed in very small increments, you have to be able to run every test each time that you add functionality or refactor the program. Therefore, the tests are embedded in a separate program that runs the tests and invokes the system that is being tested. Using this approach, you can run hundreds of separate tests in a few seconds.

Test-driven development helps programmers clarify their ideas of what a code segment is actually supposed to do. To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code. Of course, if you have incomplete knowledge or understanding, then TDD won't help.

If you don't know enough to write the tests, you won't develop the required code. For example, if your computation involves division, you should check that you are not dividing the numbers by zero. If you forget to write a test for this, then the checking code will never be included in the program.

As well as better problem understanding, other benefits of test-driven development are:

1. *Code coverage* In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in

the system has actually been executed. Code is tested as it is written, so defects are discovered early in the development process.

2. *Regression testing* A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.

3. *Simplified debugging* When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of TDD suggest that it is hardly ever necessary to use an automated debugger in test-driven development (Martin 2007).

4. *System documentation* The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

One of the most important benefits of TDD is that it reduces the costs of regression testing. Regression testing involves running test sets that have successfully executed after changes have been made to a system. The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code. Regression testing is expensive and sometimes impractical when a system is manually tested, as the costs in time and effort are very high. You have to try to choose the most relevant tests to re-run and it is easy to miss important tests.

Automated testing dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply. After making a change to a system in test-first development, all existing tests must run successfully before any further functionality is added. As a programmer, you can be confident that the new functionality that you have added has not caused or revealed problems with existing code.

Test-driven development is of most value in new software development where the functionality is either implemented in new code or by using components from standard libraries. If you are reusing large code components or legacy systems, then you need to write tests for these systems as a whole. You cannot easily decompose them into separate testable elements. Incremental test-driven development is impractical. Test-driven development may also be ineffective with multithreaded systems. The different threads may be interleaved at different times in different test runs, and so may produce different results.

If you use TDD, you still need a system testing process to validate the system, that is, to check that it meets the requirements of all of the system stakeholders. System testing also tests performance, reliability, and checks that the system does not do things that it shouldn't do, such as produce unwanted outputs. Andrea (Andrea 2007) suggests how testing tools can be extended to integrate some aspects of system testing with TDD.

Test-driven development is now a widely used and mainstream approach to software testing. Most programmers who have adopted this approach are happy with it

and find it a more productive way to develop software. It is also claimed that use of TDD encourages better structuring of a program and improved code quality. However, experiments to verify this claim have been inconclusive.

8.3 Release testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. Normally, the system release is for customers and users. In a complex project, however, the release could be for other teams that are developing related systems. For software products, the release could be for product management who then prepare it for sale.

There are two important distinctions between release testing and system testing during the development process:

1. The system development, team should not be responsible for release testing.
2. Release testing is a process of validation checking to ensure that a system meets its requirements and is good enough for use by system customers. System testing by the development team should focus on discovering bugs in the system (defect testing).

The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use. If so, it can be released as a product or delivered to the customer. Release testing, therefore, has to show that the system delivers its specified functionality, performance, and dependability, and that it does not fail during normal use.

Release testing is usually a black-box testing process whereby tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs. Another name for this is functional testing, so-called because the tester is only concerned with functionality and not the implementation of the software.

8.3.1 Requirements-based testing

A general principle of good requirements engineering practice is that requirements should be testable. That is, the requirement should be written so that a test can be designed for that requirement. A tester can then check that the requirement has been satisfied. Requirements-based testing, therefore, is a systematic approach to test-case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing—you are trying to demonstrate that the system has properly implemented its requirements.

For example, consider the following Mentcare system requirements that are concerned with checking for drug allergies:

> *If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.*
>
> *If a prescriber chooses to ignore an allergy warning, he or she shall provide a reason why this has been ignored.*

To check if these requirements have been satisfied, you may need to develop several related tests:

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
2. Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to and check that the warning is issued by the system.
3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

You can see from this list that testing a requirement does not mean just writing a single test. You normally have to write several tests to ensure that you have coverage of the requirement. You should also keep traceability records of your requirements-based testing, which link the tests to the specific requirements that you have tested.

8.3.2 Scenario testing

Scenario testing is an approach to release testing whereby you devise typical scenarios of use and use these scenarios to develop test cases for the system. A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic, and real system users should be able to relate to them. If you have used scenarios or user stories as part of the requirements engineering process (described in Chapter 4), then you may be able to reuse them as testing scenarios.

In a short paper on scenario testing, Kaner (Kaner 2003) suggests that a scenario test should be a narrative story that is credible and fairly complex. It should motivate stakeholders; that is, they should relate to the scenario and believe that it is

George is a nurse who specializes in mental health care. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients whom he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect, so he notes the problem in Jim's record and suggests that he visit the clinic to have his medication changed. Jim agrees, so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation, and the system re-encrypts Jim's record.

After finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients whom he has to contact for follow-up information and make clinic appointments.

Figure 8.10 A user story for the Mentcare system

important that the system passes the test. He also suggests that it should be easy to evaluate. If there are problems with the system, then the release testing team should recognize them.

As an example of a possible scenario from the Mentcare system, Figure 8.10 describes one way that the system may be used on a home visit. This scenario tests a number of features of the Mentcare system:

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information.
7. The system for call prompting.

If you are a release tester, you run through this scenario, playing the role of George and observing how the system behaves in response to different inputs. As George, you may make deliberate mistakes, such as inputting the wrong key phrase to decode records. This checks the response of the system to errors. You should carefully note any problems that arise, including performance problems. If a system is too slow, this will change the way that it is used. For example, if it takes too long to encrypt a record, then users who are short of time may skip this stage. If they then lose their laptop, an unauthorized person could then view the patient records.

When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking that combinations of requirements do not cause problems.

8.3.3 Performance testing

Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system. To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile (see Chapter 11) is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A, 5% of type B, and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach, of course, is not necessarily the best approach for defect testing. Experience has shown that an effective way to discover defects is to design tests around the limits of the system. In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as stress testing.

Say you are testing a transaction processing system that is designed to process up to 300 transactions per second. You start by testing this system with fewer than 300 transactions per second. You then gradually increase the load on the system beyond 300 transactions per second until it is well beyond the maximum design load of the system and the system fails.

Stress testing helps you do two things:

1. Test the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to "fail-soft" rather than collapse under its load.
2. Reveal defects that only show up when the system is fully loaded. Although it can be argued that these defects are unlikely to cause system failures in normal use, there may be unusual combinations of circumstances that the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different processes must exchange. The processes become slower and slower as they wait for the required data from other processes. Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

8.4 User testing

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. This may involve formally testing a system that has been commissioned from an external supplier. Alternatively, it may be an informal process where users experiment with a new software product to see if they like it and to check that it does what they need. User testing is essential, even when comprehensive system and release testing have been carried out. Influences from the user's working environment can have a major effect on the reliability, performance, usability, and robustness of a system.

It is practically impossible for a system developer to replicate the system's working environment, as tests in the developer's environment are inevitably artificial. For example, a system that is intended for use in a hospital is used in a clinical environment where other things are going on, such as patient emergencies and conversations with relatives. These all affect the use of a system, but developers cannot include them in their testing environment.

There are three different types of user testing:

1. *Alpha testing*, where a selected group of software users work closely with the development team to test early releases of the software.
2. *Beta testing*, where a release of the software is made available to a larger group of users to allow them to experiment and to raise problems that they discover with the system developers.
3. *Acceptance testing*, where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In alpha testing, users and developers work together to test a system as it is being developed. This means that the users can identify problems and issues that are not readily apparent to the development testing team. Developers can only really work from the requirements, but these often do not reflect other factors that affect the practical use of the software. Users can therefore provide information about practice that helps with the design of more realistic tests.

Alpha testing is often used when developing software products or apps. Experienced users of these products may be willing to get involved in the alpha testing process because this gives them early information about new system features that they can exploit. It also reduces the risk that unanticipated changes to the software will have disruptive effects on their business. However, alpha testing may also be used when custom software is being developed. Agile development methods advocate user involvement in the development process, and that users should play a key role in designing tests for the system.

Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to a larger group of customers and users for evaluation. Beta testers may be a selected group of customers who are early adopters of the system.

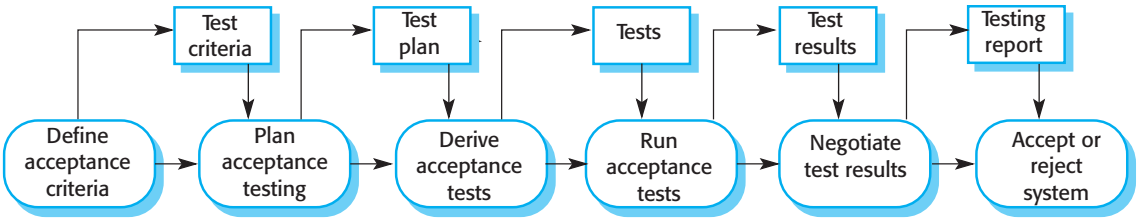


Figure 8.11 The acceptance testing process

Alternatively, the software may be made publicly available for use by anyone who is interested in experimenting with it.

Beta testing is mostly used for software products that are used in many different settings. This is important as, unlike custom product developers, there is no way for the product developer to limit the software's operating environment. It is impossible for product developers to know and replicate all the settings in which the software product will be used. Beta testing is therefore used to discover interaction problems between the software and features of its operational environment. Beta testing is also a form of marketing. Customers learn about their system and what it can do for them.

Acceptance testing is an inherent part of custom systems development. Customers test a system, using their own data, and decide if it should be accepted from the system developer. Acceptance implies that final payment should be made for the software.

Figure 8.11 shows that here are six stages in the acceptance testing process:

1. *Define acceptance criteria* This stage should ideally take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be approved by the customer and the developer. In practice, however, it can be difficult to define criteria so early in the process. Detailed requirements may not be available, and the requirements will almost certainly change during the development process.

2. *Plan acceptance testing* This stage involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

3. *Derive acceptance tests* Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should ideally provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.

4. *Run acceptance tests* The agreed acceptance tests are executed on the system. Ideally, this step should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing

environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

5. *Negotiate test results* It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be used. They must also agree on how the developer will fix the identified problems.

6. *Reject/accept system* This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

You might think that acceptance testing is a clear-cut contractual issue. If a system does not pass its acceptance tests, then it should not be accepted and payment should not be made. However, the reality is more complex. Customers want to use the software as soon as they can because of the benefits of its immediate deployment. They may have bought new hardware, trained staff, and changed their processes. They may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems.

Therefore, the outcome of negotiations may be conditional acceptance of the system. The customer may accept the system so that deployment can begin. The system provider agrees to repair urgent problems and deliver a new version to the customer as quickly as possible.

In agile methods such as Extreme Programming, there may be no separate acceptance testing activity. The end-user is part of the development team (i.e., he or she is an alpha tester) and provides the system requirements in terms of user stories. He or she is also responsible for defining the tests, which decide whether or not the developed software supports the user stories. These tests are therefore equivalent to acceptance tests. The tests are automated, and development does not proceed until the story acceptance tests have successfully been executed.

When users are embedded in a software development team, they should ideally be "typical" users with general knowledge of how the system will be used. However, it can be difficult to find such users, and so the acceptance tests may actually not be a true reflection of how a system is used in practice. Furthermore, the requirement for automated testing limits the flexibility of testing interactive systems. For such systems, acceptance testing may require groups of end-users to use the system as if it was part of their everyday work. Therefore, while an "embedded user" is an attractive notion in principle, it does not necessarily lead to high-quality tests of the system.

The problem of user involvement in agile teams is one reason why many companies use a mix of agile and more traditional testing. The system may be developed using agile techniques, but, after completion of a major release, separate acceptance testing is used to decide if the system should be accepted.

KEY POINTS

- Testing can only show the presence of errors in a program. It cannot show that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers. In the user testing process, customers or system users provide test data and check that tests are successful.
- Development testing includes unit testing in which you test individual objects and methods; component testing in which you test related groups of objects; and system testing in which you test partial or complete systems.
- When testing software, you should try to "break" the software by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development whereby tests are written before the code to be tested. Small code changes are made, and the code is refactored until all tests execute successfully.
- Scenario testing is useful because it replicates the practical use of the system. It involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process in which the aim is to decide if the software is good enough to be deployed and used in its planned operational environment.

FURTHER READING

"How to design practical test cases." A how-to article on test-case design by an author from a Japanese company that has a good reputation for delivering software with very few faults. (T. Yamaura, *IEEE Software*, 15(6), November 1998) http://dx.doi.org/10.1109/52.730835.

"Test-driven development." This special issue on test-driven development includes a good general overview of TDD as well as experience papers on how TDD has been used for different types of software. (*IEEE Software*, 24 (3) May/June 2007).

Exploratory Software Testing. This is a practical, rather than theoretical, book on software testing which develops the ideas in Whittaker's earlier book, *How to Break Software*. The author presents a set of experience-based guidelines on software testing. (J. A. Whittaker, 2009, Addison-Wesley).

How Google Tests Software. This is a book about testing large-scale cloud-based systems and poses a whole set of new challenges compared to custom software applications. While I don't think that the Google approach can be used directly, there are interesting lessons in this book for large-scale system testing. (J. Whittaker, J. Arbon, and J. Carollo, 2012, Addison-Wesley).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

EXERCISES

8.1. Explain how the number of known defects remaining in a program at the time of delivery affects product support.

8.2. Testing is meant to show that a program does what it is intended to do. Why may testers not always know what a program is intended for?

8.3. Some people argue that developers should not be involved in testing their own code but that all testing should be the responsibility of a separate team. Give arguments for and against testing by the developers themselves.

8.4. You have been asked to test a method called catWhiteSpace in a "Paragraph" object that, within the paragraph, replaces sequences of blank characters with a single blank character. Identify testing partitions for this example and derive a set of tests for the catWhiteSpace method.

8.5. What is regression testing? Explain how the use of automated tests and a testing framework such as JUnit simplifies regression testing.

8.6. The Mentcare system is constructed by adapting an off-the-shelf information system. What do you think are the differences between testing such a system and testing software that is developed using an object-oriented language such as Java?

8.7. Write a scenario that could be used to help design tests for the wilderness weather station system.

8.8. What do you understand by the term *stress testing*? Suggest how you might stress-test the Mentcare system.

8.9. What are the benefits of involving users in release testing at an early stage in the testing process? Are there disadvantages in user involvement?

8.10. A common approach to system testing is to test the more important functionalities of a system first, followed by the less important functionalities until the testing budget is exhausted. Discuss the ethics involved in identifying what "more important" means.

REFERENCES

Andrea, J. 2007. "Envisioning the Next Generation of Functional Testing Tools." *IEEE Software* 24 (3): 58–65. doi:10.1109/MS.2007.73.

Beck, K. 2002. *Test Driven Development: By Example*. Boston: Addison-Wesley.

Bezier, B. 1990. *Software Testing Techniques, 2nd ed*. New York: Van Nostrand Reinhold.

Boehm, B. W. 1979. "Software Engineering; R & D Trends and Defense Needs." In *Research Directions in Software Technology*, edited by P. Wegner, 1–9. Cambridge, MA: MIT Press.

Cusamano, M., and R. W. Selby. 1998. *Microsoft Secrets*. New York: Simon & Schuster.

Dijkstra, E. W. 1972. "The Humble Programmer." *Comm. ACM* 15 (10): 859–866. doi:10.1145/355604.361591.

Fagan, M. E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J.* 15 (3): 182–211.

Jeffries, R., and G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.

Kaner, C. 2003. "An Introduction to Scenario Testing." *Software Testing and Quality Engineering* (October 2003).

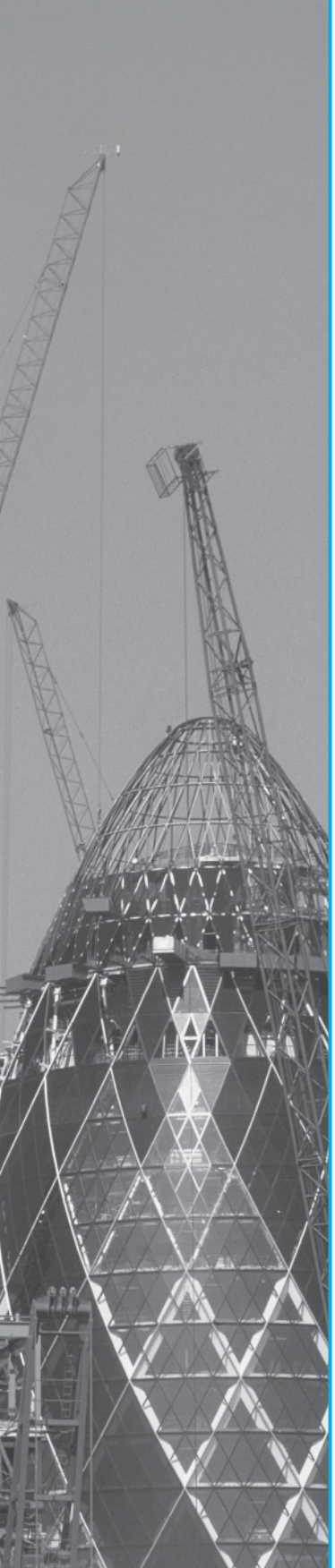
Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In *RE' 93*, 126–133. San Diego CA: IEEE. doi:o.1109/ISRE.1993.324825.

Martin, R. C. 2007. "Professionalism and Test-Driven Development." *IEEE Software* 24 (3): 32–36. doi:10.1109/MS.2007.85.

Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.

Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2nd ed.* Greenwich, CT: Manning Publications.

Whittaker, J. A. 2009. *Exploratory Software Testing*. Boston: Addison-Wesley.



9 Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is such an important part of software engineering and to describe the challenges of maintaining a large base of software systems, developed over many years. When you have read this chapter, you will:

- understand that software systems have to adapt and evolve if they are to remain useful and that software change and evolution should be considered as an integral part of software engineering;
- understand what is meant by legacy systems and why these systems are important to businesses;
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced;
- have learned about different types of software maintenance and the factors that affect the costs of making changes to legacy software systems.

Contents

9.1 Evolution processes
9.2 Legacy systems
9.3 Software maintenance

Large software systems usually have a long lifetime. For example, military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment. Successful software products and apps may have been introduced many years ago with new versions released every few years. For example, the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.

During their lifetime, operational software systems have to change if they are to emain useful. Business changes and changes to user expectations generate new requirements for the software. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics. Software products and apps have to evolve to cope with platform changes and new features introduced by their competitors. Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Businesses have to change their software to ensure that they continue to get value from it. Their systems are critical business assets, and they have to invest in change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Historical data suggests that somewhere between 60% and 90% of software costs are evolution costs (Lientz and Swanson 1980; Erlikh 2000). Jones (Jones 2006) found that about 75% of development staff in the United States in 2006 were involved in software evolution and suggested that this percentage was unlikely to fall in the foreseeable future.

Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader "system of systems." In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems. Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.

Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment. Hopkins and Jenkins (Hopkins and Jenkins 2008) have coined the term *brownfield software development* to describe situations in which software systems have to be developed and managed in an environment where they are dependent on other software systems.

The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals. Software engineering is therefore a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 9.1). You start by creating release 1 of the system. Once delivered, changes are proposed, and the development of release 2 starts almost immediately. In fact, the need for evolution may become obvious even before the system is deployed, so later releases of the software may start development before the current version has even been released.

In the last 10 years, the time between iterations of the spiral has reduced dramatically. Before the widespread use of the Internet, new versions of a software system

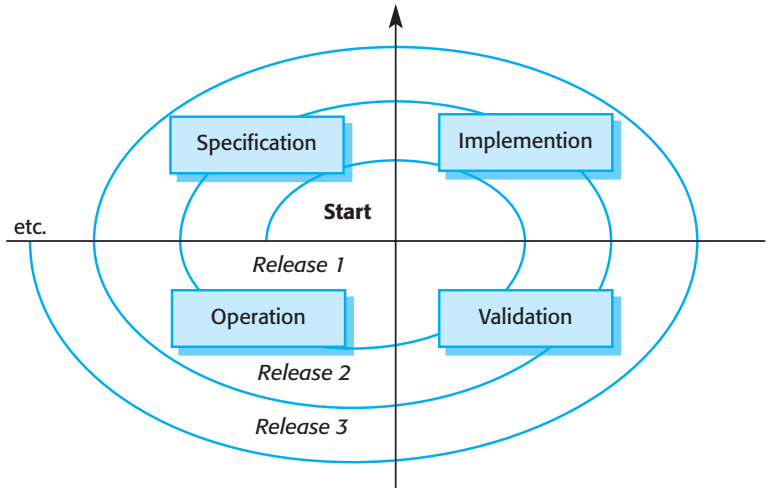


Figure 9.1 A spiral model of development and evolution

may only have been released every 2 or 3 years. Now, because of competitive pressures and the need to respond quickly to user feedback, the gap between releases of some apps and web-based systems may be weeks rather than years.

This model of software evolution is applicable when the same company is responsible for the software throughout its lifetime. There is a seamless transition from development to evolution, and the same software development methods and processes are applied throughout the lifetime of the software. Software products and apps are developed using this approach.

The evolution of custom software, however, usually follows a different model. The system customer may pay a software company to develop the software and then take over responsibility for support and evolution using its own staff. Alternatively, the software customer might issue a separate contract to a different software company for system support and evolution.

In this situation, there are likely to be discontinuities in the evolution process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize, inherit software from other companies, and then find that this has to be changed. When the transition from development to evolution is not seamless, the process of changing the software after delivery is called software maintenance. As I discuss later in this chapter, maintenance involves extra process activities, such as program understanding, in addition to the normal activities of software development.

Rajlich and Bennett (Rajlich and Bennett 2000) propose an alternative view of the software evolution life cycle for business systems. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality are made. During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure 9.2.

According to Rajlich and Bennett, when software is first used successfully, many changes to the requirements by stakeholders are proposed and implemented. This is

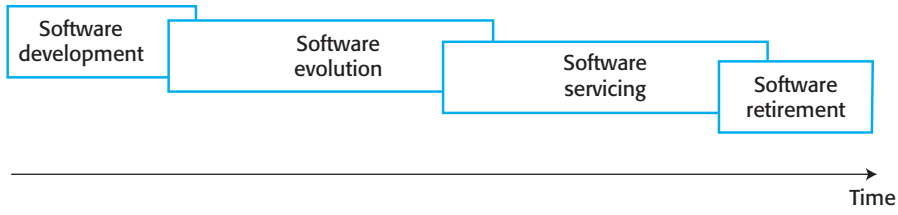


Figure 9.2 Evolution and servicing

the evolution phase. However, as the software is modified, its structure tends to degrade, and system changes become more and more expensive. This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also required. At some stage in the life cycle, the software reaches a transition point where significant changes and the implementation of new requirements become less and less cost-effective. At this stage, the software moves from evolution to servicing.

During the servicing phase, the software is still useful, but only small tactical changes are made to it. During this stage, the company is usually considering how the software can be replaced. In the final stage, the software may still be used, but only essential changes are made. Users have to work around problems that they discover. Eventually, the software is retired and taken out of use. This often incurs further costs as data is transferred from an old system to a newer replacement system.

9.1 Evolution processes

As with all software processes, there is no such thing as a standard software change or evolution process. The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved. For some types of system, such as mobile apps, evolution may be an informal process, where change requests mostly come from conversations between system users and developers. For other types of systems, such as embedded critical systems, software evolution may be formalized, with structured documentation produced at each stage in the process.

Formal or informal system change proposals are the driver for system evolution in all organizations. In a change proposal, an individual or group suggests changes and updates to an existing software system. These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system (Figure 9.3).

Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed. This analysis allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which should also ensure that the correct versions of

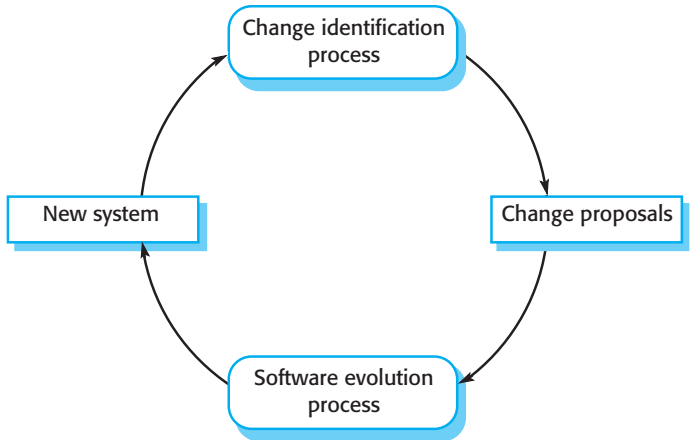


Figure 9.3 Change identification and evolution processes

components are included in each system release. I discuss change and configuration management in Chapter 25.

Figure 9.4 shows some of the activities involved in software evolution. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

In situations where development and evolution are integrated, change implementation is simply an iteration of the development process. Revisions to the system are designed, implemented, and tested. The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.

Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

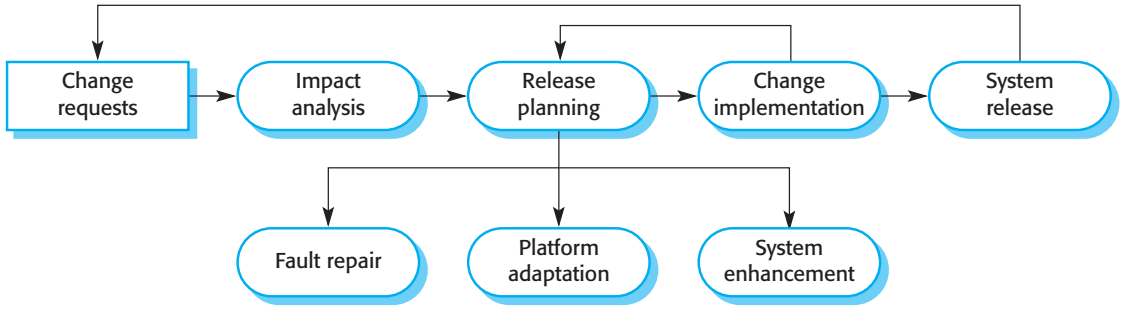


Figure 9.4 A general model of the software evolution process

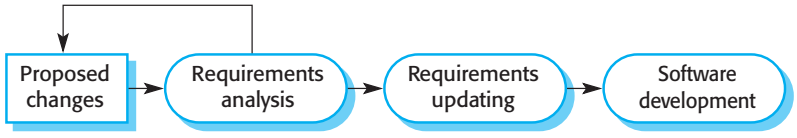


Figure 9.5 Change implementation

During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required (Figure 9.5). New software requirements should be written, and these should be analyzed and validated. If the design has been documented using UML models, these models should be updated. The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

However, change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent. While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten, and the system documentation and code are never realigned. This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

Emergency system repairs have to be completed as quickly as possible. You choose a quick and workable solution rather than the best solution as far as system structure is concerned. This tends to accelerate the process of software ageing so that future changes become progressively more difficult and maintenance costs increase. Ideally, after emergency code repairs are made, the new code should be refactored

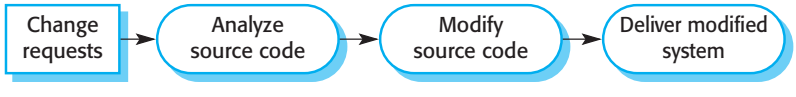


Figure 9.6 The emergency repair process

and improved to avoid program degradation. Of course, the code of the repair may be reused if possible. However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.

However, problems may arise during the handover from a development team to a separate team responsible for system evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes. There may be no definitive statement of the system requirements that can be modified as changes are made to the system.

2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests. The code in the system may not have been refactored and simplified, as is expected in agile development. In this case, some program reengineering may be required to improve the code before it can be used in an agile development process.

Agile techniques such as test-driven development and automated regression testing are useful when system changes are made. System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system. The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes. In short, evolution simply involves continuing the agile development process.

Agile methods used in development may, however, have to be modified when they are used for program maintenance and evolution. It may be practically impossible to involve users in the development team as change proposals come from a wide range of stakeholders. Short development cycles may have to be interrupted to deal with emergency repairs, and the gap between releases may have to be lengthened to avoid disrupting operational processes.

9.2 Legacy systems

Large companies started computerizing their operations in the 1960s, so for the past 50 years or so, more and more software systems have been introduced. Many of these systems have been replaced (sometimes several times) as the business has changed and evolved. However, a lot of old systems are still in use and play a critical part in the running of the business. These older software systems are sometimes called legacy systems.

Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development. Typically, they have been maintained over a long period, and their structure may have been degraded by the changes that have been made. Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures. It may be impossible to change to more effective business processes because the legacy software cannot be modified to support new processes.

Legacy systems are not just software systems but are broader sociotechnical systems that include hardware, software, libraries, and other supporting software and business processes. Figure 9.7 shows the logical parts of a legacy system and their relationships.

1. *System hardware* Legacy systems may have been written for hardware that is no longer available, that is expensive to maintain, and that may not be compatible with current organizational IT purchasing policies.
2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
3. *Application software* The application system that provides the business services is usually made up of a number of application programs that have been developed at different times. Some of these programs will also be part of other application software systems.
4. *Application data* These data are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent, may be duplicated in several files, and may be spread over a number of different databases.
5. *Business processes* These processes are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.
6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 9.8.

Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers. In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may

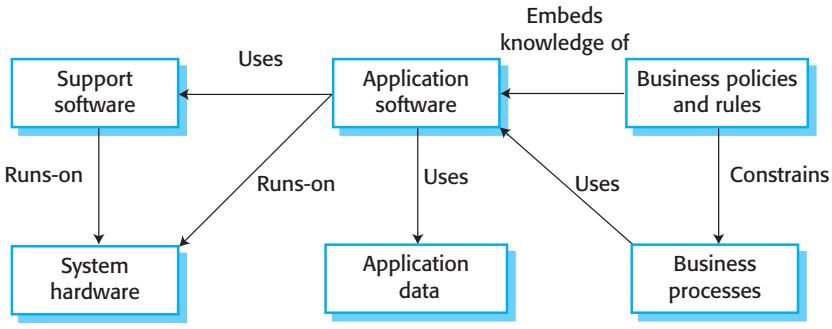


Figure 9.7 The elements of a legacy system

require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.
3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced. This is a particular problem in embedded systems where there is a tight coupling between software and hardware. Major changes to the application software may be required to make effective use of the new hardware.

It is difficult to know exactly how much legacy code is still in use, but, as an indicator, industry has estimated that there are more than 200 billion lines of COBOL code in current business systems. COBOL is a programming language designed for writing business systems, and it was the main business development language from the 1960s to the 1990s, particularly in the finance industry (Mitchell 2012). These programs still work effectively and efficiently, and the companies using them see no need to change them. A major problem that they face, however, is a shortage of COBOL programmers as the original developers of the system retire. Universities no longer teach COBOL, and younger software engineers are more interested in programming in modern languages.

Skill shortages are only one of the problems of maintaining business legacy systems. Other issues include security vulnerabilities because these systems were developed before the widespread use of the Internet and problems in interfacing with systems written in modern programming languages. The original software tool supplier may be out of business or may no longer maintain the support tools used to

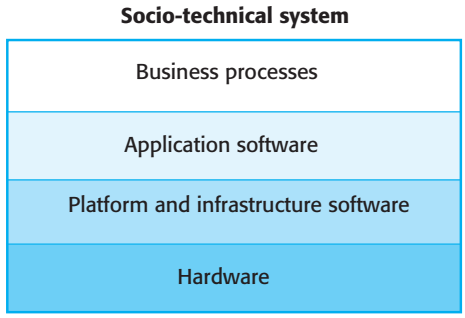


Figure 9.8 Legacy system layers

develop the system. The system hardware may be obsolete and so increasingly expensive to maintain.

Why then do businesses not simply replace these systems with more modern equivalents? The simple answer to this question is that it is too expensive and too risky to do so. If a legacy system works effectively, the costs of replacement may exceed the savings that come from the reduced support costs of a new system. Scrapping legacy systems and replacing them with more modern software open up the possibility of things going wrong and the new system failing to meet the needs of the business. Managers try to minimize those risks and therefore do not want to face the uncertainties of new software systems.

I discovered some of the problems of legacy system replacement when I was involved in analyzing a legacy system replacement project in a large organization. This enterprise used more than 150 legacy systems to run its business. It decided to replace all of these systems with a single, centrally maintained ERP system. For a number of business and technology reasons, the new system development was a failure, and it did not deliver the improvements promised. After spending more than £10 million, only a part of the new system was operational, and it worked less effectively than the systems it replaced. Users continued to use the older systems but could not integrate these with the part of the new system that had been implemented, so additional manual processing was required.

There are several reasons why it is expensive and risky to replace legacy systems with new systems:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it has been updated with all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system that is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes are likely to have evolved to take advantage of the software's services and to work around the software's shortcomings. If the system is replaced, these processes have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint that applies to some business function, and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies that could result in expensive future claims.
4. New software development is inherently risky, so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected.

Keeping legacy systems in use avoids the risks of replacement, but making changes to existing software inevitably becomes more expensive as systems get older. Legacy software systems that are more than a few years old are particularly expensive to change:

1. The program style and usage conventions are inconsistent because different people have been responsible for system changes. This problem adds to the difficulty of understanding the system code.
2. Part or all of the system may be implemented using obsolete programming languages. It may be difficult to find people who have knowledge of these languages. Expensive outsourcing of system maintenance may therefore be required.
3. System documentation is often inadequate and out of date. In some cases, the only documentation is the system source code.
4. Many years of maintenance usually degrades the system structure, making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.
5. The system may have been optimized for space utilization or execution speed so that it runs effectively on older slower hardware. This normally involves using specific machine and language optimizations, and these usually lead to software that is hard to understand. This causes problems for programmers who have learned modern software engineering techniques and who don't understand the programming tricks that have been used to optimize the software.
6. The data processed by the system may be maintained in different files that have incompatible structures. There may be data duplication, and the data itself may be out of date, inaccurate, and incomplete. Several databases from different suppliers may be used.

At same stage, the costs of managing and maintaining the legacy system become so high that it has to be replaced with a new system. In the next section, I discuss a systematic decision-making approach to making such a replacement decision.

9.2.1 Legacy system management

For new software systems developed using modern software engineering processes, such as agile development and software product lines, it is possible to plan how to integrate system development and evolution. More and more companies understand that the system development process is a whole life-cycle process. Separating software development and software evolution is unhelpful and leads to higher costs. However, as I have discussed, there is still a huge number of legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations have a limited budget for maintaining and upgrading their portfolio of legacy systems. They have to decide how to get the best return on their investment. This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems. There are four strategic options:

1. *Scrap the system completely* This option should be chosen when the system is not making an effective contribution to business processes. This usually occurs when business processes have changed since the system was installed and are no longer reliant on the legacy system.
2. *Leave the system unchanged and continue with regular maintenance* This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.
3. *Reengineer the system to improve its maintainability* This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.
4. *Replace all or part of the system with a new system* This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost. In many cases, an evolutionary replacement strategy can be adopted where major system components are replaced by off-the-shelf systems with other components reused wherever possible.

When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective (Warren 1998). From a business perspective, you have to decide whether or not the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware. You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

For example, assume that an organization has 10 legacy systems. You should assess the quality and the business value of each of these systems. You may then create a chart showing relative business value and system quality. An example of

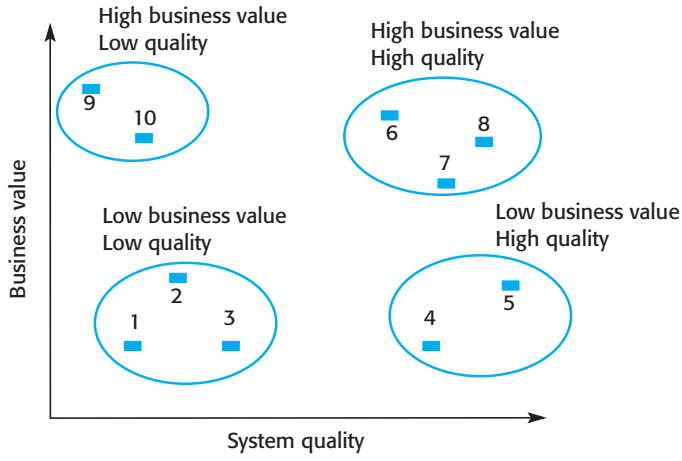


Figure 9.9 An example of a legacy system assessment

this is shown in Figure 9.9. From this diagram, you can see that there are four clusters of systems:

1. *Low quality, low business value* Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.
3. *High quality, low business value* These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems. To assess the business value of a system, you have to identify system stakeholders, such as the end-users of a system and their managers, and ask a series of questions about the system. There are four basic issues that you have to discuss:

1. *The use of the system* If a system is only used occasionally or by a small number of people, this may mean that it has a low business value. A legacy system may have been developed to meet a business need that has either changed or can now be met

more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, a university system for student registration may only be used at the beginning of each academic year. Although it is used infrequently, it is an essential system with a high business value.

2. *The business processes that are supported* When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. *System dependability* System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect business customers, or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. *The system outputs* The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

For example, assume that a company provides a travel ordering system that is used by staff responsible for arranging travel. They can place orders with an approved travel agent. Tickets are then delivered, and the company is invoiced for them. However, a business value assessment may reveal that this system is only used for a fairly small percentage of travel orders placed. People making travel arrangements find it cheaper and more convenient to deal directly with travel suppliers through their websites. This system may still be used, but there is no real point in keeping it—the same functionality is available from external systems.

Conversely, say a company has developed a system that keeps track of all previous customer orders and automatically generates reminders for customers to reorder goods. This results in a large number of repeat orders and keeps customers satisfied because they feel that their supplier is aware of their needs. The outputs from such a system are important to the business, so this system has a high business value.

To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software such as compilers, debuggers and development environments that are needed to maintain the system. The environment is important because many system changes, such as upgrades to the hardware or operating system, result from changes to the environment.

Factors that you should consider during the environment assessment are shown in Figure 9.10. Notice that these are not all technical characteristics of the environment. You also have to consider the reliability of the suppliers of the hardware and support software. If suppliers are no longer in business, their systems may not be supported, so you may have to replace these systems.

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly, but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If high costs are associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system?

Figure 9.10 Factors used in environment assessment

In the process of environmental assessment, if possible, you should ideally collect data about the system and system changes. Examples of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software.

To assess the technical quality of an application system, you have to assess those factors (Figure 9.11) that are primarily related to the system dependability, the difficulties of maintaining the system, and the system documentation. You may also collect data that will help you judge the quality of the system such as:

1. *The number of system change requests* System changes usually corrupt the system structure and make further changes more difficult. The higher this accumulated value, the lower the quality of the system.
2. *The number of user interfaces* This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
3. *The volume of data used by the system* As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data. When data has been collected over a long period of time, errors and inconsistencies are inevitable. Cleaning up old data is a very expensive and time-consuming process.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.11 Factors used in application assessment

Ideally, objective assessment should be used to inform decisions about what to do with a legacy system. However, in many cases, decisions are not really objective but are based on organizational or political considerations. For example, if two businesses merge, the most politically powerful partner will usually keep its systems and scrap the other company's systems. If senior management in an organization decides to move to a new hardware platform, then this may require applications to be replaced. If no budget is available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

9.3 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software, where separate development groups are involved before and after delivery. The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.



Program evolution dynamics

Program evolution dynamics is the study of evolving software systems, pioneered by Manny Lehman and Les Belady in the 1970s. This led to so-called Lehman's Laws, which are said to apply to all large-scale software systems. The most important of these laws are:

1. A program must continually change if it is to remain useful.
2. As an evolving program changes, its structure is degraded.
3. Over a program's lifetime, the rate of change is roughly constant and independent of the resources available.
4. The incremental change in each release of a system is roughly constant.
5. New functionality must be added to systems to increase user satisfaction.

http://software-engineering-book.com/web/program-evolution-dynamics/

There are three different types of software maintenance:

1. *Fault repairs to fix bugs and vulnerabilities.* Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
2. *Environmental adaptation to adapt the software to new platforms and environments.* This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is no clear-cut distinction between these types of maintenance. When you adapt a system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

These types of maintenance are generally recognized, but different people sometimes give them different names. "Corrective maintenance" is universally used to refer to maintenance for fault repair. However, "adaptive maintenance" sometimes means adapting to a new environment and sometimes means adapting the software to new requirements. "Perfective maintenance" sometimes means perfecting the software by implementing new requirements; in other cases, it means maintaining the functionality of the system but improving its structure and its performance. Because of this naming uncertainty, I have avoided the use of these terms in this book.

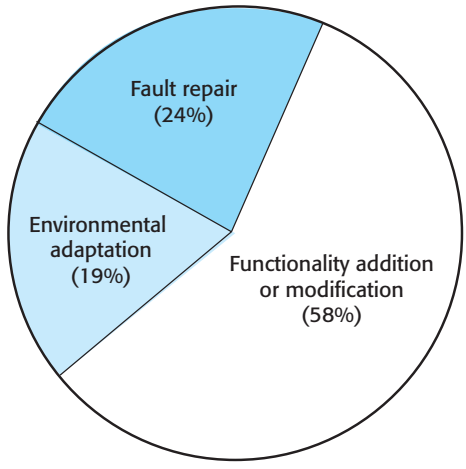


Figure 9.12 Maintenance effort distribution

Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010). This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005. The authors found that the distribution of maintenance costs had changed very little over 30 years. Although we don't have more recent data, this suggests that this distribution is still largely correct. Repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.

Experience has shown that it is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. *A new team has to understand the program being maintained.* After a system has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before they can implement changes to it.

2. *Separating maintenance and development means there is no incentive for the development team to write maintainable software.* The contract to maintain a system is usually separate from the system development contract. A different company, rather than the original software developer, may be responsible for software maintenance. In those circumstances, a development team gets no benefit from investing effort to make the software maintainable. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in future.

3. *Program maintenance work is unpopular.* Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development



Documentation

System documentation can help the maintenance process by providing maintainers with information about the structure and organization of the system and the features that it offers to system users. While proponents of agile approaches suggest that the code should be the principal documentation, higher level design models and information about dependencies and constraints can make it easier to understand and make changes to that code.

http://software-engineering-book.com/web/documentation/ (web chapter)

and is often allocated to the least experienced staff. Furthermore, old systems may be written in obsolete programming languages. The developers working on maintenance may not have much experience of these languages and must learn these languages to maintain the system.

4. *As programs age, their structure degrades and they become harder to change.* As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change. Some systems have been developed without modern software engineering techniques. They may never have been well structured and were perhaps optimized for efficiency rather than understandability. System documentation may be lost or inconsistent. Old systems may not have been subject to stringent configuration management, so developers have to spend time finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider software development and maintenance to be separate activities. Maintenance is seen as a second-class activity, and there is no incentive to spend money during development to reduce the costs of system change. The only long-term solution to this problem is to think of systems as evolving throughout their lifetime through a continual development process. Maintenance should have as high a status as new software development.

The fourth issue, the problem of degraded system structure, is, in some ways, the easiest problem to address. Software reengineering techniques (described later in this chapter) may be applied to improve the system structure and understandability. Architectural transformations can adapt the system to new hardware. Refactoring can improve the quality of the system code and make it easier to change.

In principle, it is almost always cost-effective to invest effort in designing and implementing a system to reduce the costs of future changes. Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes. Work done during development to structure the software and to make it easier to understand and change will reduce evolution costs. Good software engineering techniques such as precise specification, test-first development, the use of object-oriented development, and configuration management all help reduce maintenance cost.

These principled arguments for lifetime cost savings by investing in making systems more maintainable are, unfortunately, impossible to substantiate with real

data. Collecting data is expensive, and the value of that data is difficult to judge; therefore, the vast majority of companies do not think it is worthwhile to gather and analyze software engineering data.

In reality, most businesses are reluctant to spend more on software development to reduce longer-term maintenance costs. There are two main reasons for their reluctance:

1. Companies set out quarterly or annual spending plans, and managers are incentivized to reduce short-term costs. Investing in maintainability leads to short-term cost increases, which are measurable. However, the long-term gains can't be measured at the same time, so companies are reluctant to spend money on something with an unknown future return.
2. Developers are not usually responsible for maintaining the system they have developed. Consequently, they don't see the point of doing additional work that might reduce maintenance costs, as they will not get any benefit from it.

The only way around this problem is to integrate development and maintenance so that the original development team remains responsible for software throughout its lifetime. This is possible for software products and for companies such as Amazon, which develop and maintain their own software (O'Hanlon 2006). However, for custom software developed by a software company for a client, this is unlikely to happen.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change. If you understand this, you can design the software components that are most likely to change to make them more adaptable. You can also invest effort in improving those components to reduce their lifetime maintenance costs. By predicting changes, you can also assess the overall maintenance costs for a system in a given time period and so set a budget for maintaining the software. Figure 9.13 shows possible predictions and the questions that these predictions may answer.

Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.

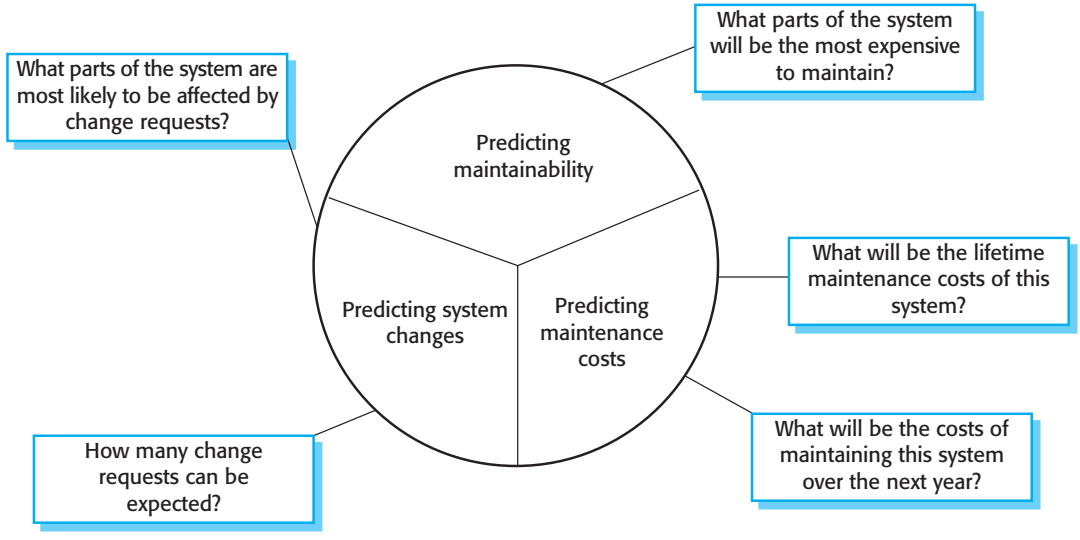


Figure 9.13
Maintenance prediction

2. *The number of inherently volatile system requirements* As I discussed in Chapter 4, requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. As a system is integrated with more and more business processes, there are increased demands for changes.

In early work on software maintenance, researchers looked at the relationships between program complexity and maintainability (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). These studies found that the more complex a system or component, the more expensive it is to maintain. Complexity measurements are particularly useful in identifying program components that are likely to be expensive to maintain. Therefore, to reduce maintenance costs you should try to replace complex system components with simpler alternatives.

After a system has been put into service, you may be able to use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.

2. *Average time required for impact analysis* This is related to the number of program components that are affected by the change request. If the time required for impact analysis increases, it implies that more and more components are affected and maintainability is decreasing.

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.
4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to predict maintenance costs. Most managers combine this information with intuition and experience to estimate costs. The COCOMO 2 model of cost estimation, discussed in Chapter 23, suggests that an estimate for software maintenance effort can be based on the effort to understand existing code and the effort to develop the new code.

9.3.2 Software reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, many systems, especially older legacy systems, are difficult to understand and change. The programs may have been optimized for performance or space utilization at the expense of understandability, or, over time, the initial program structure may have been corrupted by a series of changes.

To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Reengineering has two important advantages over replacement:

1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.
2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. Ulrich (Ulrich 1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million. The system was successfully reengineered for $12 million. I suspect that, with modern software technology, the relative cost of reimplementation is probably less than Ulrich's figure but will still be more than the costs of reengineering.

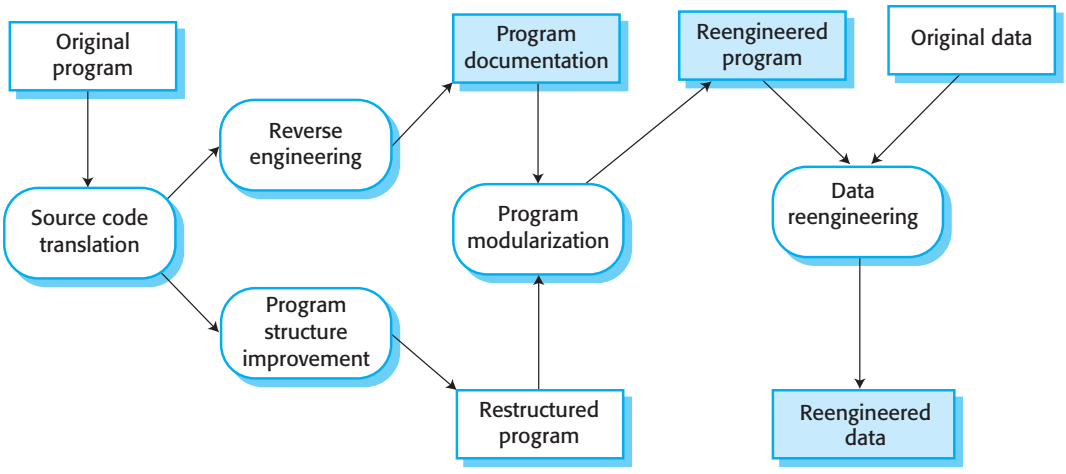


Figure 9.14 The reengineering process

Figure 9.14 is a general model of the reengineering process. The input to the process is a legacy program, and the output is an improved and restructured version of the same program. The activities in this reengineering process are:

1. *Source code translation* Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.
2. *Reverse engineering* The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. *Program structure improvement* The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.
4. *Program modularization* Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. *Data reengineering* The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Program reengineering may not necessarily require all of the steps in Figure 9.11. You don't need source code translation if you still use the application's programming language. If you can do all reengineering automatically, then recovering documentation through reverse engineering may be unnecessary. Data reengineering is required only if the data structures in the program change during system reengineering.

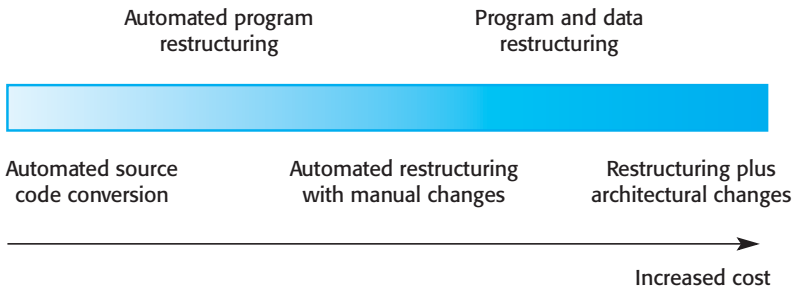


Figure 9.15 Reengineering approaches

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 18. These hide the original interfaces of the software system and present new, better-structured interfaces that can be used by other components. This process of legacy system wrapping is an important technique for developing large-scale reusable services.

The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 9.15. Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.

The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive. Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

9.3.3 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach. When you refactor a program, you should not add functionality but rather should concentrate on program improvement. You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

Refactoring is an inherent part of agile methods because these methods are based on change. Program quality is liable to degrade quickly, so agile developers frequently refactor their programs to avoid this degradation. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Any errors that are introduced should be detectable, as previously successful tests should then fail. However, refactoring is not dependent on other "agile activities."

Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing. Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Fowler et al. (Fowler et al. 1999) suggest that there are stereotypical situations (Fowler calls them "bad smells") where the code of a program can be improved. Examples of bad smells that can be improved through refactoring include:

1. *Duplicate code* The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
2. *Long methods* If a method is too long, it should be redesigned as a number of shorter methods.
3. *Switch (case) statements* These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
4. *Data clumping* Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. *Speculative generality* This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Fowler, in both his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with bad smells. Examples of these transformations include Extract method, where you remove duplication and create a new method; Consolidate conditional expression, where you replace a sequence of tests with a single test; and Pull up method, where you replace similar methods in subclasses with a single method in a superclass. Interactive development environments, such as Eclipse, usually include refactoring support in their editors. This makes it easier to find dependent parts of a program that have to be changed to implement the refactoring.

Refactoring, carried out during program development, is an effective way to reduce the long-term maintenance costs of a program. However, if you take over a program for maintenance whose structure has been significantly degraded, then it may be practically impossible to refactor the code alone. You may also have to think about design refactoring, which is likely to be a more expensive and difficult problem. Design refactoring involves identifying relevant design patterns (discussed in Chapter 7) and replacing existing code with code that implements these design patterns (Kerievsky 2004).

KEY POINTS

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.
- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether a system should be replaced, transformed, or maintained.
- There are three types of software maintenance, namely, bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software reengineering is concerned with restructuring and redocumenting software to make it easier to understand and change.
- Refactoring, making small program changes that preserve functionality, can be thought of as preventative maintenance.

FURTHER READING

Working Effectively with Legacy Code. Solid practical advice on the problems and difficulties of dealing with legacy systems. (M. Feathers, 2004, John Wiley & Sons).

"The Economics of Software Maintenance in the 21st Century." This article is a general introduction to maintenance and a comprehensive discussion of maintenance costs. Jones discusses the factors that affect maintenance costs and suggests that almost 75% of the software workforce are involved in software maintenance activities. (C. Jones, 2006) http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf

"You Can't Be Agile in Maintenance?" In spite of the title, this blog post argues that agile techniques are appropriate for maintenance and discusses which techniques as suggested in XP can be effective. (J. Bird, 2011) http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html

"Software Reengineering and Testing Considerations." This is an excellent summary white paper of maintenance issues from a major Indian software company. (Y. Kumar and Dipti, 2012) http://www.infosys.com/engineering-services/white-papers/Documents/software-re-engineering-processes.pdf

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

EXERCISES

9.1. Explain how advances in technology can force a software subsystem to undergo change or run the risk of becoming useless.

9.2. From Figure 9.4, you can see that impact analysis is an important subprocess in the software evolution process. Using a diagram, suggest what activities might be involved in change impact analysis.

9.3. Explain why legacy systems should be thought of as sociotechnical systems rather than simply software systems that were developed using old technology.

9.4. Some software subsystems are seen as "low quality, high business value." Discuss how those subsystems can be re-engineered with minimal impact on the operations of the organization.

9.5. What are the strategic options for legacy system evolution? When would you normally replace all or part of a system rather than continue maintenance of the software?

9.6. Explain why problems with support software might mean that an organization has to replace its legacy systems.

9.7. As a software project manager in a company that specializes in the development of software for the offshore oil industry, you have been given the task of discovering the factors that affect the maintainability of the systems developed by your company. Suggest how you might set up a program to analyze the maintenance process and determine appropriate maintainability metrics for the company.

9.8. Briefly describe the three main types of software maintenance. Why is it sometimes difficult to distinguish between them?

9.9. Explain the differences between software reengineering and refactoring?

9.10. Do software engineers have a professional responsibility to develop code that can be easily maintained even if their employer does not explicitly request it?

REFERENCES

Banker, R. D., S. M. Datar, C. F. Kemerer, and D. Zweig. 1993. "Software Complexity and Maintenance Costs." *Comm. ACM* 36 (11): 81–94. doi:10.1145/163359.163375.

Coleman, D., D. Ash, B. Lowther, and P. Oman. 1994. "Using Metrics to Evaluate Software System Maintainability." *IEEE Computer* 27 (8): 44–49. doi:10.1109/2.303623.

Davidsen, M. G., and J. Krogstie. 2010. "A Longitudinal Study of Development and Maintenance." *Information and Software Technology* 52 (7): 707–719. doi:10.1016/j.infsof.2010.03.003.

Erlikh, L. 2000. "Leveraging Legacy System Dollars for E-Business." *IT Professional* 2 (3 (May/June 2000)): 17–23. doi:10.1109/6294.846201.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.

Jones, T. C. 2006. "The Economics of Software Maintenance in the 21st Century." www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf.

Kerievsky, J. 2004. *Refactoring to Patterns*. Boston: Addison-Wesley.

Kozlov, D., J. Koskinen, M. Sakkinen, and J. Markkula. 2008. "Assessing Maintainability Change over Multiple Software Releases." *J. of Software Maintenance and Evolution* 20 (1): 31–58. doi:10.1002/smr.361.

Lientz, B. P., and E. B. Swanson. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.

Mitchell, R. M. 2012. "COBOL on the Mainframe: Does It Have a Future?" *Computerworld US*. http://features.techworld.com/applications/3344704/cobol-on-the-mainframe-does-it-have-a-future/

O'Hanlon, C. 2006. "A Conversation with Werner Vogels." *ACM Queue* 4 (4): 14–22. doi:10.1145/1142055.1142065.

Rajlich, V. T., and K. H. Bennett. 2000. "A Staged Model for the Software Life Cycle." *IEEE Computer* 33 (7): 66–71. doi:10.1109/2.869374.

Ulrich, W. M. 1990. "The Evolutionary Growth of Software Reengineering and the Decade Ahead." *American Programmer* 3 (10): 14–20.

Warren, I. (ed.). 1998. *The Renaissance of Legacy Systems*. London: Springer.



PART 2 Dependability and Security

As software systems are now part of all aspects of our lives, I believe that the most significant challenge that we face in software engineering is ensuring that we can trust these systems. To trust a system, we must have confidence that it will be available when required and perform as expected. It must be secure so that our computers or data are not threatened by it and it has to recover quickly in the event of failure or cyberattack. This part of the book has therefore focuses on the important topics of software system dependability and security.

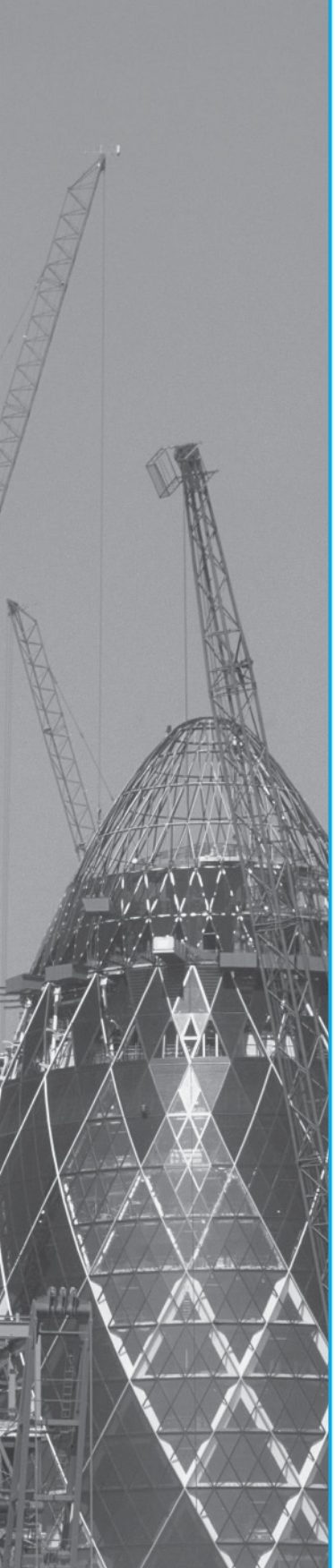
Chapter 10 introduces the basic concepts of dependability and security namely reliability, availability, safety, security and resilience. I explain why building secure, dependable systems is not simply a technical problem. I introduce redundancy and diversity as the fundamental mechanisms used to create dependable and secure systems. The individual dependability attributes are covered in more detail in the following chapters.

Chapter 11 focuses on reliability and availability and I explain how these attributes can be specified as probabilities of failure or downtime. I discuss a number of architectural patterns for fault-tolerant system architectures and development techniques that can be used to reduce the number of faults in a system. In the final section, I explain how the reliability of a system may be tested and measured.

More and more systems are safety-critical systems, where system failure can endanger people. Chapter 12 is concerned with safety engineering and techniques that may be used to develop these safety-critical systems. I explain why safety is a broader notion than reliability and discuss methods for deriving system safety requirements. I also explain why defined and documented processes for safety-critical systems engineering are important and describe software safety cases—structured documents that are used to justify why a system is safe.

Threats to the security of our systems are one of the major problems faced by today's societies and I devote two chapters to this topic. Chapter 13 is concerned with application security engineering—methods used to achieve security in individual software systems. I explain the relationships between security and other dependability attributes and cover security requirements engineering, secure systems design and security testing.

Chapter 14 is a new chapter that addresses the broader issue of resilience. A resilient system can continue to deliver its essential services even when individual parts of the system fail or are subject to a cyberattack. I explain the basics of cybersecurity and discuss how resilience is achieved by using redundancy and diversity and by empowering people as well as through technical mechanisms. Finally, I discuss systems and software design issues that can contribute to improving the resilience of a system.



10 Dependable systems

Objectives

The objective of this chapter is to introduce the topic of software dependability and what is involved in developing dependable software systems. When you have read this chapter, you will:

- understand why dependability and security are important attributes for all software systems;
- understand the five important dimensions of dependability, namely, availability, reliability, safety, security, and resilience;
- understand the notion of sociotechnical systems and why we have to consider these systems as a whole rather than just software systems;
- know why redundancy and diversity are the fundamental concepts used in achieving dependable systems and processes;
- be aware of the potential for using formal methods in dependable systems engineering.

Contents

10.1 Dependability properties
10.2 Sociotechnical systems
10.3 Redundancy and diversity
10.4 Dependable processes
10.5 Formal methods and dependability

As computer systems have become deeply embedded in our business and personal lives, the problems that result from system and software failure are increasing. A failure of server software in an e-commerce company could lead to a major loss of revenue and customers for that company. A software error in an embedded control system in a car could lead to expensive recalls of that model for repair and, in the worst case, could be a contributory factor in accidents. The infection of company PCs with malware requires expensive clean-up operations to sort out the problem and could lead to the loss of or damage to sensitive information.

Because software-intensive systems are so important to governments, companies, and individuals, we have to be able to trust these systems. The software should be available when it is needed, and it should operate correctly without undesirable side effects, such as unauthorized information disclosure. In short, we should be able to depend on our software systems.

The term *dependability* was proposed by Jean-Claude Laprie in 1995 to cover the related systems attributes of availability, reliability, safety, and security. His ideas were revised over the next few years and are discussed in a definitive paper published in 2004 (Avizienis et al. 2004). As I discuss in Section 10.1, these properties are inextricably linked, so having a single term to cover them all makes sense.

The dependability of systems is usually more important than their detailed functionality for the following reasons:

1. *System failures affect a large number of people* Many systems include functionality that is rarely used. If this functionality were left out of the system, only a small number of users would be affected. System failures that affect the availability of a system potentially affect all users of the system. Unavailable systems may mean that normal business is impossible.
2. *Users often reject systems that are unreliable, unsafe, or insecure* If users find that a system is unreliable or insecure, they will refuse to use it. Furthermore, they may also refuse to buy or use other products from the company that produced the unreliable system. They do not want a repetition of their bad experience with an undependable system.
3. *System failure costs may be enormous* For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system. Failures in systems that control critical infrastructure such as the power network have widespread economic consequences.
4. *Undependable systems may cause information loss* Data is very expensive to collect and maintain; it is usually worth much more than the computer system on which it is processed. The cost of recovering lost or corrupt data is usually very high.

However, a system can be useful without it being very dependable. I don't think that the word processor that I used to write this book is a very dependable system. It sometimes freezes and has to be restarted. Nevertheless, because it is very useful,



Critical systems

Some classes of system are "critical systems" where system failure may result in injury to people, damage to the environment, or extensive economic losses. Examples of critical systems include embedded systems in medical devices, such as an insulin pump (safety-critical), spacecraft navigation systems (mission-critical), and online money transfer systems (business critical).

Critical systems are very expensive to develop. Not only must they be developed so that failures are very rare, but they must also include recovery mechanisms to be used if and when failures occur.

http://software-engineering-book.com/web/critical-systems/

I am prepared to tolerate occasional failure. However, to reflect my lack of trust in the system, I save my work frequently and keep multiple backup copies of it. I compensate for the lack of system dependability by actions that limit the damage that could result from system failure.

Building dependable software is part of the more general process of dependable systems engineering. As I discuss in Section 10.2, software is always part of a broader system. It executes in an operational environment that includes the hardware on which the software executes, the human users of that software and the organizational or business processes where the software is used. When designing a dependable system, you therefore have to consider:

1. *Hardware failure* System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors, because of environmental factors such as dampness or high temperatures, or because components have reached the end of their natural life.
2. *Software failure* System software may fail because of mistakes in its specification, design, or implementation.
3. *Operational failure* Human users may fail to use or operate the system as intended by its designers. As hardware and software have become more reliable, failures in operation are now, perhaps, the largest single cause of system failures.

These failures are often interrelated. A failed hardware component may mean system operators have to cope with an unexpected situation and additional workload. This puts them under stress, and people under stress often make mistakes. These mistakes can cause the software to fail, which means more work for operators, even more stress, and so on.

As a result, it is particularly important that designers of dependable, software-intensive systems take a holistic sociotechnical systems perspective rather than focus on a single aspect of the system such as its software or hardware. If hardware, software, and operational processes are designed separately, without taking into account the potential weaknesses of other parts of the system, then it is more likely that errors will occur at the interfaces between the different parts of the system.

10.1 Dependability properties

All of us are familiar with the problem of computer system failure. For no obvious reason, our computers sometimes crash or go wrong in some way. Programs running on these computers may not operate as expected and occasionally may corrupt the data that is managed by the system. We have learned to live with these failures, but few of us completely trust the personal computers that we normally use.

The dependability of a computer system is a property of the system that reflects its trustworthiness. Trustworthiness here essentially means the degree of confidence a user has that the system will operate as they expect and that the system will not "fail" in normal use. It is not meaningful to express dependability numerically. Rather, relative terms such as "not dependable," "very dependable," and "ultra-dependable" can reflect the degree of trust that we might have in a system.

There are five principal dimensions to dependability, as I have shown in Figure 10.1.

1. *Availability* Informally, the availability of a system is the probability that it will be up and running and able to deliver useful services to users at any given time.
2. *Reliability* Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
3. *Safety* Informally, the safety of a system is a judgment of how likely it is that the system will cause damage to people or its environment.
4. *Security* Informally, the security of a system is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.
5. *Resilience* Informally, the resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks. Resilience is a more recent addition to the set of dependability properties that were originally suggested by Laprie.

The dependability properties shown in Figure 10.1 are complex properties that can be broken down into several simpler properties. For example, security includes "integrity" (ensuring that the systems program and data are not damaged) and "confidentiality" (ensuring that information can only be accessed by people who are authorized). Reliability includes "correctness" (ensuring the system services are as specified), "precision" (ensuring information is delivered at an appropriate level of detail), and "timeliness" (ensuring that information is delivered when it is required).

Of course, not all dependability properties are critical for all systems. For the insulin pump system, introduced in Chapter 1, the most important properties are reliability (it must deliver the correct dose of insulin) and safety (it must never deliver a dangerous dose of insulin). Security is not an issue as the pump does not store confidential information. It is not networked and so cannot be maliciously attacked. For

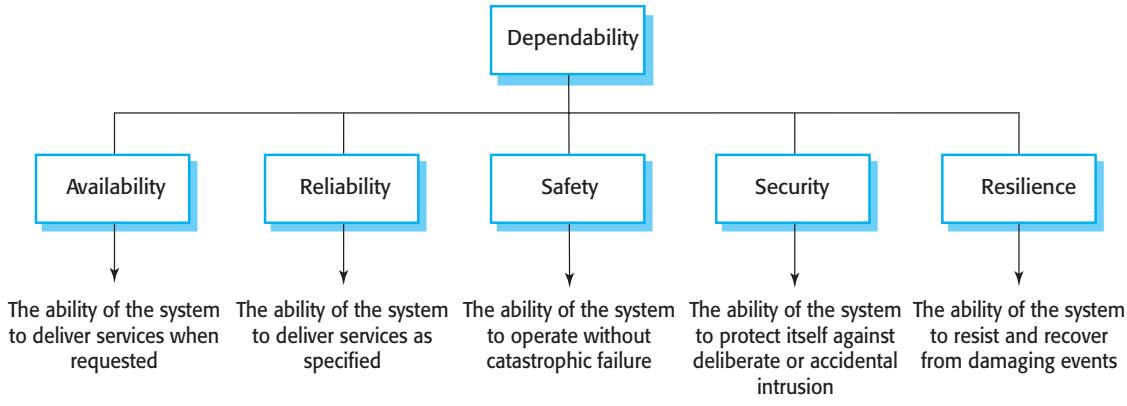


Figure 10.1 Principal dependability properties

the wilderness weather system, availability and reliability are the most important properties because the costs of repair may be very high. For the Mentcare patient information system, security and resilience are particularly important because of the sensitive private data that is maintained and the need for the system to be available for patient consultations.

Other system properties are closely related to these five dependability properties and influence a system's dependability:

1. *Repairability* System failures are inevitable, but the disruption caused by failure can be minimized if the system can be repaired quickly. It must be possible to diagnose the problem, access the component that has failed, and make changes to fix that component. Repairability in software is enhanced when the organization using the system has access to the source code and has the skills to make changes to it. Open-source software makes this easier, but the reuse of components can make it more difficult.

2. *Maintainability* As systems are used, new requirements emerge, and it is important to maintain the value of a system by changing it to include these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements, and where there is a low probability that making changes will introduce new errors into the system.

3. *Error tolerance* This property can be considered as part of usability and reflects the extent to which the system has been designed, so that user input errors are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

The notion of system dependability as an encompassing property was introduced because the dependability properties of availability, security, reliability, safety, and resilience are closely related. Safe system operation usually depends on the system being available and operating reliably. A system may become unreliable because an intruder has corrupted its data. Denial-of-service attacks on a system are intended to

compromise the system's availability. If a system is infected with a virus, you cannot then be confident in its reliability or safety because the virus may change its behavior.

To develop dependable software, you therefore need to ensure that:

1. You avoid the introduction of accidental errors into the system during software specification and development.
2. You design verification and validation processes that are effective in discovering residual errors that affect the dependability of the system.
3. You design the system to be fault tolerant so that it can continue working when things go wrong.
4. You design protection mechanisms that guard against external attacks that can compromise the availability or security of the system.
5. You configure the deployed system and its supporting software correctly for its operating environment.
6. You include system capabilities to recognize external cyberattacks and to resist these attacks.
7. You design systems so that they can quickly recover from system failures and cyberattacks without the loss of critical data.

The need for fault tolerance means that dependable systems have to include redundant code to help them monitor themselves, detect erroneous states, and recover from faults before failures occur. This affects the performance of systems, as additional checking is required each time the system executes. Therefore, designers usually have to trade off performance and dependability. You may need to leave checks out of the system because these slow the system down. However, the consequential risk here is that the system fails because a fault has not been detected.

Building dependable systems is expensive. Increasing the dependability of a system means that you incur extra costs for system design, implementation, and validation. Verification and validation costs are particularly high for systems that must be ultra-dependable such as safety-critical control systems. As well as validating that the system meets its requirements, the validation process may have to prove to an external regulator that the system is safe. For example, aircraft systems have to demonstrate to regulators, such as the Federal Aviation Authority, that the probability of a catastrophic system failure that affects aircraft safety is extremely low.

Figure 10.2 shows the relationship between costs and incremental improvements in dependability. If your software is not very dependable, you can get significant improvements fairly cheaply by using better software engineering. However, if you are already using good practice, the costs of improvement are much greater, and the benefits from that improvement are less.

There is also the problem of testing software to demonstrate that it is dependable. Solving this problem relies on running many tests and looking at the number of failures that occur. As your software becomes more dependable, you see fewer and

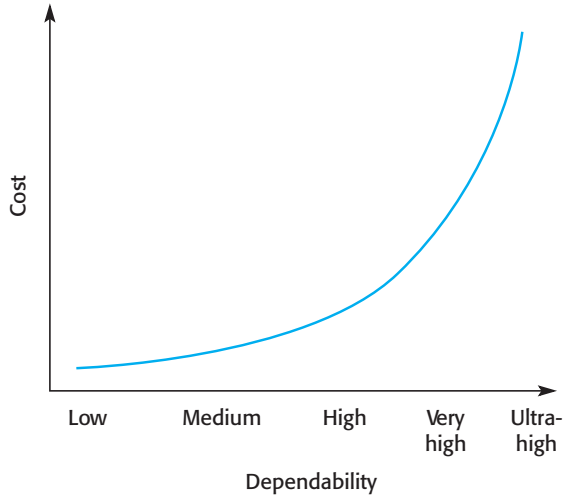


Figure 10.2 Cost/dependability curve

fewer failures. Consequently, more and more tests are needed to try and assess how many problems remain in the software. Testing is a very expensive process, so this can significantly increase the cost of high-dependability systems.

10.2 Sociotechnical systems

In a computer system, the software and the hardware are interdependent. Without hardware, a software system is an abstraction, which is simply a representation of some human knowledge and ideas. Without software, hardware is a set of inert electronic devices. However, if you put them together to form a system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system—it is more than the sum of its parts. Systems have properties that become apparent only when their components are integrated and operate together. Software systems are not isolated systems but are part of more extensive systems that have a human, social, or organizational purpose. Therefore software engineering is not an isolated activity but is an intrinsic part of systems engineering (Chapter 19).

For example, the wilderness weather system software controls the instruments in a weather station. It communicates with other software systems and is a part of wider national and international weather forecasting systems. As well as hardware and software, these systems include processes for forecasting the weather and people who operate the system and analyze its outputs. The system also includes the organizations that depend on the system to help them provide weather forecasts to individuals, government and industry.

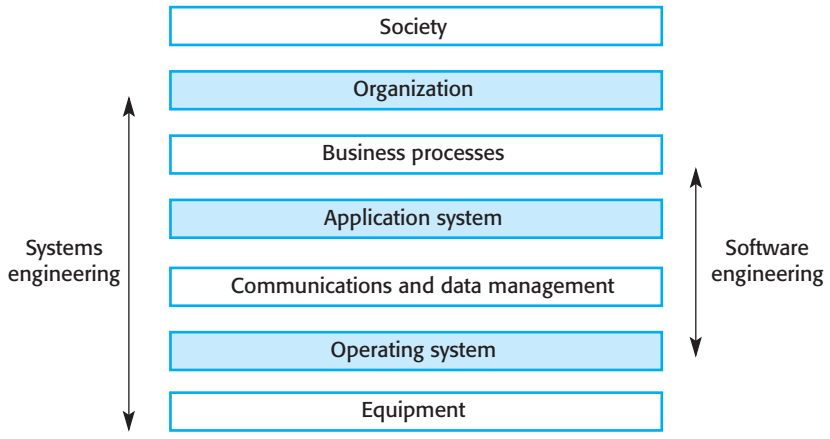


Figure 10.3 The sociotechnical systems stack

These broader systems are called *sociotechnical systems*. They include nontechnical elements such as people, processes, and regulations, as well as technical components such as computers, software, and other equipment. System dependability is influenced by all of the elements in a sociotechnical system—hardware, software, people, and organizations.

Sociotechnical systems are so complex that it is impossible to understand them as a whole. Rather, you have to view them as layers, as shown in Figure 10.3. These layers make up the sociotechnical systems stack:

1. *The equipment layer* is composed of hardware devices, some of which may be computers.
2. *The operating system layer* interacts with the hardware and provides a set of common facilities for higher software layers in the system.
3. *The communications and data management layer* extends the operating system facilities and provides an interface that allows interaction with more extensive functionality, such as access to remote systems and access to a system database. This is sometimes called middleware, as it is in between the application and the operating system.
4. *The application layer* delivers the application-specific functionality that is required. There may be many different application programs in this layer.
5. *The business process layer* includes the organizational business processes, which make use of the software system.
6. *The organizational layer* includes higher-level strategic processes as well as business rules, policies, and norms that should be followed when using the system.
7. *The social layer* refers to the laws and regulations of society that govern the operation of the system.

Notice that there is no separate "software layer." Software of one kind or another is an important part of all of the layers in the sociotechnical system. Equipment is controlled by embedded software; the operating system and applications are software. Business processes, organizations, and society rely on the Internet (software) and other global software systems.

In principle, most interactions should be between neighboring layers in the stack, with each layer hiding the detail of the layer below from the layer above. In practice, however, there can be unexpected interactions between layers, which result in problems for the system as a whole. For example, say there is a change in the law governing access to personal information. This comes from the social layer. It leads to new organizational procedures and changes to the business processes. The application system itself may not be able to provide the required level of privacy, so changes may have to be implemented in the communications and data management layer.

Thinking holistically about systems, rather than simply considering software in isolation, is essential when considering software security and dependability. Software itself is intangible and, even when damaged, is easily and cheaply restored. However, when these software failures ripple through other parts of the system, they affect the software's physical and human environment. Here, the consequences of failure are more significant. Important data may be lost or corrupted. People may have to do extra work to contain or recover from the failure; for example, equipment may be damaged, data may be lost or corrupted, or confidentiality may be breached, with unknown consequences.

You must, therefore, take a system-level view when you are designing software that has to be dependable and secure. You have to take into account the consequences of software failures for other elements in the system. You also need to understand how these other system elements may be the cause of software failure and how they can help to protect against and recover from software failures.

It is important to ensure that, wherever possible, software failure does not lead to overall system failure. You must therefore examine how the software interacts with its immediate environment to ensure that:

1. Software failures are, as far as possible, contained within the enclosing layer of the system stack and do not seriously affect the operation of other layers in the system.
2. You understand how faults and failures in the other layers of the systems stack may affect the software. You may also consider how checks may be built into the software to help detect these failures, and how support can be provided for recovering from failure.

As software is inherently flexible, unexpected system problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this

ghosting. Their solution may be to enhance the image-processing capabilities of the software to remove the ghost images. This may slow down the software so that its performance becomes unacceptable. The problem may then be characterized as a software failure, whereas, in fact, it is a failure in the design process for the system as a whole.

This sort of situation, in which software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. Many so-called software failures are not a consequence of inherent software problems but rather are the result of trying to change the software to accommodate modified system engineering requirements. A good example was the failure of the Denver airport baggage system (Swartz 1996), where the controlling software was expected to deal with limitations of the equipment used.

10.2.1 Regulation and compliance

The general model of economic organization that is now almost universal in the world is that privately owned companies offer goods and services and make a profit on these. We have a competitive environment so that these companies may compete on cost, on quality, on delivery time, and so on. However, to ensure the safety of their citizens, most governments limit the freedom of privately owned companies so that they must follow certain standards to ensure that their products are safe and secure. A company therefore cannot offer products for sale more cheaply because they have reduced their costs by reducing the safety of their products.

Governments have created a set of rules and regulations in different areas that define standards for safety and security. They have also established regulators or regulatory bodies whose job is to ensure that companies offering products in an area comply with these rules. Regulators have wide powers. They can fine companies and even imprison directors if regulations are breached. They may have a licensing role (e.g., in the aviation and nuclear industries) where they must issue a license before a new system may be used. Therefore, aircraft manufacturers have to have a certificate of airworthiness from the regulator in each country where the aircraft is used.

To achieve certification, companies that are developing safety-critical systems have to produce an extensive safety case (discussed in Chapter 13) that shows that rules and regulations have been followed. The case must convince a regulator that the system can operate safely. Developing such a safety case is very costly. It can be as expensive to develop the documentation for certification as it is to develop the system itself.

Regulation and compliance (following the rules) applies to the sociotechnical system as a whole and not simply the software element of that system. For example, a regulator in the nuclear industry is concerned that in the event of overheating, a nuclear reactor will not release radioactivity into the environment. Arguments to convince the regulator that this is the case may be based on software protection systems, the operational process used to monitor the reactor core and the integrity of structures that contain any release of radioactivity.

Each of these elements has to have its own safety case. So, the protection system must have a safety case that demonstrates that the software will operate correctly and shut down the reactor as intended. The overall case must also show that if the software protection system fails, there are alternative safety mechanisms, which do not rely on software, that are invoked.

10.3 Redundancy and diversity

Component failures in any system are inevitable. People make mistakes, undiscovered bugs in software cause undesirable behavior, and hardware burns out. We use a range of strategies to reduce the number of human failures such as replacing hardware components before the end of their predicted lifetime and checking software using static analysis tools. However, we cannot be sure that these will eliminate component failures. We should therefore design systems so that individual component failures do not lead to overall system failure.

Strategies to achieve and enhance dependability rely on both redundancy and diversity. Redundancy means that spare capacity is included in a system that can be used if part of that system fails. Diversity means that redundant components of the system are of different types, thus increasing the chances that they will not fail in exactly the same way.

We use redundancy and diversity to enhance dependability in our everyday lives. Commonly, to secure our homes we use more than one lock (redundancy), and, usually, the locks used are of different types (diversity). This means that if intruders find a way to defeat one of the locks, they have to find a different way of defeating the other locks before they can gain entry. As a matter of routine, we should all back up our computers and so maintain redundant copies of our data. To avoid problems with disk failure, backups should be kept on a separate, diverse, external device.

Software systems that are designed for dependability may include redundant components that provide the same functionality as other system components. These are switched into the system if the primary component fails. If these redundant components are diverse, that is, not the same as other components, a common fault in replicated components will not result in a system failure. Another form of redundancy is the inclusion of checking code, which is not strictly necessary for the system to function. This code can detect some kinds of problems, such as data corruption, before they cause failures. It can invoke recovery mechanisms to correct problems to ensure that the system continues to operate.

In systems for which availability is a critical requirement, redundant servers are normally used. These automatically come into operation if a designated server fails. Sometimes, to ensure that attacks on the system cannot exploit a common vulnerability, these servers may be of different types and may run different operating systems. Using different operating systems is an example of software diversity and



The Ariane 5 explosion

In 1996, the European Space Agency's Ariane 5 rocket exploded 37 seconds after lift-off on its maiden flight. The fault was caused by a software systems failure. There was a backup system but it was not diverse, and so the software in the backup computer failed in exactly the same way. The rocket and its satellite payload were destroyed.

http://software-engineering-book.com/web/ariane/

redundancy, where similar functionality is provided in different ways. (I discuss software diversity in more detail in Chapter 12.)

Diversity and redundancy may also be also used in the design of dependable software development processes. Dependable development processes avoid the introduction of faults into a system. In a dependable process, activities such as software validation do not rely on a single tool or technique. This improves software dependability because it reduces the chances of process failure, where human errors made during the software development process lead to software errors.

For example, validation activities may include program testing, manual program inspections, and static analysis as fault-finding techniques. Any one of these techniques might find faults that are missed by the other methods. Furthermore, different team members may be responsible for the same process activity (e.g., a program inspection). People tackle tasks in different ways depending on their personality, experience, and education, so this kind of redundancy provides a diverse perspective on the system.

However, as I discuss in Chapter 11, using software redundancy and diversity can itself introduce bugs into software. Diversity and redundancy make systems more complex and usually harder to understand. Not only is there more code to write and check, but additional functionality must also be added to the system to detect component failure and to switch control to alternative components. This additional complexity means that it is more likely that programmers will make errors and less likely that people checking the system will find these errors.

Some engineers therefore think that, as software cannot wear out, it is best to avoid software redundancy and diversity. Their view is that the best approach is to design the software to be as simple as possible, with extremely rigorous software verification and validation procedures (Parnas, van Schouwen, and Shu 1990). More can be spent on verification and validation because of the savings that result from not having to develop redundant software components.

Both approaches are used in commercial, safety-critical software systems. For example, the Airbus 340 flight control hardware and software is both diverse and redundant. The flight control software on the Boeing 777 runs on redundant hardware, but each computer runs the same software, which has been very extensively validated. The Boeing 777 flight control system designers have focused on simplicity rather than redundancy. Both of these aircraft are very reliable, so both the diverse and the simple approach to dependability can clearly be successful.



Dependable operational processes

This chapter discusses dependable development processes, but system operational processes are equally important contributors for system dependability. In designing these operational processes, you have to take into account human factors and always bear in mind that people are liable to make mistakes when using a system. A dependable process should be designed to avoid human errors, and, when mistakes are made, the software should detect the mistakes and allow them to be corrected.

http://software-engineering-book.com/web/human-error/

10.4 Dependable processes

Dependable software processes are software processes that are designed to produce dependable software. The rationale for investing in dependable processes is that a good software process is likely to lead to delivered software that contains fewer errors and is therefore less likely to fail in execution. A company using a dependable process can be sure that the process has been properly enacted and documented and that appropriate development techniques have been used for critical systems development. Figure 10.4 shows some of the attributes of dependable software processes.

The evidence that a dependable process has been used is often important in convincing a regulator that the most effective software engineering practice has been applied in developing the software. System developers will normally present a model of the process to a regulator, along with evidence that the process has been followed. The regulator also has to be convinced that the process is used consistently by all of the process participants and that it can be used in different development projects. This means that the process must be explicitly defined and repeatable:

1. An explicitly defined process is one that has a defined process model that is used to drive the software production process. Data must be collected during the process that proves that the development team has followed the process as defined in the process model.
2. A repeatable process is one that does not rely on individual interpretation and judgment. Rather, the process can be repeated across projects and with different team members, irrespective of who is involved in the development. This is particularly important for critical systems, which often have a long development cycle during which there are often significant changes in the development team.

Dependable processes make use of redundancy and diversity to achieve reliability. They often include different activities that have the same aim. For example, program inspections and testing aim to discover errors in a program. The approaches can be used together so that they are likely to find more errors than would be found using one technique on its own.

Process characteristic	Description
Auditable	The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Robust	The process should be able to recover from failures of individual process activities.
Standardized	A comprehensive set of software development standards covering software production and documentation should be available.

Figure 10.4 Attributes of dependable processes

The activities that are used in dependable processes obviously depend on the type of software that is being developed. In general, however, these activities should be geared toward avoiding the introduction of errors into a system, detecting and removing errors, and maintaining information about the process itself.

Examples of activities that might be included in a dependable process include:

1. Requirements reviews to check that the requirements are, as far as possible, complete and consistent.
2. Requirements management to ensure that changes to the requirements are controlled and that the impact of proposed requirements changes is understood by all developers affected by the change.
3. Formal specification, where a mathematical model of the software is created and analyzed. (I discussed the benefits of formal specification in Section 10.5.) Perhaps its most important benefit is that it forces a very detailed analysis of the system requirements. This analysis itself is likely to discover requirements problems that may have been missed in requirements reviews.
4. System modeling, where the software design is explicitly documented as a set of graphical models and the links between the requirements and these models are explicitly documented. If a model-driven engineering approach is used (see Chapter 5), code may be generated automatically from these models.
5. Design and program inspections, where the different descriptions of the system are inspected and checked by different people. A checklist of common design and programming errors may be used to focus the inspection process.
6. Static analysis, where automated checks are carried out on the source code of the program. These look for anomalies that could indicate programming errors or omissions. (I cover static analysis in Chapter 12.)
7. Test planning and management, where a comprehensive set of system tests is designed. The testing process has to be carefully managed to demonstrate that these tests provide coverage of the system requirements and have been correctly applied in the testing process.

As well as process activities that focus on system development and testing, there must also be well-defined quality management and change management processes. While the specific activities in a dependable process may vary from one company to another, the need for effective quality and change management is universal.

Quality management processes (covered in Chapter 24) establish a set of process and product standards. They also include activities that capture process information to demonstrate that these standards have been followed. For example, there may be a standard defined for carrying out program inspections. The inspection team leader is responsible for documenting the process to show that the inspection standard has been followed.

Change management, discussed in Chapter 25, is concerned with managing changes to a system, ensuring that accepted changes are actually implemented, and confirming that planned releases of the software include the planned changes. One common problem with software is that the wrong components are included in a system build. This can lead to a situation where an executing system includes components that have not been checked during the development process. Configuration management procedures must be defined as part of the change management process to ensure that this does not happen.

As agile methods have become increasingly used, researchers and practitioners have thought carefully about how to use agile approaches in dependable software development (Trimble 2012). Most companies that develop critical software systems have based their development on plan-based processes and have been reluctant to make radical changes to their development process. However, they recognize the value of agile approaches and are exploring how their dependable development processes can be more agile.

As dependable software often requires certification, both process and product documentation have to be produced. Up-front requirements analysis is also essential to discover possible requirements and requirements conflicts that may compromise the safety and security of the system. Formal change analysis is essential to assess the effect of changes on the safety and integrity of the system. These requirements conflict with the general approach in agile development of co-development of the requirements and the system and minimizing documentation.

Although most agile development uses an informal, undocumented process, this is not a fundamental requirement of agility. An agile process may be defined that incorporates techniques such as iterative development, test-first development and user involvement in the development team. As long as the team follows that process and documents their actions, agile techniques can be used. To support this notion, various proposals of modified agile methods have been made that incorporate the requirements of dependable systems engineering (Douglass 2013). These combine the most appropriate techniques from agile and plan-based development.

10.5 Formal methods and dependability

For more than 30 years, researchers have advocated the use of formal methods of software development. Formal methods are mathematical approaches to software development where you define a formal model of the software. You may then formally analyze this model to search for errors and inconsistencies, prove that a program

is consistent with this model, or you may apply a series of correctness-preserving transformations to the model to generate a program. Abrial (Abrial 2009) claims that the use of formal methods can lead to "faultless systems," although he is careful to limit what he means in this claim.

In an excellent survey, Woodcock et al. (Woodcock et al. 2009) discuss industrial applications where formal methods have been successfully applied. These include train control systems (Badeau and Amelot 2005), cash card systems (Hall and Chapman 2002), and flight control systems (Miller et al. 2005). Formal methods are the basis of tools used in static verification, such as the driver verification system used by Microsoft (Ball et al. 2006).

Using a mathematically formal approach to software development was proposed at an early stage in the development of computer science. The idea was that a formal specification and a program could be developed independently. A mathematical proof could then be developed to show that the program and its specification were consistent. Initially, proofs were developed manually but this was a long and expensive process. It quickly became clear that manual proofs could only be developed for very small systems. Program proving is now supported by large-scale automated theorem proving software, which has meant that larger systems can be proved. However, developing the proof obligations for theorem provers is a difficult and specialized task, so formal verification is not widely used.

An alternative approach, which avoids a separate proof activity, is refinement-based development. Here, a formal specification of a system is refined through a series of correctness-preserving transformations to generate the software. Because these are trusted transformations, you can be confident that the generated program is consistent with its formal specification. This was the approach used in the software development for the Paris Metro system (Badeau and Amelot 2005). It used a language called B (Abrial 2010), which was designed to support specification refinement.

Formal methods based on model-checking (Jhala and Majumdar 2009) have been used in a number of systems (Bochot et al. 2009; Calinescu and Kwiatkowska 2009). These systems rely on constructing or generating a formal state model of a system and using a model-checker to check that properties of the model, such as safety properties, always hold. The model-checking program exhaustively analyzes the specification and either reports that the system property is satisfied by the model or presents an example that shows it is not satisfied. If a model can be automatically or systematically generated from a program, this means that bugs in the program can be uncovered. (I cover model checking in safety-critical systems in Chapter 12.)

Formal methods for software engineering are effective for discovering or avoiding two classes of error in software representations:

1. *Specification and design errors and omissions.* The process of developing and analyzing a formal model of the software may reveal errors and omissions in the software requirements. If the model is generated automatically or systematically from source code, analysis using model checking can discover undesirable states that may occur, such as deadlock in a concurrent system.



Formal specification techniques

Formal system specifications may be expressed using two fundamental approaches, either as models of the system interfaces (algebraic specifications) or as models of the system state. An extra web chapter on this topic shows examples of both of these approaches. The chapter includes a formal specification of part of the insulin pump system.

http://software-engineering-book.com/web/formal-methods/ (web chapter)

2. *Inconsistencies between a specification and a program.* If a refinement method is used, mistakes made by developers that make the software inconsistent with the specification are avoided. Program proving discovers inconsistencies between a program and its specification.

The starting point for all formal methods is a mathematical system model, which acts as a system specification. To create this model, you translate the system's user requirements, which are expressed in natural language, diagrams, and tables, into a mathematical language that has formally defined semantics. The formal specification is an unambiguous description of what the system should do.

Formal specifications are the most precise way of specifying systems, and so reduce the scope for misunderstanding. Many supporters of formal methods believe that creating formal specification, even without refinement or program proof, is worthwhile. Constructing a formal specification forces a detailed analysis of the requirements and this is an effective way of discovering requirements problems. In a natural language specification, errors can be concealed by the imprecision of the language. This is not the case if the system is formally specified.

The advantages of developing a formal specification and using it in a formal development process are:

1. As you develop a formal specification in detail, you develop a deep and detailed understanding of the system requirements. Requirements problems that are discovered early are usually much cheaper to correct than if they are found at later stages in the development process.
2. As the specification is expressed in a language with formally defined semantics, you can analyze it automatically to discover inconsistencies and incompleteness.
3. If you use a method such as the B method, you can transform the formal specification into a program through a sequence of correctness-preserving transformations. The resulting program is therefore guaranteed to meet its specification.
4. Program testing costs may be reduced because you have verified the program against its specification. For example, in the development of the software for the Paris Metro systems, the use of refinement meant that there was no need for software component testing and only system testing was required.

Woodcock's survey (Woodcock et al. 2009) found that users of formal methods reported fewer errors in the delivered software. Neither the costs nor the time needed for software development were higher than in comparable development projects. There were significant benefits in using formal approaches in safety critical systems that required regulator certification. The documentation produced was an important part of the safety case (see Chapter 12) for the system.

In spite of these advantages, formal methods have had limited impact on practical software development, even for critical systems. Woodcock reports on 62 projects over 25 years that used formal methods. Even if we allow for projects that used these techniques but did not report their use, this is a tiny fraction of the total number of critical systems developed in that time. Industry has been reluctant to adopt formal methods for a number of reasons:

1. Problem owners and domain experts cannot understand a formal specification, so they cannot check that it accurately represents their requirements. Software engineers, who understand the formal specification, may not understand the application domain, so they too cannot be sure that the formal specification is an accurate reflection of the system requirements.
2. It is fairly easy to quantify the costs of creating a formal specification, but more difficult to estimate the possible cost savings that will result from its use. As a result, managers are unwilling to take the risk of adopting formal methods. They are unconvinced by reports of success as, by and large, these came from atypical projects where the developers were keen advocates of a formal approach.
3. Most software engineers have not been trained to use formal specification languages. Hence, they are reluctant to propose their use in development processes.
4. It is difficult to scale current formal methods up to very large systems. When formal methods are used, it is mostly for specifying critical kernel software rather than complete systems.
5. Tool support for formal methods is limited, and the available tools are often open source and difficult to use. The market is too small for commercial tool providers.
6. Formal methods are not compatible with agile development where programs are developed incrementally. This is not a major issue, however, as most critical systems are still developed using a plan-based approach.

Parnas, an early advocate of formal development, has criticized current formal methods and claims that these have started from a fundamentally wrong premise (Parnas 2010). He believes that these methods will not gain acceptance until they are radically simplified, which will require a different type of mathematics as a basis. My own view is that even this will not mean that formal methods are routinely adopted for critical systems engineering unless it can be clearly demonstrated that their adoption and use is cost-effective, compared to other software engineering methods.

KEY POINTS

- System dependability is important because failure of critical computer systems can lead to large economic losses, serious information loss, physical damage or threats to human life.
- The dependability of a computer system is a system property that reflects the user's degree of trust in the system. The most important dimensions of dependability are availability, reliability, safety, security, and resilience.
- Sociotechnical systems include computer hardware, software, and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- The use of a dependable, repeatable process is essential if faults in a system are to be minimized. The process should include verification and validation activities at all stages, from requirements definition through to system implementation.
- The use of redundancy and diversity in hardware, software processes, and software systems is essential to the development of dependable systems.
- Formal methods, where a formal model of a system is used as a basis for development, help reduce the number of specification and implementation errors in a system. However, formal methods have had a limited take-up in industry because of concerns about the cost-effectiveness of this approach.

FURTHER READING

"Basic Concepts and Taxonomy of Dependable and Secure Computing." This work presents a thorough discussion of dependability concepts written by some of the pioneers in the field who were responsible for developing these ideas. (A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr., *IEEE Transactions on Dependable and Secure Computing*, 1 (1), 2004) http://dx.doi.org/10.1109/TDSC.2004.2

Formal Methods: Practice and Experience. An excellent survey of the use of formal methods in industry, along with a description of some projects that have used formal methods. The authors present a realistic summary of the barriers to the use of these methods. (J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. *Computing Surveys,* 41 (1) January 2009) http://dx.doi.org/10.1145/1592434.1592436

The LSCITS Socio-technical Systems Handbook. This handbook introduces sociotechnical systems in an accessible way and provides access to more detailed papers on sociotechnical topics. (2012) http://archive.cs.st-andrews.ac.uk/STSE-Handbook/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/critical-systems/

EXERCISES

10.1. Suggest six reasons why software dependability is important in most sociotechnical systems.

10.2. Explain with an example why resilience to cyber attacks is a very important characteristic of system dependability.

10.3. Using an example, explain why it is important when developing dependable systems to consider these as sociotechnical systems and not simply as technical software and hardware systems.

10.4. Give two examples of government functions that are supported by complex sociotechnical systems and explain why, in the foreseeable future, these functions cannot be completely automated.

10.5. Explain the difference between redundancy and diversity.

10.6. Explain why it is reasonable to assume that the use of dependable processes will lead to the creation of dependable software.

10.7. Give two examples of diverse, redundant activities that might be incorporated into dependable processes.

10.8. Give two reasons why different versions of a system based on software diversity may fail in a similar way.

10.9. You are an engineer in charge of the development of a small, safety-critical train control system, which must be demonstrably safe and secure. You suggest that formal methods should be used in the development of this system, but your manager is skeptical of this approach. Write a report highlighting the benefits of formal methods and presenting a case for their use in this project.

10.10. It has been suggested that the need for regulation inhibits innovation and that regulators force the use of older methods of systems development that have been used on other systems. Discuss whether or not you think this is true and the desirability of regulators imposing their views on what methods should be used.

REFERENCES

Abrial, J. R. 2009. "Faultless Systems: Yes We Can." *IEEE Computer* 42 (9): 30–36. doi:10.1109/MC.2009.283.

———. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Avizienis, A., J. C. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. on Dependable and Secure Computing* 1 (1): 11–33. doi:10.1109/TDSC.2004.2.

Badeau, F., and A. Amelot. 2005. "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL." In *Proc. ZB 2005: Formal Specification and Development in Z and B*. Guildford, UK: Springer. doi:10.1007/11415787_20.

Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. 2006. "Thorough Static Analysis of Device Drivers." In *Proc. EuroSys 2006*. Leuven, Belgium. doi:10.1145/1218063.1217943.

Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Calinescu, R. C., and M. Z. Kwiatkowska. 2009. "Using Quantitative Analysis to Implement Autonomic IT Systems." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 100–10. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE.2009.5070512.

Douglass, B. 2013. "Agile Analysis Practices for Safety-Critical Software Development." http://www.ibm.com/developerworks/rational/library/agile-analysis-practices-safety-critical-development/.

Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18–25.doi:10.1109/52.976937.

Jhala, R., and R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4), Article 21. doi:1145/1592434.1592438.

Miller, S. P., E. A. Anderson, L. G. Wagner, M. W. Whalen, and M. P. E. Heimdahl. 2005. "Formal Verification of Flight Critical Software." In *Proc. AIAA Guidance, Navigation and Control Conference*. San Francisco. doi:10.2514/6.2005-6431.

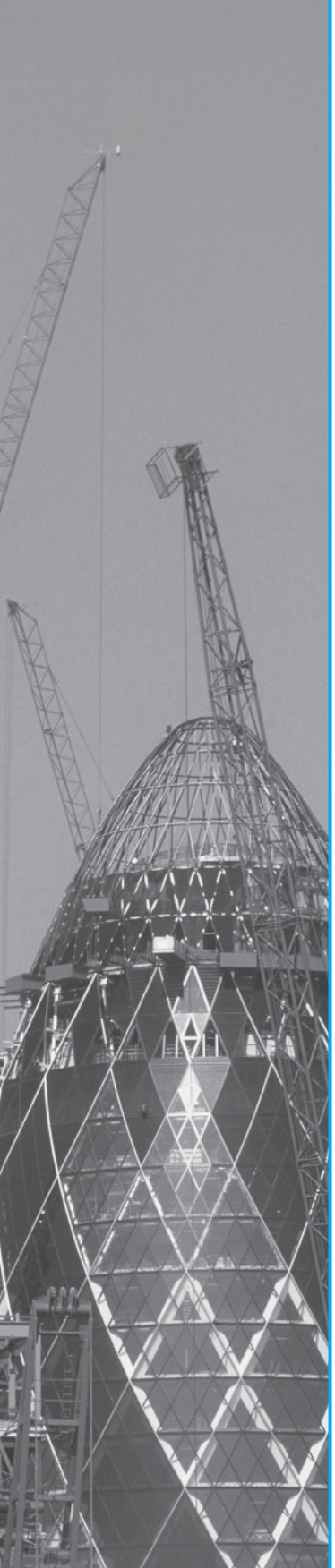
Parnas, D. 2010. "Really Rethinking Formal Methods." *IEEE Computer* 43 (1): 28–34. doi:10.1109/MC.2010.22.

Parnas, D., J. van Schouwen, and P. K. Shu. 1990. "Evaluation of Safety-Critical Software." *Comm. ACM* 33 (6): 636–651. doi:10.1145/78973.78974.

Swartz, A. J. 1996. "Airport 95: Automated Baggage System?" *ACM Software Engineering Notes* 21 (2): 79–83. doi:10.1145/227531.227544.

Trimble, J. 2012. "Agile Development Methods for Space Operations." In *SpaceOps 2012*. Stockholm. doi:10.2514/6.2012-1264554.

Woodcock, J., P. G. Larsen, J. Bicarregui, and J. Fitzgerald. 2009. "Formal Methods: Practice and Experience." *Computing Surveys* 41 (4): 1–36. doi:10.1145/1592434.1592436.



11 Reliability engineering

Objectives

The objective of this chapter is to explain how software reliability may be specified, implemented, and measured. When you have read this chapter, you will:

- understand the distinction between software reliability and software availability;
- have been introduced to metrics for reliability specification and how these are used to specify measurable reliability requirements;
- understand how different architectural styles may be used to implement reliable, fault-tolerant systems architectures;
- know about good programming practice for reliable software engineering;
- understand how the reliability of a software system may be measured using statistical testing.

Contents

11.1 Availability and reliability
11.2 Reliability requirements
11.3 Fault-tolerant architectures
11.4 Programming for reliability
11.5 Reliability measurement

Our dependence on software systems for almost all aspects of our business and personal lives means that we expect that software to be available when we need it. This may be early in the morning or late at night, at weekends or during holidays—the software must run all day, every day of the year. We expect that software will operate without crashes and failures and will preserve our data and personal information. We need to be able to trust the software that we use, which means that the software must be reliable.

The use of software engineering techniques, better programming languages, and effective quality management has led to significant improvements in software reliability over the past 20 years. Nevertheless, system failures still occur that affect the system's availability or lead to incorrect results being produced. In situations where software has a particularly critical role—perhaps in an aircraft or as part of the national critical infrastructure—special reliability engineering techniques may be used to achieve the high levels of reliability and availability that are required.

Unfortunately, it is easy to get confused when talking about system reliability, with different people meaning different things when they talk about system faults and failures. Brian Randell, a pioneer researcher in software reliability, defined a fault–error–failure model (Randell 2000) based on the notion that human errors cause faults; faults lead to errors, and errors lead to system failures. He defined these terms precisely:

1. *Human error or mistake* Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
2. *System fault* A characteristic of a software system that can lead to a system error. The fault in the above example is the inclusion of code to add 1 to a variable called Transmission_time, without a check to see if the value of Transmission_time is greater than or equal to 23.00.
3. *System error* An erroneous system state during execution that can lead to system behavior that is unexpected by system users. In this example, the value of the variable Transmission_time is set incorrectly to 24.XX rather than 00.XX when the faulty code is executed.
4. *System failure* An event that occurs at some point in time when the system does not deliver a service as expected by its users. In this case, no weather data is transmitted because the time is invalid.

System faults do not necessarily result in system errors, and system errors do not necessarily result in system failures:

1. Not all code in a program is executed. The code that includes a fault (e.g., the failure to initialize a variable) may never be executed because of the way that the software is used.

2. Errors are transient. A state variable may have an incorrect value caused by the execution of faulty code. However, before this is accessed and causes a system failure, some other system input may be processed that resets the state to a valid value. The wrong value has no practical effect.
3. The system may include fault detection and protection mechanisms. These ensure that the erroneous behavior is discovered and corrected before the system services are affected.

Another reason why the faults in a system may not lead to system failures is that users adapt their behavior to avoid using inputs that they know cause program failures. Experienced users "work around" software features that they have found to be unreliable. For example, I avoid some features, such as automatic numbering, in the word processing system that I use because my experience is that it often goes wrong. Repairing faults in such unused features makes no practical difference to the system reliability.

The distinction between faults, errors, and failures leads to three complementary approaches that are used to improve the reliability of a system:

1. *Fault avoidance* The software design and implementation process should use approaches to software development that help avoid design and programming errors and so minimize the number of faults introduced into the system. Fewer faults means less chance of runtime failures. Fault-avoidance techniques include the use of strongly typed programming language to allow extensive compiler checking and minimizing the use of error-prone programming language constructs, such as pointers.
2. *Fault detection and correction* Verification and validation processes are designed to discover and remove faults in a program, before it is deployed for operational use. Critical systems require extensive verification and validation to discover as many faults as possible before deployment and to convince the system stakeholders and regulators that the system is dependable. Systematic testing and debugging and static analysis are examples of fault-detection techniques.
3. *Fault tolerance* The system is designed so that faults or unexpected system behavior during execution are detected at runtime and are managed in such a way that system failure does not occur. Simple approaches to fault tolerance based on built-in runtime checking may be included in all systems. More specialized fault-tolerance techniques, such as the use of fault-tolerant system architectures, discussed in Section 11.3, may be used when a very high level of system availability and reliability is required.

Unfortunately, applying fault-avoidance, fault-detection, and fault-tolerance techniques is not always cost-effective. The cost of finding and removing the remaining faults in a software system rises exponentially as program faults are discovered and removed (Figure 11.1). As the software becomes more reliable, you need to spend more and more time and effort to find fewer and fewer faults. At some stage, even for critical systems, the costs of this additional effort become unjustifiable.

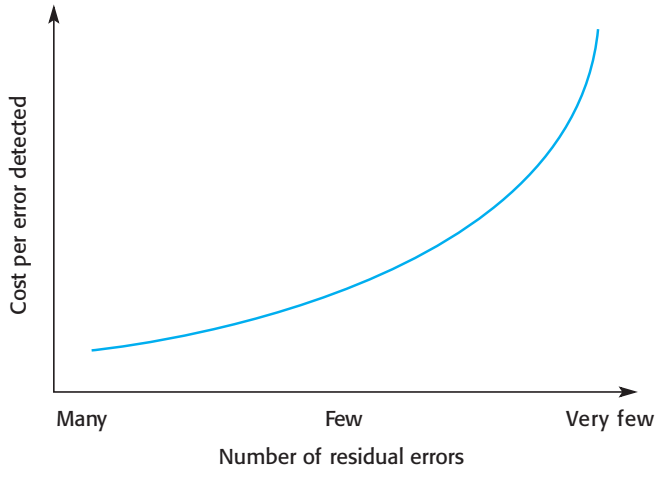


Figure 11.1 The increasing costs of residual fault removal

As a result, software companies accept that their software will always contain some residual faults. The level of faults depends on the type of system. Software products have a relatively high level of faults, whereas critical systems usually have a much lower fault density.

The rationale for accepting faults is that, if and when the system fails, it is cheaper to pay for the consequences of failure than it would be to discover and remove the faults before system delivery. However, the decision to release faulty software is not simply an economic one. The social and political acceptability of system failure must also be taken into account.

11.1 Availability and reliability

In Chapter 10, I introduced the concepts of system reliability and system availability. If we think of systems as delivering some kind of service (to deliver cash, control brakes, or connect phone calls, for example), then the availability of that service is whether or not that service is up and running and its reliability is whether or not that service delivers correct results. Availability and reliability can both be expressed as probabilities. If the availability is 0.999, this means that, over some time period, the system is available for 99.9% of that time. If, on average, 2 inputs in every 1000 result in failures, then the reliability, expressed as a rate of occurrence of failure, is 0.002.

More precise definitions of availability and reliability are:

1. *Reliability* The probability of failure-free operation over a specified time, in a given environment, for a specific purpose.
2. *Availability* The probability that a system, at a point in time, will be operational and able to deliver the requested services.

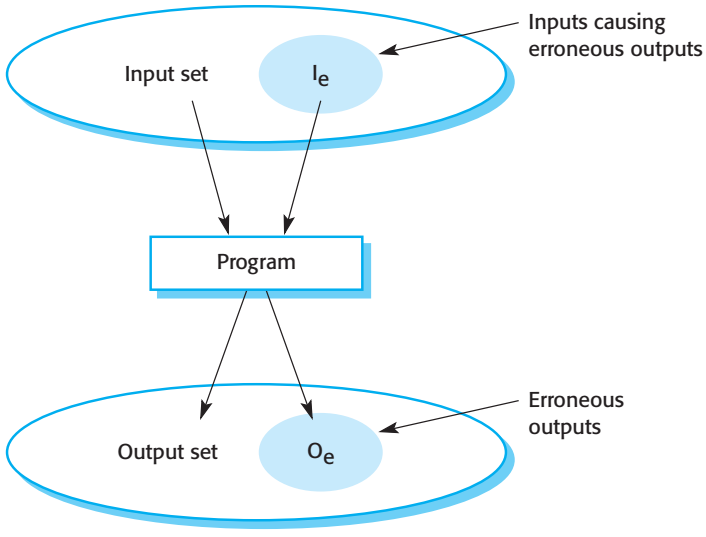


Figure 11.2 A system as an input/output mapping

System reliability is not an absolute value—it depends on where and how that system is used. For example, let's say that you measure the reliability of an application in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system. If you then measure the reliability of the same system in a university environment, then the reliability may be quite different. Here, students may explore the boundaries of the system and use it in unexpected ways. This may result in system failures that did not occur in the more constrained office environment. Therefore, the perceptions of the system's reliability in each of these environments are different.

The above definition of reliability is based on the idea of failure-free operation, where failures are external events that affect the users of a system. But what constitutes "failure"? A technical definition of failure is behavior that does not conform to the system's specification. However, there are two problems with this definition:

1. Software specifications are often incomplete or incorrect, and it is left to software engineers to interpret how the system should behave. As they are not domain experts, they may not implement the behavior that users expect. The software may behave as specified, but, for users, it is still failing.
2. No one except system developers reads software specification documents. Users may therefore anticipate that the software should behave in one way when the specification says something completely different.

Failure is therefore not something that can be objectively defined. Rather, it is a judgment made by users of a system. This is one reason why users do not all have the same impression of a system's reliability.

To understand why reliability is different in different environments, we need to think about a system as an input/output mapping. Figure 11.2 shows a software system that

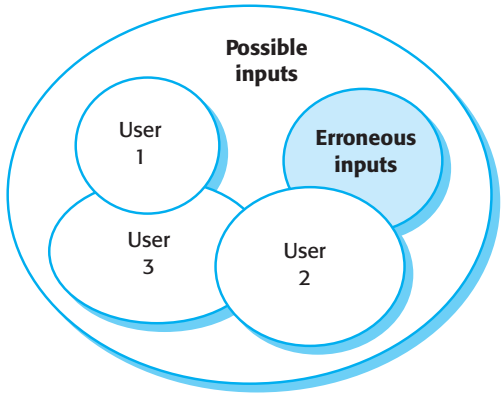


Figure 11.3 Software usage patterns

links a set of inputs with a set of outputs. Given an input or input sequence, the program responds by producing a corresponding output. For example, given an input of a URL, a web browser produces an output that is the display of the requested web page.

Most inputs do not lead to system failure. However, some inputs or input combinations, shown in the shaded ellipse I_e in Figure 11.2, cause system failures or erroneous outputs to be generated. The program's reliability depends on the number of system inputs that are members of the set of inputs that lead to an erroneous output—in other words, the set of inputs that cause faulty code to be executed and system errors to occur. If inputs in the set I_e are executed by frequently used parts of the system, then failures will be frequent. However, if the inputs in I_e are executed by code that is rarely used, then users will hardly ever see failures.

Faults that affect the reliability of the system for one user may never show up under someone else's mode of working. In Figure 11.3, the set of erroneous inputs corresponds to the ellipse labeled I_e in Figure 11.2. The set of inputs produced by User 2 intersects with this erroneous input set. User 2 will therefore experience some system failures. User 1 and User 3, however, never use inputs from the erroneous set. For them, the software will always appear to be reliable.

The availability of a system does not just depend on the number of system failures, but also on the time needed to repair the faults that have caused the failure. Therefore, if system A fails once a year and system B fails once a month, then A is apparently more reliable then B. However, assume that system A takes 6 hours to restart after a failure, whereas system B takes 5 minutes to restart. The availability of system B over the year (60 minutes of down time) is much better than that of system A (360 minutes of downtime).

Furthermore, the disruption caused by unavailable systems is not reflected in the simple availability metric that specifies the percentage of time that the system is available. The time when the system fails is also important. If a system is unavailable for an hour each day between 3 am and 4 am, this may not affect many users. However, if the same system is unavailable for 10 minutes during the working day, system unavailability has a much greater effect on users.

Reliability and availability are closely related, but sometimes one is more important than the other. If users expect continuous service from a system, then the system

has a high-availability requirement. It must be available whenever a demand is made. However, if a system can recover quickly from failures without loss of user data, then these failures may not significantly affect system users.

A telephone exchange switch that routes phone calls is an example of a system where availability is more important than reliability. Users expect to be able to make a call when they pick up a phone or activate a phone app, so the system has high-availability requirements. If a system fault occurs while a connection is being set up, this is often quickly recoverable. Exchange or base station switches can reset the system and retry the connection attempt. This can be done quickly, and phone users may not even notice that a failure has occurred. Furthermore, even if a call is interrupted, the consequences are usually not serious. Users simply reconnect if this happens.

11.2 Reliability requirements

In September 1993, a plane landed at Warsaw Airport in Poland during a thunderstorm. For 9 seconds after landing, the brakes on the computer-controlled braking system did not work. The braking system had not recognized that the plane had landed and assumed that the aircraft was still airborne. A safety feature on the aircraft had stopped the deployment of the reverse thrust system, which slows down the aircraft, because reverse thrust is catastrophic if the plane is in the air. The plane ran off the end of the runway, hit an earth bank, and caught fire.

The inquiry into the accident showed that the braking system software had operated according to its specification. There were no errors in the control system. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This incident shows that system dependability does not just depend on good engineering. It also requires attention to detail when the system requirements are derived and the specification of software requirements that are geared to ensuring the dependability of a system. Those dependability requirements are of two types:

1. *Functional requirements*, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.
2. *Non-functional requirements*, which define the required reliability and availability of the system.

As I discussed in Chapter 10, the overall reliability of a system depends on the hardware reliability, the software reliability, and the reliability of the system operators. The system software has to take this requirement into account. As well as including requirements that compensate for software failure, there may also be related reliability requirements to help detect and recover from hardware failures and operator errors.

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period—roughly, one minute per week.

Figure 11.4 Availability specification

11.2.1 Reliability metrics

Reliability can be specified as a probability that a system failure will occur when a system is in use within a specified operating environment. If you are willing to accept, for example, that 1 in any 1000 transactions may fail, then you can specify the failure probability as 0.001. This doesn't mean that there will be exactly 1 failure in every 1000 transactions. It means that if you observe *N* thousand transactions, the number of failures that you observe should be about *N*.

Three metrics may be used to specify reliability and availability:

1. *Probability of failure on demand* (*POFOD*) If you use this metric, you define the probability that a demand for service from a system will result in a system failure. So, POFOD = 0.001 means that there is a 1/1000 chance that a failure will occur when a demand is made.
2. *Rate of occurrence of failures* (*ROCOF*) This metric sets out the probable number of system failures that are likely to be observed relative to a certain time period (e.g., an hour), or to the number of system executions. In the example above, the ROCOF is 1/1000. The reciprocal of ROCOF is the *mean time to failure (MTTF)*, which is sometimes used as a reliability metric. MTTF is the average number of time units between observed system failures. A ROCOF of two failures per hour implies that the mean time to failure is 30 minutes.
3. *Availability (AVAIL)* AVAIL is the probability that a system will be operational when a demand is made for service. Therefore, an availability of 0.9999 means that, on average, the system will be available for 99.99% of the operating time. Figure 11.4 shows what different levels of availability mean in practice.

POFOD should be used in situations where a failure on demand can lead to a serious system failure. This applies irrespective of the frequency of the demands. For example, a protection system that monitors a chemical reactor and shuts down the reaction if it is overheating should have its reliability specified using POFOD. Generally, demands on a protection system are infrequent as the system is a last line of defense, after all other recovery strategies have failed. Therefore a POFOD of 0.001 (1 failure in 1000 demands)

might seem to be risky. However, if there are only two or three demands on the system in its entire lifetime, then the system is unlikely to ever fail.

ROCOF should be used when demands on systems are made regularly rather than intermittently. For example, in a system that handles a large number of transactions, you may specify a ROCOF of 10 failures per day. This means that you are willing to accept that an average of 10 transactions per day will not complete successfully and will have to be canceled and resubmitted. Alternatively, you may specify ROCOF as the number of failures per 1000 transactions.

If the absolute time between failures is important, you may specify the reliability as the mean time to failures (MTTF). For example, if you are specifying the required reliability for a system with long transactions (such as a computer-aided design system), you should use this metric. The MTTF should be much longer than the average time that a user works on his or her models without saving the user's results. This means that users are unlikely to lose work through a system failure in any one session.

11.2.2 Non-functional reliability requirements

Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF, or AVAIL) described in the previous section. Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems. However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Quantitative reliability specification is useful in a number of ways:

1. The process of deciding the required level of the reliability helps to clarify what stakeholders really need. It helps stakeholders understand that there are different types of system failure, and it makes clear to them that high levels of reliability are expensive to achieve.
2. It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
3. It is a means of assessing different design strategies intended to improve the reliability of a system. You can make a judgment about how each strategy might lead to the required levels of reliability.
4. If a regulator has to approve a system before it goes into service (e.g., all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

To avoid incurring excessive and unnecessary costs, it is important that you specify the reliability that you really need rather than simply choose a very high level of reliability for the whole system. You may have different requirements for different



Overspecification of reliability

Overspecification of reliability means defining a level of required reliability that is higher than really necessary for the practical operation of the software. Overspecification of reliability increases development costs disproportionately. The reason for this is that the costs of reducing faults and verifying reliability increase exponentially as reliability increases

http://software-engineering-book.com/web/over-specifying-reliability/

parts of the system if some parts are more critical than others. You should follow these three guidelines when specifying reliability requirements:

1. Specify the availability and reliability requirements for different types of failure. There should be a lower probability of high-cost failures than failures that don't have serious consequences.
2. Specify the availability and reliability requirements for different types of system service. Critical system services should have the highest reliability but you may be willing to tolerate more failures in less critical services. You may decide that it is only cost-effective to use quantitative reliability specification for the most critical system services.
3. Think about whether high reliability is really required. For example, you may use error-detection mechanisms to check the outputs of a system and have error-correction processes in place to correct errors. There may then be no need for a high level of reliability in the system that generates the outputs as errors can be detected and corrected.

To illustrate these guidelines, think about the reliability and availability requirements for a bank ATM system that dispenses cash and provides other services to customers. Banks have two concerns with such systems:

1. To ensure that they carry out customer services as requested and that they properly record customer transactions in the account database.
2. To ensure that these systems are available for use when required.

Banks have many years of experience with identifying and correcting incorrect account transactions. They use accounting methods to detect when things have gone wrong. Most transactions that fail can simply be canceled, resulting in no loss to the bank and minor customer inconvenience. Banks that run ATM networks therefore accept that ATM failures may mean that a small number of transactions are incorrect, but they think it more cost-effective to fix these errors later rather than incur high costs in avoiding faulty transactions. Therefore, the absolute reliability required of an ATM may be relatively low. Several failures per day may be acceptable.

For a bank (and for the bank's customers), the availability of the ATM network is more important than whether or not individual ATM transactions fail. Lack of availability means increased demand on counter services, customer dissatisfaction, engineering costs to repair the network, and so on. Therefore, for transaction-based systems such as banking and e-commerce systems, the focus of reliability specification is usually on specifying the availability of the system.

To specify the availability of an ATM network, you should identify the system services and specify the required availability for each of these services, notably:

- the customer account database service; and
- the individual services provided by an ATM such as "withdraw cash" and "provide account information."

The database service is the most critical as failure of this service means that all of the ATMs in the network are out of action. Therefore, you should specify this service to have a high level of availability. In this case, an acceptable figure for database availability (ignoring issues such as scheduled maintenance and upgrades) would probably be around 0.9999, between 7 am and 11 pm. This means a downtime of less than 1 minute per week.

For an individual ATM, the overall availability depends on mechanical reliability and the fact that it can run out of cash. Software issues are probably less significant than these factors. Therefore, a lower level of software availability for the ATM software is acceptable. The overall availability of the ATM software might therefore be specified as 0.999, which means that a machine might be unavailable for between 1 and 2 minutes each day. This allows for the ATM software to be restarted in the event of a problem.

The reliability of control systems is usually specified in terms of the probability that the system will fail when a demand is made (POFOD). Consider the reliability requirements for the control software in the insulin pump, introduced in Chapter 1. This system delivers insulin a number of times per day and monitors the user's blood glucose several times per hour.

There are two possible types of failure in the insulin pump:

1. *Transient software failures*, which can be repaired by user actions such as resetting or recalibrating the machine. For these types of failure, a relatively low value of POFOD (say 0.002) may be acceptable. This means that one failure may occur in every 500 demands made on the machine. This is approximately once every 3.5 days, because the blood sugar is checked about 5 times per hour.
2. *Permanent software failures*, which require the software to be reinstalled by the manufacturer. The probability of this type of failure should be much lower. Roughly once a year is the minimum figure, so POFOD should be no more than 0.00002.

RR1: A predefined range for all operator inputs shall be defined, and the system shall check that all operator inputs fall within this predefined range. (Checking)
RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)
RR3: *N*-version programming shall be used to implement the braking control system. (Redundancy)
RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Figure 11.5 Examples of functional reliability requirements

Failure to deliver insulin does not have immediate safety implications, so commercial factors rather than safety factors govern the level of reliability required. Service costs are high because users need fast repair and replacement. It is in the manufacturer's interest to limit the number of permanent failures that require repair.

11.2.3 Functional reliability specification

To achieve a high level of reliability and availability in a software-intensive system, you use a combination of fault-avoidance, fault-detection, and fault-tolerance techniques. This means that functional reliability requirements have to be generated which specify how the system should provide fault avoidance, detection, and tolerance.

These functional reliability requirements should specify the faults to be detected and the actions to be taken to ensure that these faults do not lead to system failures. Functional reliability specification, therefore, involves analyzing the non-functional requirements (if these have been specified), assessing the risks to reliability and specifying system functionality to address these risks.

There are four types of functional reliability requirements:

1. *Checking requirements* These requirements identify checks on inputs to the system to ensure that incorrect or out-of-range inputs are detected before they are processed by the system.
2. *Recovery requirements* These requirements are geared to helping the system recover after a failure has occurred. These requirements are usually concerned with maintaining copies of the system and its data and specifying how to restore system services after failure.
3. *Redundancy requirements* These specify redundant features of the system that ensure that a single component failure does not lead to a complete loss of service. I discuss this in more detail in the next chapter.
4. *Process requirements* These are fault-avoidance requirements, which ensure that good practice is used in the development process. The practices specified should reduce the number of faults in a system.

Some examples of these types of reliability requirement are shown in Figure 11.5.

There are no simple rules for deriving functional reliability requirements. Organizations that develop critical systems usually have organizational knowledge about possible reliability requirements and how these requirements reflect the actual reliability of a system. These organizations may specialize in specific types of systems, such as railway control systems, so the reliability requirements can be reused across a range of systems.

11.3 Fault-tolerant architectures

Fault tolerance is a runtime approach to dependability in which systems include mechanisms to continue in operation, even after a software or hardware fault has occurred and the system state is erroneous. Fault-tolerance mechanisms detect and correct this erroneous state so that the occurrence of a fault does not lead to a system failure. Fault tolerance is required in systems that are safety or security critical and where the system cannot move to a safe state when an error is detected.

To provide fault tolerance, the system architecture has to be designed to include redundant and diverse hardware and software. Examples of systems that may need fault-tolerant architectures are aircraft systems that must be available throughout the duration of the flight, telecommunication systems, and critical command and control systems.

The simplest realization of a dependable architecture is in replicated servers, where two or more servers carry out the same task. Requests for processing are channeled through a server management component that routes each request to a particular server. This component also keeps track of server responses. In the event of server failure, which can be detected by a lack of response, the faulty server is switched out of the system. Unprocessed requests are resubmitted to other servers for processing.

This replicated server approach is widely used for transaction processing systems where it is easy to maintain copies of transactions to be processed. Transaction processing systems are designed so that data is only updated once a transaction has finished correctly. Delays in processing do not affect the integrity of the system. It can be an efficient way of using hardware if the backup server is one that is normally used for low-priority tasks. If a problem occurs with a primary server, its unprocessed transactions are transferred to the backup server, which gives that work the highest priority.

Replicated servers provide redundancy but not usually diversity. The server hardware is usually identical, and the servers run the same version of the software. Therefore, they can cope with hardware failures and software failures that are localized to a single machine. They cannot cope with software design problems that cause all versions of the software to fail at the same time. To handle software design failures, a system has to use diverse software and hardware.

Torres-Pomales surveys a range of software fault-tolerance techniques (Torres-Pomales 2000), and Pullum (Pullum 2001) describes different types of fault-tolerant architecture. In the following sections, I describe three architectural patterns that have been used in fault-tolerant systems.

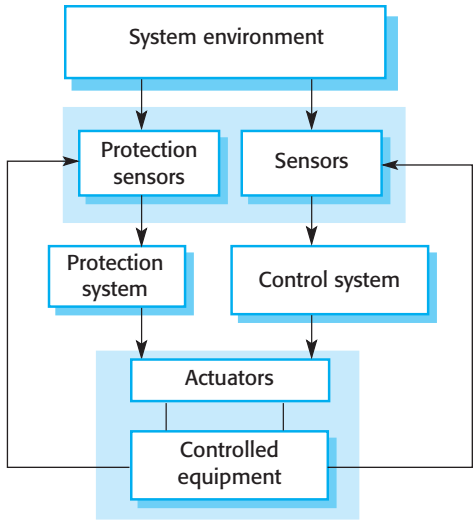


Figure 11.6 Protection system architecture

11.3.1 Protection systems

A protection system is a specialized system that is associated with some other system. This is usually a control system for some process, such as a chemical manufacturing process, or an equipment control system, such as the system on a driverless train. An example of a protection system might be a system on a train that detects if the train has gone through a red signal. If there is no indication that the train control system is slowing down the train, then the protection system automatically applies the train brakes to bring it to a halt. Protection systems independently monitor their environment. If sensors indicate a problem that the controlled system is not dealing with, then the protection system is activated to shut down the process or equipment.

Figure 11.6 illustrates the relationship between a protection system and a controlled system. The protection system monitors both the controlled equipment and the environment. If a problem is detected, it issues commands to the actuators to shut down the system or invoke other protection mechanisms such as opening a pressure-release valve. Notice that there are two sets of sensors. One set is used for normal system monitoring and the other specifically for the protection system. In the event of sensor failure, backups are in place that will allow the protection system to continue in operation. The system may also have redundant actuators.

A protection system only includes the critical functionality that is required to move the system from a potentially unsafe state to a safe state (which could be system shutdown). It is an instance of a more general fault-tolerant architecture in which a principal system is supported by a smaller and simpler backup system that only includes essential functionality. For example, the control software for the U.S. Space Shuttle had a backup system with "get you home" functionality. That is, the backup system could land the vehicle if the principal control system failed but had no other control functions.

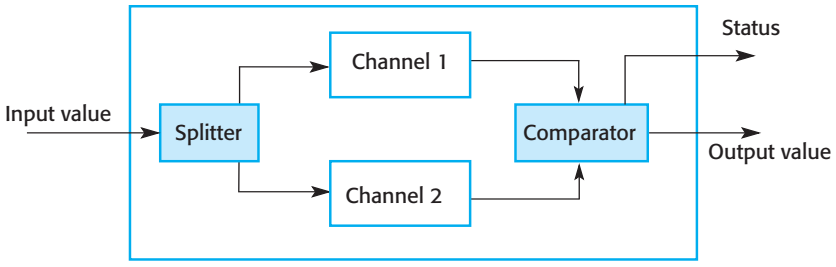


Figure 11.7 Self-monitoring architecture

The advantage of this architectural style is that protection system software can be much simpler than the software that is controlling the protected process. The only function of the protection system is to monitor operation and to ensure that the system is brought to a safe state in the event of an emergency. Therefore, it is possible to invest more effort in fault avoidance and fault detection. You can check that the software specification is correct and consistent and that the software is correct with respect to its specification. The aim is to ensure that the reliability of the protection system is such that it has a very low probability of failure on demand (say, 0.001). Given that demands on the protection system should be rare, a probability of failure on demand of 1/1000 means that protection system failures should be very rare.

11.3.2 Self-monitoring architectures

A self-monitoring architecture (Figure 11.7) is a system architecture in which the system is designed to monitor its own operation and to take some action if a problem is detected. Computations are carried out on separate channels, and the outputs of these computations are compared. If the outputs are identical and are available at the same time, then the system is judged to be operating correctly. If the outputs are different, then a failure is assumed. When this occurs, the system raises a failure exception on the status output line. This signals that control should be transferred to some other system.

To be effective in detecting both hardware and software faults, self-monitoring systems have to be designed so that:

1. The hardware used in each channel is diverse. In practice, this might mean that each channel uses a different processor type to carry out the required computations, or the chipset making up the system may be sourced from different manufacturers. This reduces the probability of common processor design faults affecting the computation.
2. The software used in each channel is diverse. Otherwise, the same software error could arise at the same time on each channel.

On its own, this architecture may be used in situations where it is important for computations to be correct, but where availability is not essential. If the answers

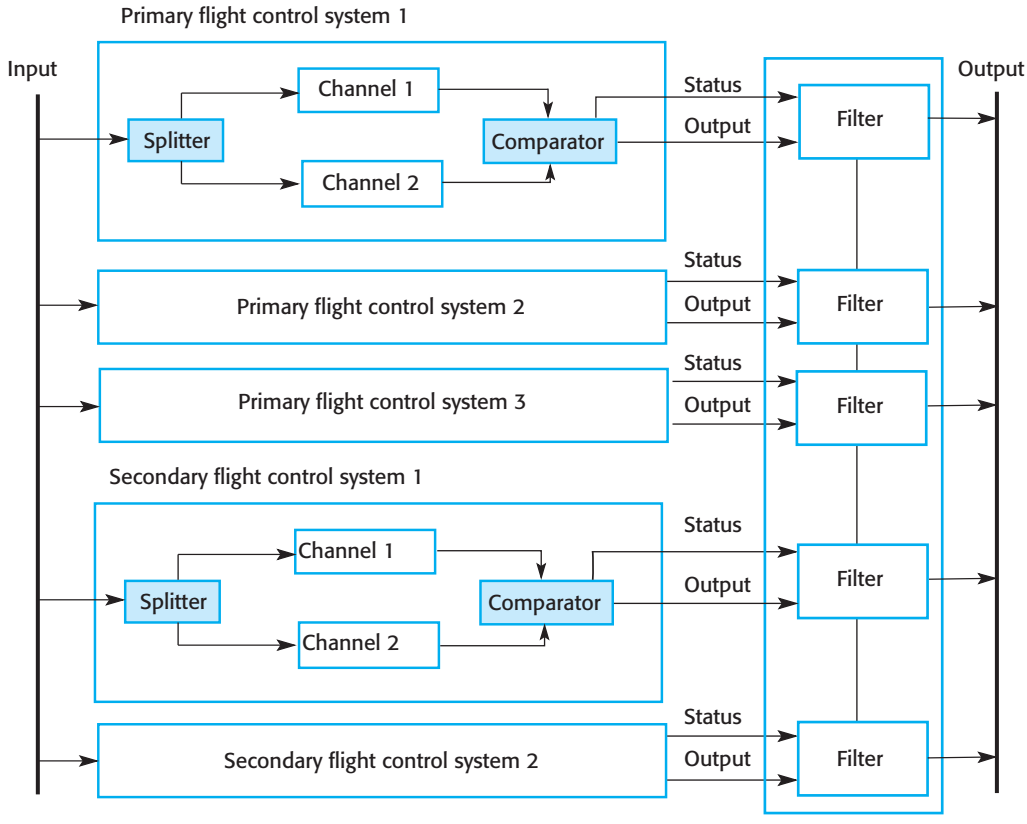


Figure 11.8 The Airbus flight control system architecture

from each channel differ, the system shuts down. For many medical treatment and diagnostic systems, reliability is more important than availability because an incorrect system response could lead to the patient receiving incorrect treatment. However, if the system shuts down in the event of an error, this is an inconvenience but the patient will not usually be harmed.

In situations that require high availability, you have to use several self-checking systems in parallel. You need a switching unit that detects faults and selects a result from one of the systems, where both channels are producing a consistent response. This approach is used in the flight control system for the Airbus 340 series of aircraft, which uses five self-checking computers. Figure 11.8 is a simplified diagram of the Airbus flight control system that shows the organization of the self-monitoring systems.

In the Airbus flight control system, each of the flight control computers carries out the computations in parallel, using the same inputs. The outputs are connected to hardware filters that detect if the status indicates a fault and, if so, that the output from that computer is switched off. The output is then taken from an alternative system. Therefore, it is possible for four computers to fail and for the aircraft operation to continue. In more than 15 years of operation, there have been no reports of situations where control of the aircraft has been lost due to total flight control system failure.

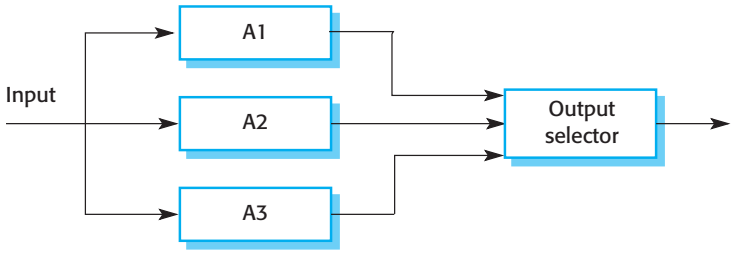


Figure 11.9 Triple modular redundancy

The designers of the Airbus system have tried to achieve diversity in a number of different ways:

1. The primary flight control computers use a different processor from the secondary flight control systems.
2. The chipset that is used in each channel in the primary and secondary systems is supplied by a different manufacturer.
3. The software in the secondary flight control systems provides critical functionality only—it is less complex than the primary software.
4. The software for each channel in both the primary and the secondary systems is developed using different programming languages and by different teams.
5. Different programming languages are used in the secondary and primary systems.

As I discuss in Section 11.3.4, these do not guarantee diversity but they reduce the probability of common failures in different channels.

11.3.3 *N*-version programming

Self-monitoring architectures are examples of systems in which multiversion programming is used to provide software redundancy and diversity. This notion of multiversion programming has been derived from hardware systems where the notion of triple modular redundancy (TMR) has been used for many years to build systems that are tolerant of hardware failures (Figure 11.9).

In a TMR system, the hardware unit is replicated three (or sometimes more) times. The output from each unit is passed to an output comparator that is usually implemented as a voting system. This system compares all of its inputs, and, if two or more are the same, then that value is output. If one of the units fails and does not produce the same output as the other units, its output is ignored. A fault manager may try to repair the faulty unit automatically, but if this is impossible, the system is automatically reconfigured to take the unit out of service. The system then continues to function with two working units.

This approach to fault tolerance relies on most hardware failures being the result of component failure rather than design faults. The components are therefore likely

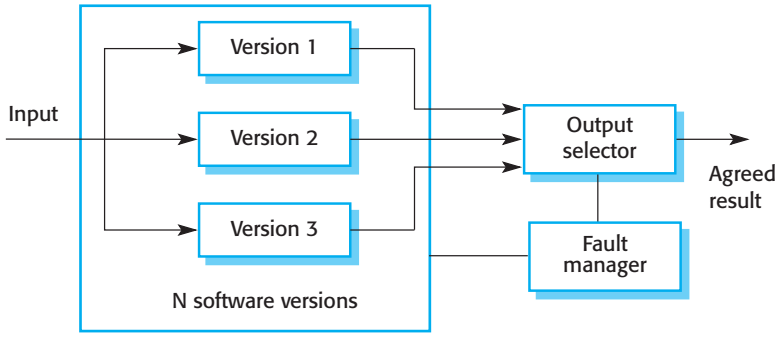


Figure 11.10 *N*-version programming

to fail independently. It assumes that, when fully operational, all hardware units perform to specification. There is therefore a low probability of simultaneous component failure in all hardware units.

Of course, the components could all have a common design fault and thus all produce the same (wrong) answer. Using hardware units that have a common specification but that are designed and built by different manufacturers reduces the chances of such a common mode failure. It is assumed that the probability of different teams making the same design or manufacturing error is small.

A similar approach can be used for fault-tolerant software where *N* diverse versions of a software system execute in parallel (Avizienis 1995). This approach to software fault tolerance, illustrated in Figure 11.10, has been used in railway signaling systems, aircraft systems, and reactor protection systems.

Using a common specification, the same software system is implemented by a number of teams. These versions are executed on separate computers. Their outputs are compared using a voting system, and inconsistent outputs or outputs that are not produced in time are rejected. At least three versions of the system should be available so that two versions should be consistent in the event of a single failure.

N-version programming may be less expensive than self-checking architectures in systems for which a high level of availability is required. However, it still requires several different teams to develop different versions of the software. This leads to very high software development costs. As a result, this approach is only used in systems where it is impractical to provide a protection system that can guard against safety-critical failures.

11.3.4 Software diversity

All of the above fault-tolerant architectures rely on software diversity to achieve fault tolerance. This is based on the assumption that diverse implementations of the same specification (or a part of the specification, for protection systems) are independent. They should not include common errors and so will not fail in the same way, at the same time. The software should therefore be written by different teams who should not communicate during the development process. This requirement reduces the chances of common misunderstandings or misinterpretations of the specification.

The company that is procuring the system may include explicit diversity policies that are intended to maximize the differences between the system versions. For example:

1. By including requirements that different design methods should be used. For example, one team may be required to produce an object-oriented design, and another team may produce a function-oriented design.
2. By stipulating that the programs should be implemented using different programming languages. For example, in a three-version system, Ada, C++, and Java could be used to write the software versions.
3. By requiring the use of different tools and development environments for the system.
4. By requiring different algorithms to be used in some parts of the implementation. However, this limits the freedom of the design team and may be difficult to reconcile with system performance requirements.

Ideally, the diverse versions of the system should have no dependencies and so should fail in completely different ways. If this is the case, then the overall reliability of a diverse system is obtained by multiplying the reliabilities of each channel. So, if each channel has a probability of failure on demand of 0.001, then the overall POFOD of a three-channel system (with all channels independent) is a million times greater than the reliability of a single channel system.

In practice, however, achieving complete channel independence is impossible. It has been shown experimentally that independent software design teams often make the same mistakes or misunderstand the same parts of the specification (Brilliant, Knight, and Leveson 1990; Leveson 1995). There are several reasons for this misunderstanding:

1. Members of different teams are often from the same cultural background and may have been educated using the same approach and textbooks. This means that they may find the same things difficult to understand and have common difficulties in communicating with domain experts. It is quite possible that they will, independently, make the same mistakes and design the same algorithms to solve a problem.
2. If the requirements are incorrect or they are based on misunderstandings about the environment of the system, then these mistakes will be reflected in each implementation of the system.
3. In a critical system, the detailed system specification that is derived from the system's requirements should provide an unambiguous definition of the system's behavior. However, if the specification is ambiguous, then different teams may misinterpret the specification in the same way.

One way to reduce the possibility of common specification errors is to develop detailed specifications for the system independently and to define the specifications in different languages. One development team might work from a formal specification,

another from a state-based system model, and a third from a natural language specification. This approach helps avoid some errors of specification interpretation, but does not get around the problem of requirements errors. It also introduces the possibility of errors in the translation of the requirements, leading to inconsistent specifications.

In an analysis of the experiments, Hatton (Hatton 1997) concluded that a three-channel system was somewhere between 5 and 9 times more reliable than a single-channel system. He concluded that improvements in reliability that could be obtained by devoting more resources to a single version could not match this and so *N*-version approaches were more likely to lead to more reliable systems than single-version approaches.

What is unclear, however, is whether the improvements in reliability from a multiversion system are worth the extra development costs. For many systems, the extra costs may not be justifiable, as a well-engineered single-version system may be good enough. It is only in safety- and mission-critical systems, where the costs of failure are very high, that multiversion software may be required. Even in such situations (e.g., a spacecraft system), it may be enough to provide a simple backup with limited functionality until the principal system can be repaired and restarted.

11.4 Programming for reliability

I have deliberately focused in this book on programming-language independent aspects of software engineering. It is almost impossible to discuss programming without getting into the details of a specific programming language. However, when considering reliability engineering, there are a set of accepted good programming practices that are fairly universal and that help reduce faults in delivered systems.

A list of eight good practice guidelines is shown in Figure 11.11. They can be applied regardless of the particular programming language used for systems development, although the way they are used depends on the specific languages and notations that are used for system development. Following these guidelines also reduces the chances of introducing security-related vulnerabilities into programs.

Guideline 1: Control the visibility of information in a program

A security principle that is adopted by military organizations is the "need to know" principle. Only those individuals who need to know a particular piece of information in order to carry out their duties are given that information. Information that is not directly relevant to their work is withheld.

When programming, you should adopt an analogous principle to control access to the variables and data structures that you use. Program components should only be allowed access to data that they need for their implementation. Other program data should be inaccessible and hidden from them. If you hide information, it cannot be corrupted by program components that are not supposed to use it. If the interface remains the same, the data representation may be changed without affecting other components in the system.

Dependable programming guidelines
1. Limit the visibility of information in a program.
2. Check all inputs for validity.
3. Provide a handler for all exceptions.
4. Minimize the use of error-prone constructs.
5. Provide restart capabilities.
6. Check array bounds.
7. Include timeouts when calling external components.
8. Name all constants that represent real-world values.

Figure 11.11 Good practice guidelines for dependable programming

You can achieve this by implementing data structures in your program as abstract data types. An abstract data type is one in which the internal structure and representation of a variable of that type are hidden. The structure and attributes of the type are not externally visible, and all access to the data is through operations.

For example, you might have an abstract data type that represents a queue of requests for service. Operations should include get and put, which add and remove items from the queue, and an operation that returns the number of items in the queue. You might initially implement the queue as an array but subsequently decide to change the implementation to a linked list. This can be achieved without any changes to code using the queue, because the queue representation is never directly accessed.

In some object-oriented languages, you can implement abstract data types using interface definitions, where you declare the interface to an object without reference to its implementation. For example, you can define an interface Queue, which supports methods to place objects onto the queue, remove them from the queue, and query the size of the queue. In the object class that implements this interface, the attributes and methods should be private to that class.

Guideline 2: Check all inputs for validity

All programs take inputs from their environment and process them. The specification makes assumptions about these inputs that reflect their real-world use. For example, it may be assumed that a bank account number is always an eight-digit positive integer. In many cases, however, the system specification does not define what actions should be taken if the input is incorrect. Inevitably, users will make mistakes and will sometimes enter the wrong data. As I discuss in Chapter 13, malicious attacks on a system may rely on deliberately entering invalid information. Even when inputs come from sensors or other systems, these systems can go wrong and provide incorrect values.

You should therefore always check the validity of inputs as soon as they are read from the program's operating environment. The checks involved obviously depend on the inputs themselves, but possible checks that may be used are:

1. *Range checks* You may expect inputs to be within a particular range. For example, an input that represents a probability should be within the range 0.0 to 1.0; an input that represents the temperature of a liquid water should be between 0 degrees Celsius and 100 degrees Celsius, and so on.

2. *Size checks* You may expect inputs to be a given number of characters, for example, 8 characters to represent a bank account. In other cases, the size may not be fixed, but there may be a realistic upper limit. For example, it is unlikely that a person's name will have more than 40 characters.

3. *Representation checks* You may expect an input to be of a particular type, which is represented in a standard way. For example, people's names do not include numeric characters, email addresses are made up of two parts, separated by a @ sign, and so on.

4. *Reasonableness checks* Where an input is one of a series and you know something about the relationships between the members of the series, then you can check that an input value is reasonable. For example, if the input value represents the readings of a household electricity meter, then you would expect the amount of electricity used to be approximately the same as in the corresponding period in the previous year. Of course, there will be variations. but order of magnitude differences suggest that something has gone wrong.

The actions that you take if an input validation check fails depend on the type of system being implemented. In some cases, you report the problem to the user and request that the value is re-input. Where a value comes from a sensor, you might use the most recent valid value. In embedded real-time systems, you might have to estimate the value based on previous data, so that the system can continue in operation.

Guideline 3: Provide a handler for all exceptions

During program execution, errors or unexpected events inevitably occur. These may arise because of a program fault, or they may be a result of unpredictable external circumstances. An error or an unexpected event that occurs during the execution of a program is called an exception. Examples of exceptions might be a system power failure, an attempt to access nonexistent data, or numeric overflow or underflow.

Exceptions may be caused by hardware or software conditions. When an exception occurs, it must be managed by the system. This can be done within the program itself, or it may involve transferring control to a system exception-handling mechanism. Typically, the system's exception management mechanism reports the error and shuts down execution. Therefore, to ensure that program exceptions do not cause system failure, you should define an exception handler for all possible exceptions that may arise; you should also make sure that all exceptions are detected and explicitly handled.

Languages such as Java, C++, and Python have built-in exception-handling constructs. When an exceptional situation occurs, the exception is signaled and the language runtime system transfers control to an exception handler. This is a code section that states exception names and appropriate actions to handle each exception (Figure 11.12). The exception handler is outside the normal flow of control, and this normal control flow does not resume after the exception has been handled.

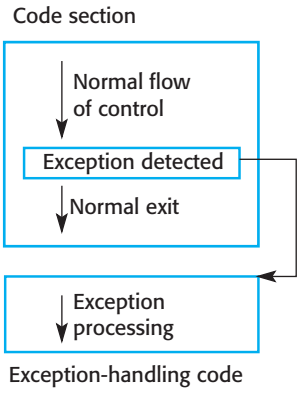


Figure 11.12 Exception handling

An exception handler usually does one of three things:

1. Signals to a higher-level component that an exception has occurred and provides information to that component about the type of exception. You use this approach when one component calls another and the calling component needs to know if the called component has executed successfully. If not, it is up to the calling component to take action to recover from the problem.
2. Carries out some alternative processing to that which was originally intended. Therefore, the exception handler takes some actions to recover from the problem. Processing may then continue as normal. Alternatively, the exception handler may indicate that an exception has occurred so that a calling component is aware of and can deal with the exception.
3. Passes control to the programming language runtime support system that handles the exception. This is often the default when faults occur in a program, for example, when a numeric value overflows. The usual action of the runtime system is to halt processing. You should only use this approach when it is possible to move the system to a safe and quiescent state, before handing over control to the runtime system.

Handling exceptions within a program makes it possible to detect and recover from some input errors and unexpected external events. As such, it provides a degree of fault tolerance. The program detects faults and can take action to recover from them. As most input errors and unexpected external events are usually transient, it is often possible to continue normal operation after the exception has been processed.

Guideline 4: Minimize the use of error-prone constructs

Faults in programs, and therefore many program failures, are usually a consequence of human error. Programmers make mistakes because they lose track of the numerous relationships between the state variables. They write program statements that result in unexpected behavior and system state changes. People will always make



Error-prone constructs

Some programming language features are more likely than others to lead to the introduction of program bugs. Program reliability is likely to be improved if you avoid using these constructs. Wherever possible, you should minimize the use of go to statements, floating-point numbers, pointers, dynamic memory allocation, parallelism, recursion, interrupts, aliasing, unbounded arrays, and default input processing.

http://software-engineering-book.com/web/error-prone-constructs/

mistakes, but in the late 1960s it became clear that some approaches to programming were more likely to introduce errors into a program than others.

For example, you should try to avoid using floating-point numbers because the precision of floating point numbers is limited by their hardware representation. Comparisons of very large or very small numbers are unreliable. Another construct that is potentially error-prone is dynamic storage allocation where you explicitly manage storage in the program. It is very easy to forget to release storage when it's no longer needed, and this can lead to hard to detect runtime errors.

Some standards for safety-critical systems development completely prohibit the use of error-prone constructs. However, such an extreme position is not normally practical. All of these constructs and techniques are useful, though they must be used with care. Wherever possible, their potentially dangerous effects should be controlled by using them within abstract data types or objects. These act as natural "firewalls" limiting the damage caused if errors occur.

Guideline 5: Provide restart capabilities

Many organizational information systems are based on short transactions where processing user inputs takes a relatively short time. These systems are designed so that changes to the system's database are only finalized after all other processing has been successfully completed. If something goes wrong during processing, the database is not updated and so does not become inconsistent. Virtually all e-commerce systems, where you only commit to your purchase on the final screen, work in this way.

User interactions with e-commerce systems usually last a few minutes and involve minimal processing. Database transactions are short and are usually completed in less than a second. However, other types of system such as CAD systems and word processing systems involve long transactions. In a long transaction system, the time between starting to use the system and finishing work may be several minutes or hours. If the system fails during a long transaction, then all of the work may be lost. Similarly, in computationally intensive systems such as some e-science systems, minutes or hours of processing may be required to complete the computation. All of this time is lost in the event of a system failure.

In all of these types of systems, you should provide a restart capability that is based on keeping copies of data collected or generated during processing. The restart facility should allow the system to restart using these copies, rather than having to

start all over from the beginning. These copies are sometimes called checkpoints. For example:

1. In an e-commerce system, you can keep copies of forms filled in by a user and allow them to access and submit these forms without having to fill them in again.
2. In a long transaction or computationally intensive system, you can automatically save data every few minutes and, in the event of a system failure, restart with the most recently saved data. You should also allow for user error and provide a way for users to go back to the most recent checkpoint and start again from there.

If an exception occurs and it is impossible to continue normal operation, you can handle the exception using backward error recovery. This means that you reset the state of the system to the saved state in the checkpoint and restart operation from that point.

Guideline 6: Check array bounds

All programming languages allow the specification of arrays—sequential data structures that are accessed using a numeric index. These arrays are usually laid out in contiguous areas within the working memory of a program. Arrays are specified to be of a particular size, which reflects how they are used. For example, if you wish to represent the ages of up to 10,000 people, then you might declare an array with 10,000 locations to hold the age data.

Some programming languages, such as Java, always check that when a value is entered into an array, the index is within that array. So, if an array A is indexed from 0 to 10,000, an attempt to enter values into elements A [-5] or A [12345] will lead to an exception being raised. However, programming languages such as C and C++ do not automatically include array bound checks and simply calculate an offset from the beginning of the array. Therefore, A [12345] would access the word that was 12345 locations from the beginning of the array, irrespective of whether or not this was part of the array.

These languages do not include automatic array bound checking because this introduces an overhead every time the array is accessed and so it increases program execution time. However, the lack of bound checking leads to security vulnerabilities, such as buffer overflow, which I discuss in Chapter 13. More generally, it introduces a system vulnerability that can lead to system failure. If you are using a language such as C or C++ that does not include array bound checking, you should always include checks that the array index is within bounds.

Guideline 7: Include timeouts when calling external components

In distributed systems, components of the system execute on different computers, and calls are made across the network from component to component. To receive some service, component A may call component B. A waits for B to respond before continuing execution. However, if component B fails to respond for some reason, then component A cannot continue. It simply waits indefinitely for a response. A person

who is waiting for a response from the system sees a silent system failure, with no response from the system. They have no alternative but to kill the waiting process and restart the system.

To avoid this prospect, you should always include timeouts when calling external components. A timeout is an automatic assumption that a called component has failed and will not produce a response. You define a time period during which you expect to receive a response from a called component. If you have not received a response in that time, you assume failure and take back control from the called component. You can then attempt to recover from the failure or tell the system users what has happened and allow them to decide what to do.

Guideline 8: Name all constants that represent real-world values

All nontrivial programs include a number of constant values that represent the values of real-world entities. These values are not modified as the program executes. Sometimes, these are absolute constants and never change (e.g., the speed of light), but more often they are values that change relatively slowly over time. For example, a program to calculate personal tax will include constants that are the current tax rates. These change from year to year, and so the program must be updated with the new constant values.

You should always include a section in your program in which you name all real-world constant values that are used. When using the constants, you should refer to them by name rather than by their value. This has two advantages as far as dependability is concerned:

1. You are less likely to make mistakes and use the wrong value. It is easy to mistype a number, and the system will often be unable to detect a mistake. For example, say a tax rate is 34%. A simple transposition error might lead to this being mistyped as 43%. However, if you mistype a name (such as Standard-tax-rate), this error can be detected by the compiler as an undeclared variable.
2. When a value changes, you do not have to look through the whole program to discover where you have used that value. All you need do is to change the value associated with the constant declaration. The new value is then automatically included everywhere that it is needed.

11.5 Reliability measurement

To assess the reliability of a system, you have to collect data about its operation. The data required may include:

1. The number of system failures given a number of requests for system services. This is used to measure the POFOD and applies irrespective of the time over which the demands are made.

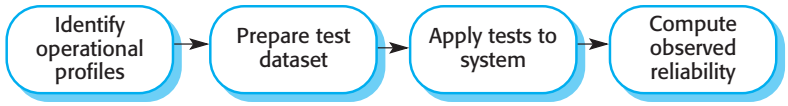


Figure 11.13 Statistical testing for reliability measurement

2. The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.

3. The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

The time units that may be used in these metrics are calendar time or a discrete unit such as number of transactions. You should use calendar time for systems that are in continuous operation. Monitoring systems, such as process control systems, fall into this category. Therefore, the ROCOF might be the number of failures per day. Systems that process transactions such as bank ATMs or airline reservation systems have variable loads placed on them depending on the time of day. In these cases, the unit of "time" used could be the number of transactions; that is, the ROCOF would be number of failed transactions per *N* thousand transactions.

Reliability testing is a statistical testing process that aims to measure the reliability of a system. Reliability metrics such as POFOD, the probability of failure on demand, and ROCOF, the rate of occurrence of failure, may be used to quantitatively specify the required software reliability. You can check on the reliability testing process if the system has achieved that required reliability level.

The process of measuring the reliability of a system is sometimes called statistical testing (Figure 11.13). The statistical testing process is explicitly geared to reliability measurement rather than fault finding. Prowell et al. (Prowell et al. 1999) give a good description of statistical testing in their book on Cleanroom software engineering.

There are four stages in the statistical testing process:

1. You start by studying existing systems of the same type to understand how these are used in practice. This is important as you are trying to measure the reliability as experienced by system users. Your aim is to define an operational profile. An operational profile identifies classes of system inputs and the probability that these inputs will occur in normal use.

2. You then construct a set of test data that reflect the operational profile. This means that you create test data with the same probability distribution as the test data for the systems that you have studied. Normally, you use a test data generator to support this process.

3. You test the system using these data and count the number and type of failures that occur. The times of these failures are also logged. As I discussed in Chapter 10, the time units chosen should be appropriate for the reliability metric used.

4. After you have observed a statistically significant number of failures, you can compute the software reliability and work out the appropriate reliability metric value.

This conceptually attractive approach to reliability measurement is not easy to apply in practice. The principal difficulties that arise are due to:

1. *Operational profile uncertainty* The operational profiles based on experience with other systems may not be an accurate reflection of the real use of the system.
2. *High costs of test data generation* It can be very expensive to generate the large volume of data required in an operational profile unless the process can be totally automated.
3. *Statistical uncertainty when high reliability is specified* You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it is difficult to generate new failures.
4. *Recognizing failure* It is not always obvious whether or not a system failure has occurred. If you have a formal specification, you may be able to identify deviations from that specification, but, if the specification is in natural language, there may be ambiguities that mean observers could disagree on whether the system has failed.

By far the best way to generate the large dataset required for reliability measurement is to use a test data generator, which can be set up to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems because the inputs are often a response to system outputs. Datasets for these systems have to be generated manually, with correspondingly higher costs. Even where complete automation is possible, writing commands for the test data generator may take a significant amount of time.

Statistical testing may be used in conjunction with fault injection to gather data about how effective the process of defect testing has been. Fault injection (Voas and McGraw 1997) is the deliberate injection of errors into a program. When the program is executed, these lead to program faults and associated failures. You then analyze the failure to discover if the root cause is one of the errors that you have added to the program. If you find that X% of the injected faults lead to failures, then proponents of fault injection argue that this suggests that the defect testing process will also have discovered X% of the actual faults in the program.

This approach assumes that the distribution and type of injected faults reflect the actual faults in the system. It is reasonable to think that this might be true for faults due to programming errors, but it is less likely to be true for faults resulting from requirements or design problems. Fault injection is ineffective in predicting the number of faults that stem from anything but programming errors.



Reliability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process. As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

http://software-engineering-book.com/web/reliability-growth-modeling/

11.5.1 Operational profiles

The operational profile of a software system reflects how it will be used in practice. It consists of a specification of classes of input and the probability of their occurrence. When a new software system replaces an existing automated system, it is reasonably easy to assess the probable pattern of usage of the new software. It should correspond to the existing usage, with some allowance made for the new functionality that is (presumably) included in the new software. For example, an operational profile can be specified for telephone switching systems because telecommunication companies know the call patterns that these systems have to handle.

Typically, the operational profile is such that the inputs that have the highest probability of being generated fall into a small number of classes, as shown on the left of Figure 11.14. There are many classes where inputs are highly improbable but not impossible. These are shown on the right of Figure 11.14. The ellipsis (. . .) means that there are many more of these uncommon inputs than are shown.

Musa (Musa 1998) discusses the development of operational profiles in telecommunication systems. As there is a long history of collecting usage data in that domain, the process of operational profile development is relatively straightforward. It simply reflects the historical usage data. For a system that required about 15 person-years of development effort, an operational profile was developed in about 1 person-month. In other cases, operational profile generation took longer (2–3 person-years), but the cost was spread over a number of system releases.

When a software system is new and innovative, however, it is difficult to anticipate how it will be used. Consequently, it is practically impossible to create an accurate operational profile. Many different users with different expectations, backgrounds, and experience may use the new system. There is no historical usage database. These users may make use of systems in ways that the system developers did not anticipate.

Developing an accurate operational profile is certainly possible for some types of system, such as telecommunication systems, that have a standardized pattern of use. However, for other types of system, developing an accurate operational profile may be difficult or impossible:

1. A system may have many different users who each have their own ways of using the system. As I explained earlier in this chapter, different users have

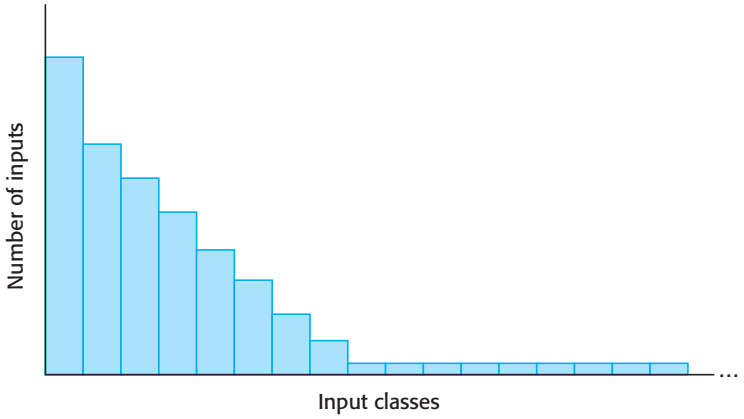


Figure 11.14
Distribution of inputs in an operational profile

different impressions of reliability because they use a system in different ways. It is difficult to match all of these patterns of use in a single operational profile.

2. Users change the ways that they use a system over time. As users learn about a new system and become more confident with it, they start to use it in more sophisticated ways. Therefore, an operational profile that matches the initial usage pattern of a system may not be valid after users become familiar with the system.

For these reasons, it is often impossible to develop a trustworthy operational profile. If you use an out-of-date or incorrect operational profile, you cannot be confident about the accuracy of any reliability measurements that you make.

KEY POINTS

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault-tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification. Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF), and availability (AVAIL).
- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.

- Dependable system architectures are system architectures that are designed for fault tolerance. A number of architectural styles support fault tolerance, including protection systems, self-monitoring architectures, and *N*-version programming.
- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

FURTHER READING

Software Fault Tolerance Techniques and Implementation. A comprehensive discussion of techniques to achieve software fault tolerance and fault-tolerant architectures. The book also covers general issues of software dependability. Reliability engineering is a mature area, and the techniques discussed here are still current. (L. L. Pullum, Artech House, 2001).

"Software Reliability Engineering: A Roadmap." This survey paper by a leading researcher in software reliability summarizes the state of the art in software reliability engineering and discusses research challenges in this area. (M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.24

"Mars Code." This paper discusses the approach to reliability engineering used in the development of software for the Mars Curiosity Rover. This relied on the use of good programming practice, redundancy, and model checking (covered in Chapter 12). (G. J. Holzmann, *Comm. ACM.*, 57 (2), 2014) http://dx.doi.org/10.1145/2560217.2560218

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

More information on the Airbus flight control system:

http://software-engineering-book.com/case-studies/airbus-340/

EXERCISES

11.1. Explain why it is practically impossible to validate reliability specifications when these are expressed in terms of a very small number of failures over the total lifetime of a system.

11.2. Suggest appropriate reliability metrics for the classes of software system below. Give reasons for your choice of metric. Predict the usage of these systems and suggest appropriate values for the reliability metrics.

- a system that monitors patients in a hospital intensive care unit
- a word processor
- an automated vending machine control system
- a system to control braking in a car
- a system to control a refrigeration unit
- a management report generator

11.3. Imagine that a network operations center monitors and controls the national telecommunications network of a country. This includes controlling and monitoring the operational status of switching and transmission equipment and keeping track of nationwide equipment inventories. The center needs to have redundant systems. Explain three reliability metrics you would use to specify the needs of such systems.

11.4. What is the common characteristic of all architectural styles that are geared to supporting software fault tolerance?

11.5. Suggest circumstances where it is appropriate to use a fault-tolerant architecture when implementing a software-based control system and explain why this approach is required.

11.6. You are responsible for the design of a communications switch that has to provide 24/7 availability but that is not safety-critical. Giving reasons for your answer, suggest an architectural style that might be used for this system.

11.7. It has been suggested that the control software for a radiation therapy machine, used to treat patients with cancer, should be implemented using *N*-version programming. Comment on whether or not you think this is a good suggestion.

11.8. Explain why all the versions in a system designed around software diversity may fail in a similar way.

11.9. Explain how programming language support of exception handling can contribute to the reliability of software systems.

11.10. Software failures can cause considerable inconvenience to users of the software. Is it ethical for companies to release software that they know includes faults that could lead to software failures? Should they be liable for compensating users for losses that are caused by the failure of their software? Should they be required by law to offer software warranties in the same way that consumer goods manufacturers must guarantee their products?

REFERENCES

Avizienis, A. A. 1995. "A Methodology of N-Version Programming." In *Software Fault Tolerance*, edited by M. R. Lyu, 23–46. Chichester, UK: John Wiley & Sons.

Brilliant, S. S., J. C. Knight, and N. G. Leveson. 1990. "Analysis of Faults in an N-Version Software Experiment." *IEEE Trans. On Software Engineering* 16 (2): 238–247. doi:10.1109/32.44387.

Hatton, L. 1997. "N-Version Design Versus One Good Version." *IEEE Software* 14 (6): 71–76. doi:10.1109/52.636672.

Leveson, N. G. 1995. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.

Musa, J. D. 1998. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.

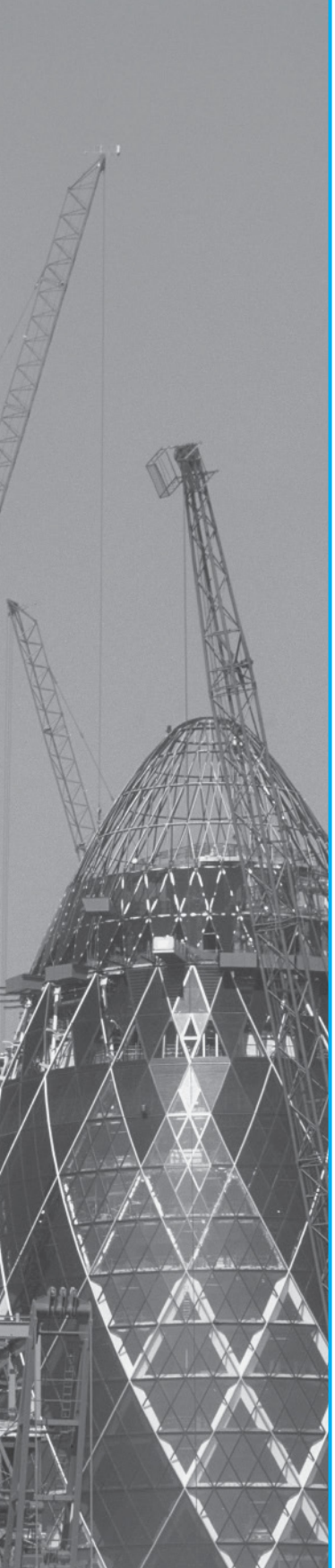
Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.

Pullum, L. 2001. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House.

Randell, B. 2000. "Facing Up To Faults." *Computer J.* 45 (2): 95–106. doi:10.1093/comjnl/43.2.95.

Torres-Pomales, W. 2000. "Software Fault Tolerance: A Tutorial." NASA. http://ntrs.nasa.gov/archive/nasa/casi. . ./20000120144_2000175863.pdf

Voas, J., and G. McGraw. 1997. *Software Fault Injection: Innoculating Programs Against Errors*. New York: John Wiley & Sons.



12
Safety engineering

Objectives

The objective of this chapter is to explain techniques that are used to ensure safety when developing critical systems. When you have read this chapter, you will:

- understand what is meant by a safety-critical system and why safety has to be considered separately from reliability in critical systems engineering;
- understand how an analysis of hazards can be used to derive safety requirements;
- know about processes and tools that are used for software safety assurance;
- understand the notion of a safety case that is used to justify the safety of a system to regulators, and how formal arguments may be used in safety cases.

Contents

12.1 Safety-critical systems
12.2 Safety requirements
12.3 Safety engineering processes
12.4 Safety cases

In Section 11.2, I briefly described an air accident at Warsaw Airport where an Airbus crashed on landing. Two people were killed and 54 were injured. The subsequent inquiry showed that a major contributory cause of the accident was a failure of the control software that reduced the efficiency of the aircraft's braking system. This is one of the, thankfully rare, examples of where the behavior of a software system has led to death or injury. It illustrates that software is now a central component in many systems that are critical to preserving and maintaining life. These are safety-critical software systems, and a range of specialized methods and techniques have been developed for safety-critical software engineering.

As I discussed in Chapter 10, safety is one of the principal dependability properties. A system can be considered to be safe if it operates without catastrophic failure, that is, failure that causes or may cause death or injury to people. Systems whose failure may lead to environmental damage may also be safety-critical as environmental damage (such as a chemical leak) can lead to subsequent human injury or death.

Software in safety-critical systems has a dual role to play in achieving safety:

1. The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical. Therefore, the software behavior is directly related to the overall safety of the system.
2. Software is extensively used for checking and monitoring other safety-critical components in a system. For example, all aircraft engine components are monitored by software looking for early indications of component failure. This software is safety-critical because, if it fails, other components may fail and cause an accident.

Safety in software systems is achieved by developing an understanding of the situations that might lead to safety-related failures. The software is engineered so that such failures do not occur. You might therefore think that if a safety-critical system is reliable and behaves as specified, it will therefore be safe. Unfortunately, it isn't quite as simple as that. System reliability is necessary for safety achievement, but it isn't enough. Reliable systems can be unsafe and vice versa. The Warsaw Airport accident was an example of such a situation, which I'll discuss in more detail in Section 12.2.

Software systems that are reliable may not be safe for four reasons:

1. We can never be 100% certain that a software system is fault-free and fault-tolerant. Undetected faults can be dormant for a long time, and software failures can occur after many years of reliable operation.
2. The specification may be incomplete in that it does not describe the required behavior of the system in some critical situations. A high percentage of system malfunctions are the result of specification rather than design errors. In a study of errors in embedded systems, Lutz (Lutz 1993) concludes that "difficulties with requirements are the key root cause of the safety-related software errors, which have persisted until integration and system testing.[†]"

[†]Lutz, R R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In RE'93, 126–133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.

More recent work by Veras et al. (Veras et al. 2010) in space systems confirms that requirements errors are still a major problem for embedded systems.

3. Hardware malfunctions may cause sensors and actuators to behave in an unpredictable way. When components are close to physical failure, they may behave erratically and generate signals that are outside the ranges that can be handled by the software. The software may then either fail or wrongly interpret these signals.

4. The system operators may generate inputs that are not individually incorrect but that, in some situations, can lead to a system malfunction. An anecdotal example of this occurred when an aircraft undercarriage collapsed while the aircraft was on the ground. Apparently, a technician pressed a button that instructed the utility management software to raise the undercarriage. The software carried out the mechanic's instruction perfectly. However, the system should have disallowed the command unless the plane was in the air.

Therefore, safety has to be considered as well as reliability when developing safety-critical systems. The reliability engineering techniques that I introduced in Chapter 11 are obviously applicable for safety-critical systems engineering. I therefore do not discuss system architectures and dependable programming here but instead focus on techniques for improving and assuring system safety.

12.1 Safety-critical systems

Safety-critical systems are systems in which it is essential that system operation is always safe. That is, the system should never damage people or the system's environment, irrespective of whether or not the system conforms to its specification. Examples of safety-critical systems include control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants, and automobile control systems.

Safety-critical software falls into two classes:

1. *Primary safety-critical software* This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage. The insulin pump software that I introduced in Chapter 1 is an example of a primary safety-critical system. System failure may lead to user injury.

   The insulin pump system is a simple system, but software control is also used in very complex safety-critical systems. Software rather than hardware control is essential because of the need to manage large numbers of sensors and actuators, which have complex control laws. For example, advanced, aerodynamically unstable, military aircraft require continual software-controlled adjustment of their flight surfaces to ensure that they do not crash.

2. *Secondary safety-critical software* This is software that can indirectly result in an injury. An example of such software is a computer-aided engineering design system

whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions. Another example of a secondary safety-critical system is the Mentcare system for mental health patient management. Failure of this system, whereby an unstable patient may not be treated properly, could lead to that patient injuring himself or others.

Some control systems, such as those controlling critical national infrastructure (electricity supply, telecommunications, sewage treatment, etc.), are secondary safety-critical systems. Failure of these systems is unlikely to have immediate human consequences. However, a prolonged outage of the controlled systems could lead to injury and death. For example, failure of a sewage treatment system could lead to a higher level of infectious disease as raw sewage is released into the environment.

I explained in Chapter 11 how software and system availability and reliability are achieved through fault avoidance, fault detection and removal, and fault tolerance. Safety-critical systems development uses these approaches and augments them with hazard-driven techniques that consider the potential system accidents that may occur:

1. *Hazard avoidance* The system is designed so that hazards are avoided. For example, a paper-cutting system that requires an operator to use two hands to press separate buttons simultaneously avoids the hazard of the operator's hands being in the blade's pathway.
2. *Hazard detection and removal* The system is designed so that hazards are detected and removed before they result in an accident. For example, a chemical plant system may detect excessive pressure and open a relief valve to reduce pressure before an explosion occurs.
3. *Damage limitation* The system may include protection features that minimize the damage that may result from an accident. For example, an aircraft engine normally includes automatic fire extinguishers. If there is an engine fire, it can often be controlled before it poses a threat to the aircraft.

A hazard is a system state that could lead to an accident. Using the above example of the paper-cutting system, a hazard arises when the operator's hand is in a position where the cutting blade could injure it. Hazards are not accidents—we often get ourselves into hazardous situations and get out of them without any problems. However, accidents are always preceded by hazards, so reducing hazards reduces accidents.

A hazard is one example of the specialized vocabulary that is used in safety-critical systems engineering. I explain other terminology used in safety-critical systems in Figure 12.1.

We are now actually pretty good at building systems that can cope with one thing going wrong. We can design mechanisms into the system that can detect and recover from single problems. However, when several things go wrong at the same time, accidents are more likely. As systems become more and more complex, we don't understand the relationships between the different parts of the system. Consequently, we cannot predict the consequences of a combination of unexpected system events or failures.

In an analysis of serious accidents, Perrow (Perrow 1984) suggested that almost all of the accidents were due to a combination of failures in different parts of a system.

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events that results in human death or injury, damage to property or to the environment. An overdose of insulin is an example of an accident.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could lead to serious injury or the death of the user of the insulin pump.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from "probable" (say 1/100 chance of a hazard occurring) to "implausible" (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that overestimates the user's blood sugar level is low.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is "very high."
Risk	A measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is medium to low.

Figure 12.1 Safety terminology

Unanticipated combinations of subsystem failures led to interactions that resulted in overall system failure. For example, failure of an air conditioning system may lead to overheating. Once hardware gets hot, its behavior becomes unpredictable, so overheating may lead to the system hardware generating incorrect signals. These wrong signals may then cause the software to react incorrectly.

Perrow made the point that, in complex systems, it is impossible to anticipate all possible combinations of failures. He therefore coined the phrase "normal accidents," with the implication that accidents have to be considered as inevitable when we build complex safety-critical systems.

To reduce complexity, we could use simple hardware controllers rather than software control. However, software-controlled systems can monitor a wider range of conditions than simpler electromechanical systems. They can be adapted relatively easily. They use computer hardware, which has high inherent reliability and which is physically small and lightweight.

Software-controlled systems can provide sophisticated safety interlocks. They can support control strategies that reduce the amount of time people need to spend in hazardous environments. Although software control may introduce more ways in which a system can go wrong, it also allows better monitoring and protection. Therefore, software control can contribute to improvements in system safety.

It is important to maintain a sense of proportion about safety-critical systems. Critical software systems operate without problems most of the time. Relatively few people worldwide have been killed or injured because of faulty software. Perrow is right in say-



Risk-based requirements specification

Risk-based specification is an approach that has been widely used by safety and security-critical systems developers. It focuses on those events that could cause the most damage or that are likely to occur frequently. Events that have only minor consequences or that are extremely rare may be ignored. The risk-based specification process involves understanding the risks faced by the system, discovering their root causes, and generating requirements to manage these risks.

http://software-engineering-book.com/web/risk-based-specification/

ing that accidents will always be a possibility. It is impossible to make a system 100% safe, and society has to decide whether or not the consequences of an occasional accident are worth the benefits that come from the use of advanced technologies.

12.2 Safety requirements

In the introduction to this chapter, I described an air accident at Warsaw Airport where the braking system on an Airbus failed. The inquiry into this accident showed that the braking system software had operated according to its specification. There were no errors in the program. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This episode illustrates that system safety does not just depend on good engineering. It requires attention to detail when the system requirements are derived and the inclusion of special software requirements that are geared to ensuring the safety of a system. Safety requirements are functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.

The starting point for generating functional safety requirements is usually domain knowledge, safety standards, and regulations. These lead to high-level requirements that are perhaps best described as "shall not" requirements. By contrast with normal functional requirements that define what the system shall do, "shall not" requirements define system behavior that is unacceptable. Examples of "shall not" requirements are:

"The system shall not allow reverse thrust mode to be selected when the aircraft is in flight."

"The system shall not allow the simultaneous activation of more than three alarm signals."

"The navigation system shall not allow users to set the required destination when the car is moving."

These "shall not" requirements cannot be implemented directly but have to be decomposed into more specific software functional requirements. Alternatively, they may be implemented through system design decisions such as a decision to use particular types of equipment in the system.

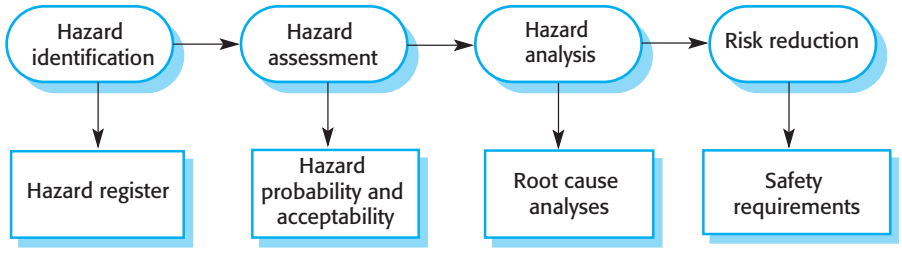


Figure 12.2 Hazard-driven requirements specification

Safety requirements are primarily protection requirements and are not concerned with normal system operation. They may specify that the system should be shut down so that safety is maintained. In deriving safety requirements, you therefore need to find an acceptable balance between safety and functionality and avoid overprotection. There is no point in building a very safe system if it does not operate in a cost-effective way.

Risk-based requirements specification is a general approach used in critical systems engineering where risks faced by the system are identified and requirements to avoid or mitigate these risks are identified. It may be used for all types of dependability requirements. For safety-critical systems, it translates into a process driven by identified hazards. As I discussed in the previous section, a hazard is something that could (but need not) result in death or injury to a person.

There are four activities in a hazard-driven safety specification process:

1. *Hazard identification* The hazard identification process identifies hazards that may threaten the system. These hazards may be recorded in a hazard register. This is a formal document that records the safety analyses and assessments and that may be submitted to a regulator as part of a safety case.
2. *Hazard assessment* The hazard assessment process decides which hazards are the most dangerous and/or the most likely to occur. These should be prioritized when deriving safety requirements.
3. *Hazard analysis* This is a process of root-cause analysis that identifies the events that can lead to the occurrence of a hazard.
4. *Risk reduction* This process is based on the outcome of hazard analysis and leads to identification of safety requirements. These requirements may be concerned with ensuring that a hazard does not arise or lead to an accident or that if an accident does occur, the associated damage is minimized.

Figure 12.2 illustrates this hazard-driven safety requirements specification process.

12.2.1 Hazard identification

In safety-critical systems, hazard identification starts by identifying different classes of hazards, such as physical, electrical, biological, radiation, and service failure hazards. Each of these classes can then be analyzed to discover specific hazards that could occur. Possible combinations of hazards that are potentially dangerous must also be identified.

Experienced engineers, working with domain experts and professional safety advisers, identify hazards from previous experience and from an analysis of the application domain. Group working techniques such as brainstorming may be used, where a group meets to exchange ideas. For the insulin pump system, people who may be involved include doctors, medical physicists and engineers, and software designers.

The insulin pump system that I introduced in Chapter 1 is a safety-critical system, because failure can cause injury or even death to the system user. Accidents that may occur when using this machine include the user suffering from long-term consequences of poor blood sugar control (eye, heart, and kidney problems), cognitive dysfunction as a result of low blood sugar levels, or the occurrence of some other medical conditions, such as an allergic reaction.

Some of the hazards that may arise in the insulin pump system are:

- insulin overdose computation (service failure);
- insulin underdose computation (service failure);
- failure of the hardware monitoring system (service failure);
- power failure due to exhausted battery (electrical);
- electrical interference with other medical equipment such as a heart pacemaker (electrical);
- poor sensor and actuator contact caused by incorrect fitting (physical);
- parts of machine breaking off in patient's body (physical);
- infection caused by introduction of machine (biological); and
- allergic reaction to the materials or insulin used in the machine (biological).

Software-related hazards are normally concerned with failure to deliver a system service or with the failure of monitoring and protection systems. Monitoring and protection systems may be included in a device to detect conditions, such as a low battery level, which could lead to device failure.

A hazard register may be used to record the identified hazards with an explanation of why the hazard has been included. The hazard register is an important legal document that records all safety-related decisions about each hazard. It can be used to show that the requirements engineers have paid due care and attention in considering all foreseeable hazards and that these hazards have been analyzed. In the event of an accident, the hazard register may be used in a subsequent inquiry or legal proceedings to show that the system developers have not been negligent in their system safety analysis.

12.2.2 Hazard assessment

The hazard assessment process focuses on understanding the factors that lead to the occurrence of a hazard and the consequences if an accident or incident associated with that hazard should occur. You need to carry out this analysis to understand

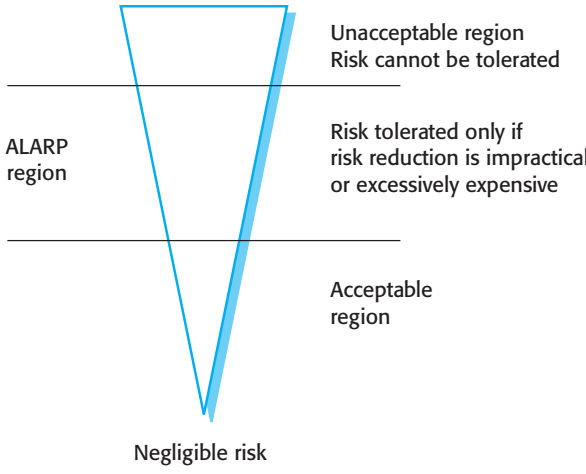


Figure 12.3 The risk triangle

whether a hazard is a serious threat to the system or environment. The analysis also provides a basis for deciding on how to manage the risk associated with the hazard.

For each hazard, the outcome of the analysis and classification process is a statement of acceptability. This is expressed in terms of risk, where the risk takes into account the likelihood of an accident and its consequences. There are three risk categories that are used in hazard assessment:

1. *Intolerable risks* in safety-critical systems are those that threaten human life. The system must be designed so that such hazards either cannot arise or, that if they do, features in the system will ensure that they are detected before they cause an accident. In the case of the insulin pump, an intolerable risk is that an overdose of insulin should be delivered.

2. *As low as reasonably practical (ALARP) risks* are those that have less serious consequences or that are serious but have a very low probability of occurrence. The system should be designed so that the probability of an accident arising because of a hazard is minimized, subject to other considerations such as cost and delivery. An ALARP risk for an insulin pump might be the failure of the hardware monitoring system. The consequences of this failure are, at worst, a short-term insulin underdose. This situation would not lead to a serious accident.

3. *Acceptable risks* are those where the associated accidents normally result in minor damage. System designers should take all possible steps to reduce "acceptable" risks, as long as these measures do not significantly increase costs, delivery time, or other non-functional system attributes. An acceptable risk in the case of the insulin pump might be the risk of an allergic reaction arising in the user. This reaction usually causes only minor skin irritation. It would not be worth using special, more expensive materials in the device to reduce this risk.

Figure 12.3 shows these three regions. The width of the triangle reflects the costs of ensuring that risks do not result in incidents or accidents. The highest

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1. Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

Figure 12.4 Risk classification for the insulin pump

costs are incurred by risks at the top of the diagram, the lowest costs by risks at the apex of the triangle.

The boundaries between the regions in Figure 12.3 are not fixed but depend on how acceptable risks are in the societies where the system will be deployed. This varies from country to country—some societies are more risk averse and litigious than others. Over time, however, all societies have become more risk-averse, so the boundaries have moved downward. For rare events, the financial costs of accepting risks and paying for any resulting accidents may be less than the costs of accident prevention. However, public opinion may demand that money be spent to reduce the likelihood of a system accident irrespective of cost.

For example, it may be cheaper for a company to clean up pollution on the rare occasion it occurs, rather than to install systems for pollution prevention. However, because the public and the media will not tolerate such accidents, clearing up the damage rather than preventing the accident is no longer acceptable. Events in other systems may also lead to a reclassification of risk. For example, risks that were thought to be improbable (and hence in the ALARP region) may be reclassified as intolerable because of external events, such as terrorist attacks, or natural phenomena, such as tsunamis.

Figure 12.4 shows a risk classification for the hazards identified in the previous section for the insulin delivery system. I have separated the hazards that relate to the incorrect computation of insulin into an insulin overdose and an insulin underdose. An insulin overdose is potentially more serious than an insulin underdose in the short term. Insulin overdose can result in cognitive dysfunction, coma, and ultimately death. Insulin underdoses lead to high levels of blood sugar. In the short term, these high levels cause tiredness but are not very serious; in the longer term, however, they can lead to serious heart, kidney, and eye problems.

Hazards 4–9 in Figure 12.4 are not software related, but software nevertheless has a role to play in hazard detection. The hardware monitoring software should monitor the system state and warn of potential problems. The warning will often allow the hazard to

be detected before it causes an accident. Examples of hazards that might be detected are power failure, which is detected by monitoring the battery, and incorrect fitting of the machine, which may be detected by monitoring signals from the blood sugar sensor.

The monitoring software in the system is, of course, safety-related. Failure to detect a hazard could result in an accident. If the monitoring system fails but the hardware is working correctly, then this is not a serious failure. However, if the monitoring system fails and hardware failure cannot then be detected, then this could have more serious consequences.

Hazard assessment involves estimating the hazard probability and risk severity. This is difficult as hazards and accidents are uncommon. Consequently, the engineers involved may not have direct experience of previous incidents or accidents. In estimating probabilities and accident severity, it makes sense to use relative terms such as *probable*, *unlikely*, *rare*, *high*, *medium*, and *low*. Quantifying these terms is practically impossible because not enough statistical data is available for most types of accident.

12.2.3 Hazard analysis

Hazard analysis is the process of discovering the root causes of hazards in a safety-critical system. Your aim is to find out what events or combination of events could cause a system failure that results in a hazard. To do this, you can use either a top-down or a bottom-up approach. Deductive, top-down techniques, which are easier to use, start with the hazard and work from that to the possible system failure. Inductive, bottom-up techniques start with a proposed system failure and identify what hazards might result from that failure.

Various techniques have been proposed as possible approaches to hazard decomposition or analysis (Storey 1996). One of the most commonly used techniques is fault tree analysis, a top-down technique that was developed for the analysis of both hardware and software hazards (Leveson, Cha, and Shimeall 1991). This technique is fairly easy to understand without specialist domain knowledge.

To do a fault tree analysis, you start with the hazards that have been identified. For each hazard, you then work backwards to discover the possible causes of that hazard. You put the hazard at the root of the tree and identify the system states that can lead to that hazard. For each of these states, you then identify further system states that can lead to them. You continue this decomposition until you reach the root cause(s) of the risk. Hazards that can only arise from a combination of root causes are usually less likely to lead to an accident than hazards with a single root cause.

Figure 12.5 is a fault tree for the software-related hazards in the insulin delivery system that could lead to an incorrect dose of insulin being delivered. In this case, I have merged insulin underdose and insulin overdose into a single hazard, namely, "incorrect insulin dose administered." This reduces the number of fault trees that are required. Of course, when you specify how the software should react to this hazard, you have to distinguish between an insulin underdose and an insulin overdose. As I have said, they are not equally serious—in the short term, an overdose is the more serious hazard.

From Figure 12.5, you can see that:

1. Three conditions could lead to the administration of an incorrect dose of insulin. (1) The level of blood sugar may have been incorrectly measured, so the insulin requirement has been computed with an incorrect input. (2) The delivery system

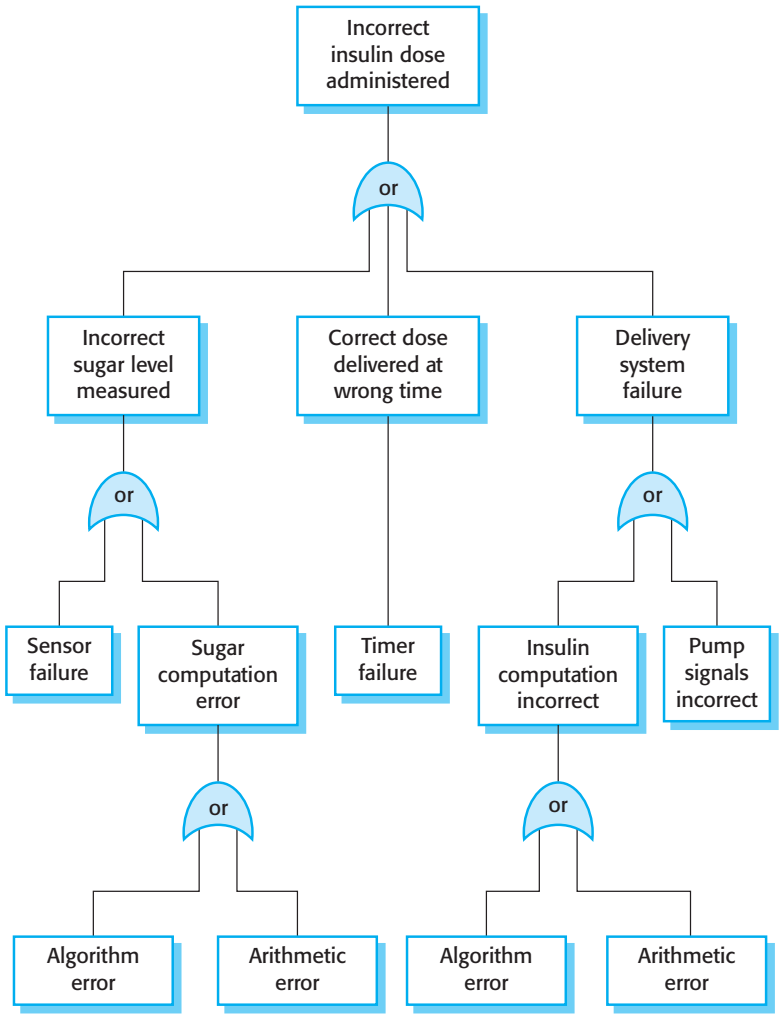


Figure 12.5 An example of a fault tree

may not respond correctly to commands specifying the amount of insulin to be injected. Alternatively, (3) the dose may be correctly computed, but it is delivered too early or too late.

2. The left branch of the fault tree, concerned with incorrect measurement of the blood sugar level, identifies how this might happen. This could occur either because the sensor that provides an input to calculate the sugar level has failed or because the calculation of the blood sugar level has been carried out incorrectly. The sugar level is calculated from some measured parameter, such as the conductivity of the skin. Incorrect computation can result from either an incorrect algorithm or an arithmetic error that results from the use of floating-point numbers.

3. The central branch of the tree is concerned with timing problems and concludes that these can only result from system timer failure.

4. The right branch of the tree, concerned with delivery system failure, examines possible causes of this failure. These could result from an incorrect computation of the insulin requirement or from a failure to send the correct signals to the pump that delivers the insulin. Again, an incorrect computation can result from algorithm failure or arithmetic errors.

Fault trees are also used to identify potential hardware problems. Hardware fault trees may provide insights into requirements for software to detect and, perhaps, correct these problems. For example, insulin doses are not administered frequently—no more than five or six times per hour and sometimes less often than that. Therefore, processor capacity is available to run diagnostic and self-checking programs. Hardware errors such as sensor, pump, or timer errors can be discovered and warnings issued before they have a serious effect on the patient.

12.2.4 Risk reduction

Once potential risks and their root causes have been identified, you are then able to derive safety requirements that manage the risks and ensure that incidents or accidents do not occur. You can use three possible strategies:

1. *Hazard avoidance*, where a system is designed so that the hazard cannot occur.
2. *Hazard detection and removal*, where a system is designed so that hazards are detected and neutralized before they result in an accident.
3. *Damage limitation*, where a system is designed so that the consequences of an accident are minimized.

Normally, designers of critical systems use a combination of these approaches. In a safety-critical system, intolerable hazards may be handled by minimizing their probability and adding a protection system (see Chapter 11) that provides a safety backup. For example, in a chemical plant control system, the system will attempt to detect and avoid excess pressure in the reactor. However, there may also be an independent protection system that monitors the pressure and opens a relief valve if high pressure is detected.

In the insulin delivery system, a safe state is a shutdown state where no insulin is injected. Over a short period, this is not a threat to the diabetic's health. For the software failures that could lead to an incorrect dose of insulin, the following "solutions" might be developed:

1. *Arithmetic error* This error may occur when an arithmetic computation causes a representation failure. The specification should identify all possible arithmetic errors that may occur and state that an exception handler must be included for each possible error. The specification should set out the action to be taken for each of these errors. The default safe action is to shut down the delivery system and activate a warning alarm.
2. *Algorithmic error* This is a more difficult situation as there is no clear program exception that must be handled. This type of error could be detected by comparing

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message as defined in Table 4 shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Note: Tables 3 and 4 relate to tables that are included in the requirements document; they are not shown here.

Figure 12.6 Examples of safety requirements

the required insulin dose computed with the previously delivered dose. If it is much higher, this may mean that the amount has been computed incorrectly. The system may also keep track of the dose sequence. After a number of above-average doses have been delivered, a warning may be issued and further dosage limited.

Some of the resulting safety requirements for the insulin pump software are shown in Figure 12.6. The requirements in Figure 12.6 are user requirements. Naturally, they would be expressed in more detail in a more detailed system requirements specification.

12.3 Safety engineering processes

The software processes used to develop safety-critical software are based on the processes used in software reliability engineering. In general, a great deal of care is taken in developing a complete, and often very detailed, system specification. The design and implementation of the system usual follow a plan-based, waterfall model, with reviews and checks at each stage in the process. Fault avoidance and fault detection are the drivers of the process. For some types of system, such as aircraft systems, fault-tolerant architectures, as I discussed in Chapter 11, may be used.

Reliability is a prerequisite for safety-critical systems. Because of the very high costs and potentially tragic consequences of system failure, additional verification activities may be used in safety-critical systems development. These activities may include developing formal models of a system, analyzing them to discover errors and inconsistencies, and using static analysis software tools that parse the software source code to discover potential faults.

Safe systems have to be reliable, but, as I have discussed, reliability is not enough. Requirements and verification errors and omissions may mean that reliable systems are unsafe. Therefore, safety-critical systems development processes should include

safety reviews, where engineers and system stakeholders examine the work done and explicitly look for potential issues that could affect the safety of the system.

Some types of safety-critical systems are regulated, as I explained in Chapter 10. National and international regulators require detailed evidence that the system is safe. This evidence might include:

1. The specification of the system that has been developed and records of the checks made on that specification.
2. Evidence of the verification and validation processes that have been carried out and the results of the system verification and validation.
3. Evidence that the organizations developing the system have defined and dependable software processes that include safety assurance reviews. There must also be records showing that these processes have been properly enacted.

Not all safety-critical systems are regulated. For example, there is no regulator for automobiles, although cars now have many embedded computer systems. The safety of car-based systems is the responsibility of the car manufacturer. However, because of the possibility of legal action in the event of an accident, developers of unregulated systems have to maintain the same detailed safety information. If a case is brought against them, they have to be able to show that they have not been negligent in the development of the car's software.

The need for this extensive process and product documentation is another reason why agile processes cannot be used, without significant change, for safety-critical systems development. Agile processes focus on the software itself and (rightly) argue that a great deal of process documentation is never actually used after it has been produced. However, where you have to keep records for legal or regulatory reasons, you must maintain documentation about both the processes used and the system itself.

Safety-critical systems, like other types of system that have high dependability requirements, need to be based on dependable processes (see Chapter 10). A dependable process will normally include activities such as requirements management, change management and configuration control, system modeling, reviews and inspections, test planning, and test coverage analysis. When a system is safety-critical, there may be additional safety assurance and verification and analyses processes.

12.3.1 Safety assurance processes

Safety assurance is a set of activities that check that a system will operate safely. Specific safety assurance activities should be included at all stages in the software development process. These activities record the safety analyses that have been carried out and the person or persons responsible for these analyses. Safety assurance activities have to be thoroughly documented. This documentation may be part of the evidence that is used to convince a regulator or system owner that a system will operate safely.

Examples of safety assurance activities are:

1. *Hazard analysis and monitoring*, where hazards are traced from preliminary hazard analysis through to testing and system validation.
2. *Safety reviews*, which are used throughout the development process.
3. *Safety certification*, where the safety of critical components is formally certified. This involves a group external to the system development team examining the available evidence and deciding whether or not a system or component should be considered to be safe before it is made available for use.

To support these safety assurance processes, project safety engineers should be appointed who have explicit responsibility for the safety aspects of a system. These individuals will be accountable if a safety-related system failure occurs. They must be able to demonstrate that the safety assurance activities have been properly carried out.

Safety engineers work with quality managers to ensure that a detailed configuration management system is used to track all safety-related documentation and keep it in step with the associated technical documentation. There is little point in having stringent validation procedures if a failure of configuration management means that the wrong system is delivered to the customer. Quality and configuration management are covered in Chapters 24 and 25.

Hazard analysis is an essential part of safety-critical systems development. It involves identifying hazards, their probability of occurrence, and the probability of a hazard leading to an accident. If there is program code that checks for and handles each hazard, then you can argue that these hazards will not result in accidents. Where external certification is required before a system is used (e.g., in an aircraft), it is usually a condition of certification that this traceability can be demonstrated.

The central safety document that should be produced is the hazard register. This document provides evidence of how identified hazards have been taken into account during software development. This hazard register is used at each stage of the software development process to document how that development stage has taken the hazards into account.

A simplified example of a hazard register entry for the insulin delivery system is shown in Figure 12.7. This register documents the process of hazard analysis and shows design requirements that have been generated during this process. These design requirements are intended to ensure that the control system can never deliver an insulin overdose to a user of the insulin pump.

Individuals who have safety responsibilities should be explicitly identified in the hazard register. Personal identification is important for two reasons:

1. When people are identified, they can be held accountable for their actions. They are likely to take more care because any problems can be traced back to their work.
2. In the event of an accident, there may be legal proceedings or an inquiry. It is important to be able to identify those responsible for safety assurance so that they can defend their actions as part of the legal process.

Hazard Register. **Page 4: Printed 20.02.2012**

System: Insulin Pump System *File:* InsulinPump/Safety/HazardLog
Safety Engineer: James Brown *Log version:* 1/3

Identified Hazard	Insulin overdose delivered to patient				
Identified by	Jane Williams				
Criticality class	1				
Identified risk	High				
Fault tree identified	YES	*Date*	24.01.11	*Location*	Hazard register, Page 5
Fault tree creators	Jane Williams and Bill Smith				
Fault tree checked	YES	*Date*	28.01.11	*Checker*	James Brown

System safety design requirements

1. The system shall include self-testing software that will test the sensor system, the clock, and the insulin delivery system.
2. The self-checking software shall be executed once per minute.
3. In the event of the self-checking software discovering a fault in any of the system components, an audible warning shall be issued and the pump display shall indicate the name of the component where the fault has been discovered. The delivery of insulin shall be suspended.
4. The system shall incorporate an override system that allows the system user to modify the computed dose of insulin that is to be delivered by the system.
5. The amount of override shall be no greater than a pre-set value (maxOverride), which is set when the system is configured by medical staff.

Figure 12.7 A simplified hazard register entry

Safety reviews are reviews of the software specification, design, and source code whose aim is to discover potentially hazardous conditions. These are not automated processes but involve people carefully checking for errors that have been made and for assumptions or omissions that may affect the safety of a system. For example, in the aircraft accident that I introduced earlier, a safety review might have questioned the assumption that an aircraft is on the ground when there is weight on both wheels and the wheels are rotating.

Safety reviews should be driven by the hazard register. For each of the identified hazards, a review team examines the system and judges whether or not it would cope with that hazard in a safe way. Any doubts raised are flagged in the review team's report and have to be addressed by the system development team. I discuss reviews of different types in more detail in Chapter 24, which covers software quality assurance.

Software safety certification is used when external components are incorporated into a safety-critical system. When all parts of a system have been locally developed, complete information about the development processes used can be maintained. However, it is not cost-effective to develop components that are readily available from other vendors. The problem for safety-critical systems development is that these external components may have been developed to different standards than locally developed components. Their safety is unknown.

Consequently, it may be a requirement that all external components must be certified before they can be integrated with a system. The safety certification team, which is separate from the development team, carries out extensive verification and validation of



Licensing of software engineers

In some areas of engineering, safety engineers must be licensed engineers. Inexperienced, poorly qualified engineers are not allowed to take responsibility for safety. In 30 states of the United States, there is some form of licensing for software engineers involved in safety-related systems development. These states require that engineering involved in safety-critical software development should be licensed engineers, with a defined minimum level of qualifications and experience. This is a controversial issue, and licensing is not required in many other countries.

http://software-engineering-book.com/safety-licensing/

the components. If appropriate, they liaise with the component developers to check that the developers have used dependable processes to create these components and to examine the component source code. Once the safety certification team is satisfied that a component meets its specification and does not have "hidden" functionality, they may issue a certificate allowing that component to be used in safety-critical systems.

12.3.2 Formal verification

Formal methods of software development, as I discussed in Chapter 10, rely on a formal model of the system that serves as a system specification. These formal methods are mainly concerned with mathematically analyzing the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.

The need for assurance in safety-critical systems has been one of the principal drivers in the development of formal methods. Comprehensive system testing is extremely expensive and cannot be guaranteed to uncover all of the faults in a system. This is particularly true of systems that are distributed, so that system components are running concurrently. Several safety-critical railway systems were developed using formal methods in the 1990s (Dehbonei and Mejia 1995; Behm et al. 1999). Companies such as Airbus routinely use formal methods in their software development for critical systems (Souyris et al. 2009).

Formal methods may be used at different stages in the V & V process:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions. Model checking, discussed in the next section, is a particularly effective approach to specification analysis.
2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification. It is effective in discovering programming and some design errors.

Because of the wide semantic gap between a formal system specification and program code, it is difficult and expensive to prove that a separately developed program is

consistent with its specification. Work on program verification is now mostly based on transformational development. In a transformational development process, a formal specification is systematically transformed through a series of representations to program code. Software tools support the development of the transformations and help verify that corresponding representations of the system are consistent. The B method is probably the most widely used formal transformational method (Abrial 2010). It has been used for the development of train control systems and avionics software.

Advocates of formal methods claim that the use of these methods leads to more reliable and safer systems. Formal verification demonstrates that the developed program meets its specification and that implementation errors will not compromise the dependability of the system. If you develop a formal model of concurrent systems using a specification written in a language such as CSP (Schneider 1999), you can discover conditions that might result in deadlock in the final program, and you will be able to address these problems. This is very difficult to do by testing alone.

However, formal specification and proof do not guarantee that the software will be safe in practical use:

1. The specification may not reflect the real requirements of users and other system stakeholders. As I discussed in Chapter 10, system system stakeholders rarely understand formal notations, so they cannot directly read the formal specification to find errors and omissions. This means that there it is likely that the formal specification is not an accurate representation of the system requirements.
2. The proof may contain errors. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
3. The proof may make incorrect assumptions about the way that the system is used. If the system is not used as anticipated, then the system's behavior lies outside the scope of the proof.

Verifying a nontrivial software system takes a great deal of time. It requires mathematical expertise and specialized software tools, such as theorem provers. It is an expensive process, and, as the system size increases, the costs of formal verification increase disproportionately.

Many software engineers therefore think that formal verification is not cost-effective. They believe that the same level of confidence in the system can be achieved more cheaply by using other validation techniques, such as inspections and system testing. However, companies such as Airbus that make use of formal verification claim that unit testing of components is not required, which leads to significant cost savings (Moy et al. 2013).

I am convinced that that formal methods and formal verification have an important role to play in the development of critical software systems. Formal specifications are very effective in discovering some types of specification problems that may lead to system failure. Although formal verification remains impractical for large systems, it can be used to verify critical safety and security critical core components.

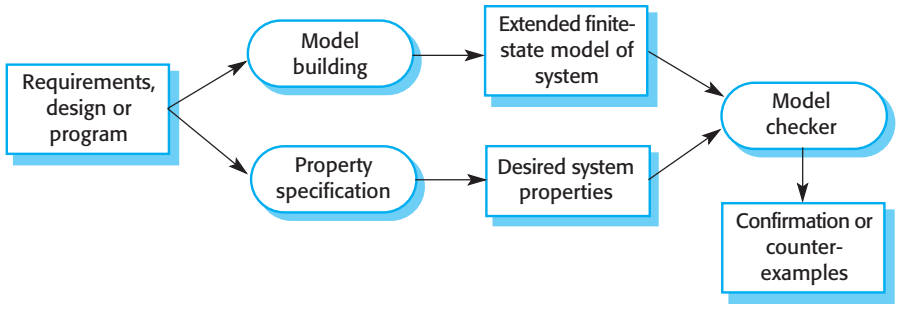


Figure 12.8 Model checking

12.3.3 Model checking

Formally verifying programs using a deductive approach is difficult and expensive, but alternative approaches to formal analysis have been developed that are based on a more restricted notion of correctness. The most successful of these approaches is called model checking (Jhala and Majumdar 2009). Model checking involves creating a formal state model of a system and checking the correctness of that model using specialized software tools. The stages involved in model checking are shown in Figure 12.8.

Model checking has been widely used to check hardware systems designs. It is increasingly being used in critical software systems such as the control software in NASA's Mars exploration vehicles (Regan and Hamilton 2004; Holzmann 2014) and by Airbus in avionics software development (Bochot et al. 2009).

Many different model-checking tools have been developed. SPIN was an early example of a software model checker (Holzmann, 2003). More recent systems include SLAM from Microsoft (Ball, Levin, and Rajamani 2011) and PRISM (Kwiatkowska, Norman, and Parker 2011).

The models used by model-checking systems are extended finite-state models of the software. Models are expressed in the language of whatever model-checking system is used—for example, the SPIN model checker uses a language called Promela. A set of desirable system properties are identified and written in a formal notation, usually based on temporal logic. For example, in the wilderness weather system, a property to be checked might be that the system will always reach the "transmitting" state from the "recording" state.

The model checker then explores all paths through the model (i.e., all possible state transitions), checking that the property holds for each path. If it does, then the model checker confirms that the model is correct with respect to that property. If it does not hold for a particular path, the model checker outputs a counterexample illustrating where the property is not true. Model checking is particularly useful in the validation of concurrent systems, which are notoriously difficult to test because of their sensitivity to time. The checker can explore interleaved, concurrent transitions and discover potential problems.

A key issue in model checking is the creation of the system model. If the model has to be created manually (from a requirements or design document), it is an expensive process as model creation takes a great deal of time. In addition, there is the possibility that the model created will not be an accurate model of the requirements or design. It is therefore

best if the model can be created automatically from the program source code. Model checkers are available that work directly from programs in Java, C, C++, and Ada.

Model checking is computationally very expensive because it uses an exhaustive approach to check all paths through the system model. As the size of a system increases, so too does the number of states, with a consequent increase in the number of paths to be checked. For large systems, therefore, model checking may be impractical, due to the computer time required to run the checks. However, better algorithms are under development that can identify parts of the state that do not have to be explored when checking a particular property. As these algorithms are incorporated into model checkers, it will be increasingly possible to use model checking routinely in large-scale critical systems development.

12.3.4 Static program analysis

Automated static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies. They parse the program text and thus recognize the different types of statements in a program. They can then detect whether or not statements are well formed, make inferences about the control flow in the program, and, in many cases, compute the set of all possible values for program data. They complement the error-detection facilities provided by the language compiler, and they can be used as part of the inspection process or as a separate V & V process activity.

Automated static analysis is faster and cheaper than detailed code reviews and is very effective in discovering some types of program faults. However, it cannot discover some classes of errors that could be identified in program inspection meetings.

Static analysis tools (Lopes, Vicente, and Silva 2009) work on the source code of a system, and, for some types of analysis at least, no further inputs are required. This means that programmers do not need to learn specialized notations to write program specifications, so the benefits of analysis can be immediately clear. This makes automated static analysis easier to introduce into a development process than formal verification or model checking.

The intention of automatic static analysis is to draw a code reader's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused, or data whose values could go out of range. Examples of the problems that can be detected by static analysis are shown in Figure 12.9.

Of course, the specific checks made by the static analyzer are programming-language-specific and depend on what is and isn't allowed in the language. Anomalies are often a result of programming errors or omissions, so they highlight things that could go wrong when the program is executed. However, these anomalies are not necessarily program faults; they may be deliberate constructs introduced by the programmer, or the anomaly may have no adverse consequences.

Three levels of checking may be implemented in static analyzers:

1. *Characteristic error checking* At this level, the static analyzer knows about common errors that are made by programmers in languages such as Java or C. The tool analyzes the code looking for patterns that are characteristic of that problem

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Nonusage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Figure 12.9 Automated static analysis checks

and highlights these to the programmer. Though relatively simple, analysis based on common errors can be very cost-effective. Zheng and his collaborators (Zheng et al. 2006) analyzed a large code base in C and C++. They discovered that 90% of the errors in the programs resulted from 10 types of characteristic error.

2. *User-defined error checking* In this approach, the users of the static analyzer define error patterns to be detected. These may relate to the application domain or may be based on knowledge of the specific system that is being developed. An example of an error pattern is "maintain ordering"; for example, method A must always be called before method B. Over time, an organization can collect information about common bugs that occur in their programs and extend the static analysis tools with error patterns to highlight these errors.

3. *Assertion checking* This is the most general and most powerful approach to static analysis. Developers include formal assertions (often written as stylized comments) in their program that state relationships that must hold at that point in a program. For example, the program might include an assertion stating that the value of some variable must lie in the range x..y. The analyzer symbolically executes the code and highlights statements where the assertion may not hold.

Static analysis is effective in finding errors in programs but, commonly, generates a large number of false positives. These are code sections where there are no errors but where the static analyzer's rules have detected a potential for error. The number of false positives can be reduced by adding more information to the program in the form of assertions, but this requires additional work by the developer of the code. Work has to be done in screening out these false positives before the code itself can be checked for errors.

Many organizations now routinely use static analysis in their software development processes. Microsoft introduced static analysis in the development of device

drivers where program failures can have a serious effect. They extended the approach across a much wider range of their software to look for security problems as well as errors that affect program reliability (Ball, Levin, and Rajamani 2011). Checking for well-known problems, such as buffer overflow, is effective for improving security as attackers often base their attacks on those common vulnerabilities. Attacks may target little-used code sections that may not have been thoroughly tested. Static analysis is a cost-effective way of finding these types of vulnerability.

12.4 Safety cases

As I have discussed, many safety-critical, software-intensive systems are regulated. An external authority has significant influence on their development and deployment. Regulators are government bodies whose job is to ensure that commercial companies do not deploy systems that pose threats to public and environmental safety or the national economy. The owners of safety-critical systems must convince regulators that they have made the best possible efforts to ensure that their systems are safe. The regulator assesses the safety case for the system, which presents evidence and arguments that normal operation of the system will not cause harm to a user.

This evidence is collected during the systems development process. It may include information about hazard analysis and mitigation, test results, static analyses, information about the development processes used, records of review meetings, and so on. It is assembled and organized into a safety case, a detailed presentation of why the system owners and developers believe that a system is safe.

A safety case is a set of documents that includes a description of the system to be certified, information about the processes used to develop the system, and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, Bishop and Bloomfield (Bishop and Bloomfield 1998) define a safety case as:

> *A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment*[†].

The organization and contents of a safety case depend on the type of system that is to be certified and its context of operation. Figure 12.10 shows one possible structure for a safety case, but there are no universal industrial standards in this area. Safety case structures vary, depending on the industry and the maturity of the domain. For example, nuclear safety cases have been required for many years. They are very comprehensive and presented in a way that is familiar to nuclear engineers. However, safety cases for medical devices have been introduced more recently. The case structure is more flexible, and the cases themselves are less detailed than nuclear cases.

A safety case refers to a system as a whole, and, as part of that case, there may be a subsidiary software safety case. When constructing a software safety case, you have to relate software failures to wider system failures and demonstrate either that

[†]Bishop, P., and R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In Proc. Safety-Critical Systems Symposium. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements taken from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments (see Section 12.4.1) that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken, and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

Figure 12.10 Possible contents of a software safety case

these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

Safety cases are large and complex documents, and so they are very expensive to produce and maintain. Because of these high costs, safety-critical system developers have to take the requirements of the safety case into account in the development process:

1. Graydon et al. (Graydon, Knight, and Strunk 2007) argue that the development of a safety case should be tightly integrated with system design and implementation. This means that system design decisions may be influenced by the requirements of the safety case. Design choices that may add significantly to the difficulties and costs of case development can then be avoided.
2. Regulators have their own views on what is acceptable and unacceptable in a safety case. It therefore makes sense for a development team to work with them from early in the development to establish what the regulator expects from the system safety case.

The development of safety cases is expensive because of the costs of the record keeping required as well as the costs of comprehensive system validation and safety assurance processes. System changes and rework also add to the costs of a safety

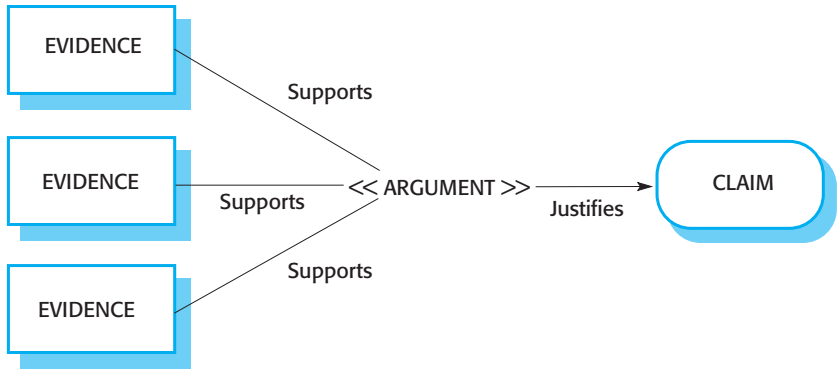


Figure 12.11 Structured arguments

case. When software or hardware changes are made to a system, a large part of the safety case may have to be rewritten to demonstrate that the system safety has not been affected by the change.

12.4.1 Structured arguments

The decision on whether or not a system is operationally safe should be based on logical arguments. These arguments should demonstrate that the evidence presented supports the claims about a system's security and dependability. These claims may be absolute (event X will or will not happen) or probabilistic (the probability of occurrence of event Y is 0.n). An argument links the evidence and the claim. As shown in Figure 12.11, an argument is a relationship between what is thought to be the case (the claim) and a body of evidence that has been collected. The argument essentially explains why the claim, which is an assertion about system security or dependability, can be inferred from the available evidence.

Arguments in a safety case are usually presented as "claim based" arguments. Some claim about system safety is made, and, on the basis of available evidence, an argument is presented as to why that claim holds. For example, the following argument might be used to justify a claim that computations carried out by the control software in an insulin pump will not lead to an overdose of insulin being delivered. Of course, this is a very simplified presentation of the argument. In a real safety case, more detailed references to the evidence would be presented.

Claim: The maximum single dose computed by the insulin pump will not exceed maxDose, where maxDose has been assessed as a safe single dose for a particular patient.

Evidence: Safety argument for insulin pump software control program (covered later in this section).

Evidence: Test datasets for the insulin pump. In 400 tests, which provided complete code coverage, the value of the dose of insulin to be delivered, currentDose, never exceeded maxDose.

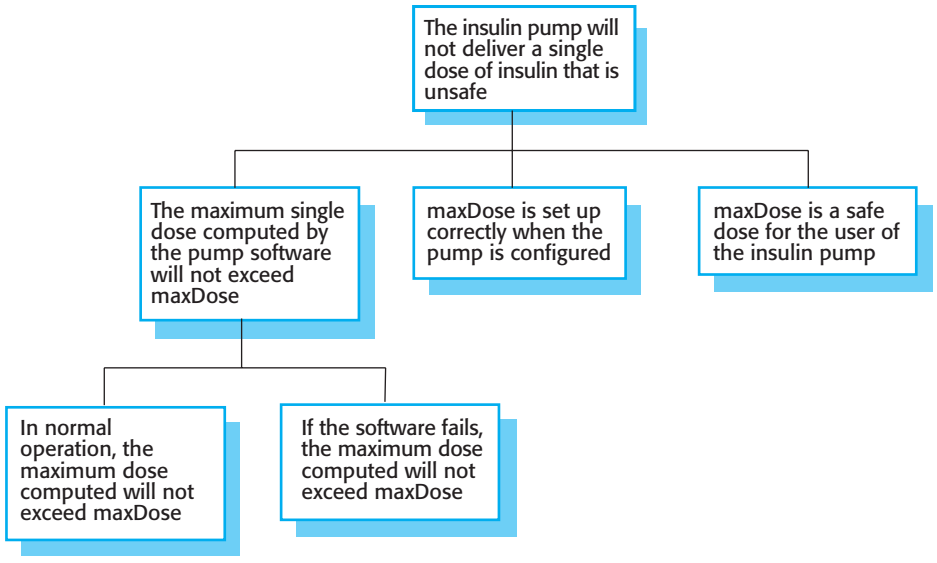


Figure 12.12 A safety claim hierarchy for the insulin pump

Evidence: A static analysis report for the insulin pump control program. The static analysis of the control software revealed no anomalies that affected the value of currentDose, the program variable that holds the dose of insulin to be delivered.

Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed is equal to maxDose.

It is therefore reasonable to assume, with a high level of confidence, that the evidence justifies the claim that the insulin pump will not compute a dose of insulin to be delivered that exceeds the maximum single safe dose.

The evidence presented is both redundant and diverse. The software is checked using several different mechanisms with significant overlap between them. As I discussed in Chapter 10, using redundant and diverse processes increases confidence. If omissions and mistakes are not detected by one validation process, there is a good chance that they will be found by one of the other processes.

There will normally be many claims about the safety of a system, with the validity of one claim often depending on whether or not other claims are valid. Therefore, claims may be organized in a hierarchy. Figure 12.12 shows part of this claim hierarchy for the insulin pump. To demonstrate that a high-level claim is valid, you first have to work through the arguments for lower-level claims. If you can show that each of these lower-level claims is justified, then you may be able to infer that the higher-level claims are justified.

12.4.2 Software safety arguments

A general assumption that underlies work in system safety is that the number of system faults that can lead to safety hazards is significantly less than the total number of faults that may exist in the system. Safety assurance can therefore concentrate on

these faults, which have hazard potential. If it can be demonstrated that these faults cannot occur or, if they occur, that the associated hazard will not result in an accident, then the system is safe. This is the basis of software safety arguments.

Software safety arguments are a type of structured argument which demonstrates that a program meets its safety obligations. In a safety argument, it is not necessary to prove that the program works as intended. It is only necessary to show that program execution cannot result in it reaching a potentially unsafe state. Safety arguments are therefore cheaper to make than correctness arguments. You don't have to consider all program states—you can simply concentrate on states that could lead to a hazard.

Safety arguments demonstrate that, assuming normal execution conditions, a program should be safe. They are usually based on contradiction, where you assume that the system is unsafe and then show that it is impossible to reach an unsafe state. The steps involved in creating a safety argument are:

1. You start by assuming that an unsafe state, which has been identified by the system hazard analysis, can be reached by executing the program.
2. You write a predicate (a logical expression) that defines this unsafe state.
3. You then systematically analyze a system model or the program and show that, for all program paths leading to that state, the terminating condition of these paths, also defined as a predicate, contradicts the unsafe state predicate. If this is the case, you may then claim that the initial assumption of an unsafe state is incorrect.
4. When you have repeated this analysis for all identified hazards, then you have strong evidence that the system is safe.

Safety arguments can be applied at different levels, from requirements through design models to code. At the requirements level, you are trying to demonstrate that there are no missing safety requirements and that the requirements do not make invalid assumptions about the system. At the design level, you might analyze a state model of the system to find unsafe states. At the code level, you consider all of the paths through the safety-critical code to show that the execution of all paths leads to a contradiction.

As an example, consider the code outlined in Figure 12.13, which is a simplified description of part of the implementation of the insulin delivery system. The code computes the dose of insulin to be delivered and then applies some safety checks that this is not an overdose for that patient. Developing a safety argument for this code involves demonstrating that the dose of insulin administered is never greater than the maximum safe level for a single dose. This dose is established for each individual diabetic user in discussions with their medical advisors.

To demonstrate safety, you do not have to prove that the system delivers the "correct" dose, but merely that it never delivers an overdose to the patient. You work on the assumption that `maxDose` is the safe level for that system user.

To construct the safety argument, you identify the predicate that defines the unsafe state, which is that `currentDose > maxDose`. You then demonstrate that all program paths lead to a contradiction of this unsafe assertion. If this is the case, the unsafe condition cannot be true. If you can prove a contradiction, you can be confident that

```
— The insulin dose to be delivered is a function of
— blood sugar level, the previous dose delivered and
— the time of delivery of the previous dose

currentDose = computeInsulin () ;

// Safety check—adjust currentDose if necessary.

// if statement 1
if (previousDose == 0)
{
    if (currentDose > maxDose/2)
       currentDose = maxDose/2 ;
}
else
    if (currentDose > (previousDose * 2) )
       currentDose = previousDose * 2 ;

// if statement 2

if ( currentDose < minimumDose )
   currentDose = 0 ;
else if ( currentDose > maxDose )
   currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Figure 12.13 Insulin dose computation with safety checks

the program will not compute an unsafe dose of insulin. You can structure and present the safety arguments graphically as shown in Figure 12.14.

The safety argument shown in Figure 12.14 presents three possible program paths that lead to the call to the **administerInsulin** method. You have to show that the amount of insulin delivered never exceeds **maxDose**. All possible program paths to **administerInsulin** are considered:

1. Neither branch of if-statement 2 is executed. This can only happen if **currentDose** is outside of the range **minimumDose..maxDose**. The postcondition predicate is therefore:

   currentDose >= minimumDose and currentDose <= maxDose

2. The then-branch of if-statement 2 is executed. In this case, the assignment setting **currentDose** to zero is executed. Therefore, its postcondition predicate is **currentDose = 0**.

3. The else-if-branch of if-statement 2 is executed. In this case, the assignment setting **currentDose to maxDose** is executed. Therefore, after this statement has been executed, we know that the postcondition is **currentDose = maxDose**.

In all three cases, the postcondition predicates contradict the unsafe precondition that **currentDose > maxDose**. As both cannot be true, we can claim that our initial assumption was incorrect, and so the computation is safe.

To construct a structured argument that a program does not make an unsafe computation, you first identify all possible paths through the code that could lead to a potentially

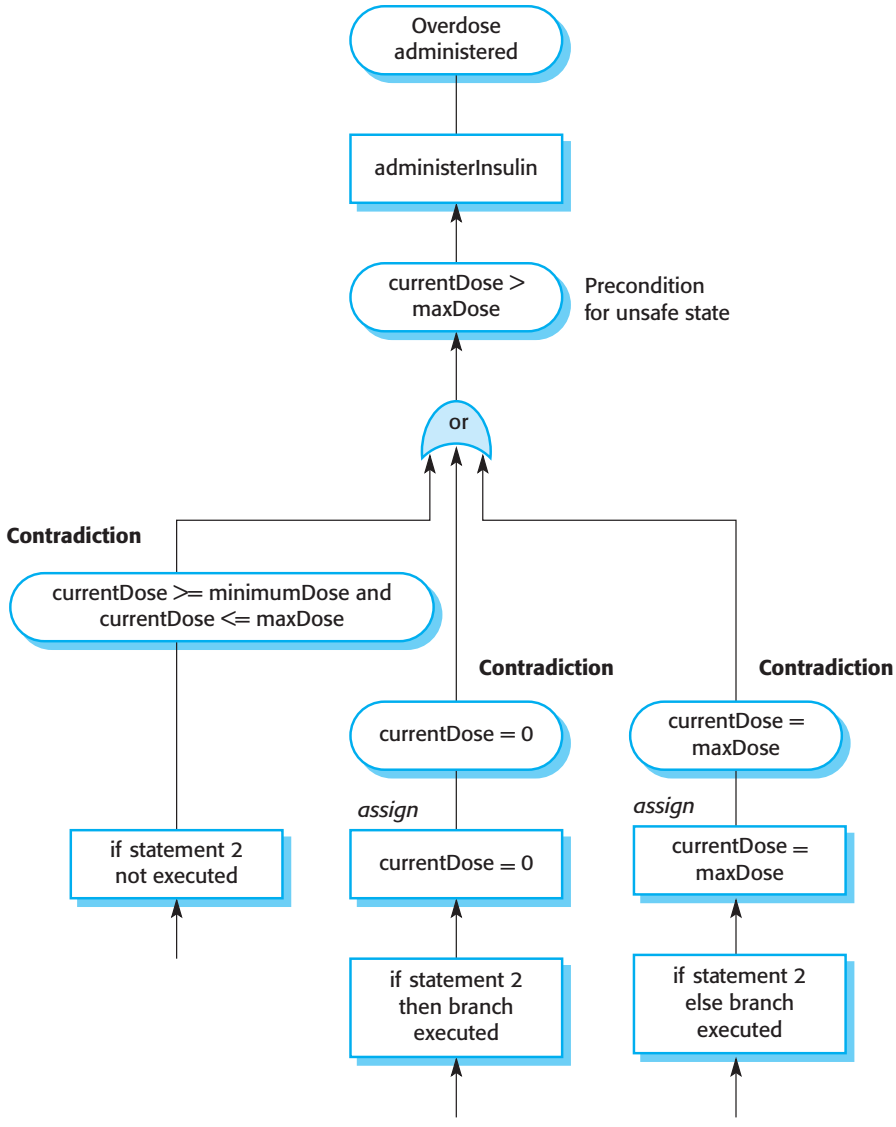


Figure 12.14 Informal safety argument based on demonstrating contradictions

unsafe assignment. You work backwards from the unsafe state and consider the last assignment to all of the state variables on each path leading to this unsafe state. If you can show that none of the values of these variables is unsafe, then you have shown that your initial assumption (that the computation is unsafe) is incorrect.

Working backwards is important because it means that you can ignore all intermediate states apart from the final states that lead to the exit condition for the code. The previous values don't matter to the safety of the system. In this example, all you need be concerned with is the set of possible values of currentDose immediately before the administerInsulin method is executed. You can ignore computations, such as if-statement 1 in Figure 12.13 in the safety argument because their results are overwritten in later program statements.

KEY POINTS

- Safety-critical systems are systems whose failure can lead to human injury or death.
- A hazard-driven approach may be used to understand the safety requirements for safety-critical systems. You identify potential hazards and decompose them (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- It is important to have a well-defined, certified process for safety-critical systems development. The process should include the identification and monitoring of potential hazards.
- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- Safety and dependability cases collect all of the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.

FURTHER READING

Safeware: System Safety and Computers. Although now 20 years old, this book still offers the best and most thorough coverage of safety-critical systems. It is particularly strong in its description of hazard analysis and the derivation of requirements from it. (N. Leveson, Addison-Wesley, 1995).

"Safety-Critical Software." A special edition of *IEEE Software* magazine that focuses on safety-critical systems. It includes papers on model-based development of safety-critical systems, model checking and formal methods. (*IEEE Software*, 30 (3), May/June 2013).

"Constructing Safety Assurance Cases for Medical Devices." This short paper gives a practical example of how a safety case can be created for an analgesic pump. (A. Ray and R. Cleaveland, Proc. Workshop on Assurance Cases for Software-Intensive Systems, San Francisco, 2013) http://dx.doi.org/10.1109/ASSURE.2013.6614270

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

EXERCISES

12.1. Identify six consumer products that are likely to be controlled by safety-critical software systems.

12.2. A software system is to be deployed for a company that has extremely high safety standards and allows for almost no risks, not even minor injuries. How will this affect the look of the risk triangle in Figure 12.3?

12.3. In the insulin pump system, the user has to change the needle and insulin supply at regular intervals and may also change the maximum single dose and the maximum daily dose that may be administered. Suggest three user errors that might occur and propose safety requirements that would avoid these errors resulting in an accident.

12.4. A safety-critical software system for managing roller coasters controls two main components:

- The lock and release of the roller coaster harness which is supposed to keep riders in place as the coaster performs sharp and sudden moves. The roller coaster could not move with any unlocked harnesses.
- The minimum and maximum speeds of the roller coaster as it moves along the various segments of the ride to prevent derailing, given the number of people riding the roller coaster.

Identify three hazards that may arise in this system. For each hazard, suggest a defensive requirement that will reduce the probability that these hazards will result in an accident. Explain why your suggested defense is likely to reduce the risk associated with the hazard.

12.5. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded, or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). There are two critical-safety requirements for this train protection system:

The train shall not enter a segment of track that is signaled with a red light.

The train shall not exceed the specified speed limit for a section of track.

Assuming that the signal status and the speed limit for the track segment are transmitted to on-board software on the train before it enters the track segment, propose five possible functional system requirements for the onboard software that may be generated from the system safety requirements.

12.6. Explain when it may be cost-effective to use formal specification and verification in the development of safety-critical software systems. Why do you think that some critical systems engineers are against the use of formal methods?

12.7. Explain why using model checking is sometimes a more cost-effective approach to verification than verifying a program's correctness against a formal specification.

12.8. List four types of systems that may require software safety cases, explaining why safety cases are required.

12.9. The door lock control mechanism in a nuclear waste storage facility is designed for safe operation. It ensures that entry to the storeroom is only permitted when radiation shields are

```
1       entryCode = lock.getEntryCode () ;
2       if (entryCode == lock.authorizedCode)
3       {
4               shieldStatus = Shield.getStatus ();
5               radiationLevel = RadSensor.get ();
6               if (radiationLevel < dangerLevel)
7                       state = safe;
8               else
9                       state = unsafe;
10              if (shieldStatus == Shield.inPlace() )
11                      state = safe;
12              if (state == safe)
13                      {
14                              Door.locked = false ;
15                              Door.unlock ();
16                      }
17              else
18              {
19                      Door.lock ( );
20                      Door.locked := true ;
21              }
22      }
```

Figure 12.15 Door entry code

in place or when the radiation level in the room falls below some given value (dangerLevel). So:

(i) If remotely controlled radiation shields are in place within a room, an authorized operator may open the door.

(ii) If the radiation level in a room is below a specified value, an authorized operator may open the door.

(iii) An authorized operator is identified by the input of an authorized door entry code.

The code shown in Figure 12.15 controls the door-locking mechanism. Note that the safe state is that entry should not be permitted. Using the approach discussed in this chapter, develop a safety argument for this code. Use the line numbers to refer to specific statements. If you find that the code is unsafe, suggest how it should be modified to make it safe.

12.10. Should software engineers working on the specification and development of safety-related systems be professionally certified or licensed in some way? Explain your reasoning.

REFERENCES

Abrial, J. R. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Ball, T., V. Levin, and S. K. Rajamani. 2011. "A Decade of Software Model Checking with SLAM." *Communications of the ACM* 54 (7) (July 1): 68. doi:10.1145/1965724.1965743.

Behm, P., P. Benoit, A. Faivre, and J-M. Meynadier. 1999. “Meteor: A Successful Application of B in a Large Project.” In *Formal Methods’ 99*, 369–387. Berlin: Springer-Verlag. doi:10.1007/3-540-48119-2_22.

Bishop, P., and R. E. Bloomfield. 1998. “A Methodology for Safety Case Development.” In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. “Model Checking Flight Control Systems: The Airbus Experience.” In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Dehbonei, B., and F. Mejia. 1995. “Formal Development of Safety-Critical Software Systems in Railway Signalling.” In *Applications of Formal Methods*, edited by M. Hinchey and J. P. Bowen, 227–252. London: Prentice-Hall.

Graydon, P. J., J. C. Knight, and E. A. Strunk. 2007. “Assurance Based Development of Critical Systems.” In *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, 347–357. Edinburgh, Scotland. doi:10.1109/DSN.2007.17.

Holzmann, G. J. 2014. “Mars Code.” *Comm ACM* 57 (2): 64–73. doi:10.1145/2560217.2560218.

Jhala, R., and R. Majumdar. 2009. “Software Model Checking.” *Computing Surveys* 41 (4). doi:10.1145/1592434.1592438.

Kwiatkowska, M., G. Norman, and D. Parker. 2011. “PRISM 4.0: Verification of Probabilistic Real-Time Systems.” In *Proc. 23rd Int. Conf. on Computer Aided Verification*, 585–591. Snowbird, UT: Springer-Verlag. doi:10.1007/978-3-642-22110-1_47.

Leveson, N. G., S. S. Cha, and T. J. Shimeall. 1991. “Safety Verification of Ada Programs Using Software Fault Trees.” *IEEE Softwar*e 8 (4): 48–59. doi:10.1109/52.300036.

Lopes, R., D. Vicente, and N. Silva. 2009. “Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification.” In *Proceedings of DASIA 2009 Data Systems in Aerospace*. Noordwijk, Netherlands: European Space Agency.

Lutz, R. R. 1993. “Analysing Software Requirements Errors in Safety-Critical Embedded Systems.” In *RE’ 93*, 126–133. San Diego, CA: IEEE. doi:0.1109/ISRE.1993.324825.

Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate. 2013. “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience.” *IEEE Software* 30 (3) (May 1): 50–57. doi:10.1109/MS.2013.43.

Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.

Regan, P., and S. Hamilton. 2004. “NASA’s Mission Reliable.” *IEEE Computer* 37 (1): 59–68. doi:10.1109/MC.2004.1260727.

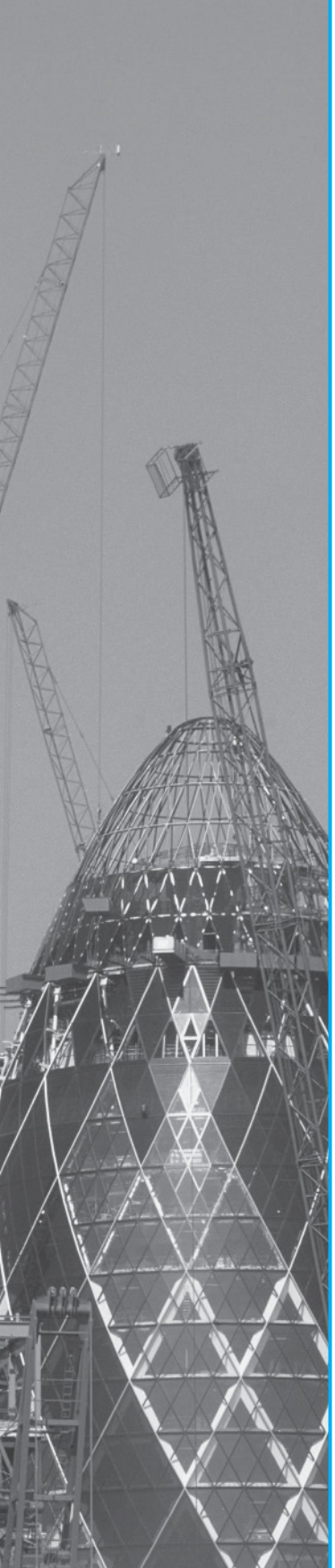
Schneider, S. 1999. *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons.

Souyris, J., V. Weils, D. Delmas, and H. Delseny. 2009. "Formal Verification of Avionics Software Products." In *Formal Methods' 09: Proceedings of the 2nd World Congress on Formal Methods*, 532–546. Springer-Verlag. doi:10.1007/978-3-642-05089-3_34.

Storey, N. 1996. *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Veras, P. C., E. Villani, A. M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. 2010. "Errors in Space Software Requirements: A Field Study and Application Scenarios." In *21st Int. Symp. on Software Reliability Engineering*. San Jose, CA. doi:10.1109/ISSRE.2010.37.

Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Trans. on Software Eng*. 32 (4): 240–253. doi:10.1109/TSE.2006.38.



13
Security engineering

Objectives

The objective of this chapter is to introduce security issues that you should consider when you are developing application systems. When you have read this chapter, you will:

- understand the importance of security engineering and the difference between application security and infrastructure security;
- know how a risk-based approach can be used to derive security requirements and analyze system designs;
- know of software architectural patterns and design guidelines for secure systems engineering;
- understand why security testing and assurance is difficult and expensive.

Contents

13.1 Security and dependability
13.2 Security and organizations
13.3 Security requirements
13.4 Secure systems design
13.5 Security testing and assurance

The widespread adoption of the Internet in the 1990s introduced a new challenge for software engineers—designing and implementing systems that were secure. As more and more systems were connected to the Internet, a variety of different external attacks were devised to threaten these systems. The problems of producing dependable systems were hugely increased. Systems engineers had to consider threats from malicious and technically skilled attackers as well as problems resulting from accidental mistakes in the development process.

It is now essential to design systems to withstand external attacks and to recover from such attacks. Without security precautions, attackers will inevitably compromise a networked system. They may misuse the system hardware, steal confidential data, or disrupt the services offered by the system.

You have to take three security dimensions into account in secure systems engineering:

1. *Confidentiality* Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information. For example, the theft of credit card data from an e-commerce system is a confidentiality problem.
2. *Integrity* Information in a system may be damaged or corrupted, making it unusual or unreliable. For example, a worm that deletes data in a system is an integrity problem.
3. *Availability* Access to a system or its data that is normally available may not be possible. A denial-of-service attack that overloads a server is an example of a situation where the system availability is compromised.

These dimensions are closely related. If an attack makes the system unavailable, then you will not be able to update information that changes with time. This means that the integrity of the system may be compromised. If an attack succeeds and the integrity of the system is compromised, then it may have to be taken down to repair the problem. Therefore, the availability of the system is reduced.

From an organizational perspective, security has to be considered at three levels:

1. *Infrastructure security,* which is concerned with maintaining the security of all systems and networks that provide an infrastructure and a set of shared services to the organization.
2. *Application security,* which is concerned with the security of individual application systems or related groups of systems.
3. *Operational security,* which is concerned with the secure operation and use of the organization's systems.

Figure 13.1 is a diagram of an application system stack that shows how an application system relies on an infrastructure of other systems in its operation. The lower levels of the infrastructure are hardware, but the software infrastructure for application systems may include:

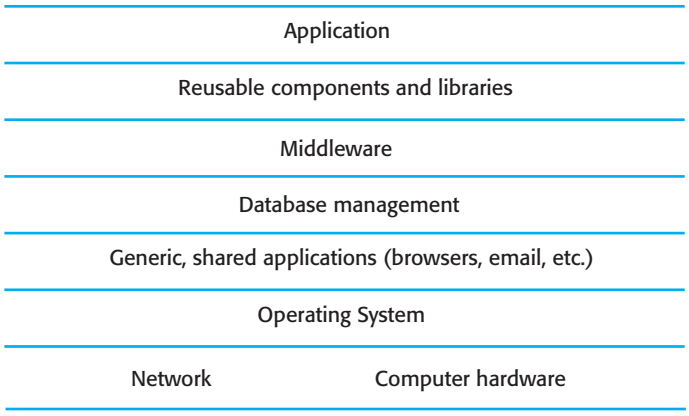


Figure 13.1 System layers where security may be compromised

- an operating system platform, such as Linux or Windows;
- other generic applications that run on that system, such as web browsers and email clients;
- a database management system;
- middleware that supports distributed computing and database access; and
- libraries of reusable components that are used by the application software.

Network systems are software controlled, and networks may be subject to security threats where an attacker intercepts and reads or changes network packets. However, this requires specialized equipment, so the majority of security attacks are on the software infrastructure of systems. Attackers focus on software infrastructures because infrastructure components, such as web browsers, are universally available. Attackers can probe these systems for weaknesses and share information about vulnerabilities that they have discovered. As many people use the same software, attacks have wide applicability.

Infrastructure security is primarily a system management problem, where system managers configure the infrastructure to resist attacks. System security management includes a range of activities such as user and permission management, system software deployment and maintenance, and attack monitoring, detection, and recovery:

1. User and permission management involves adding and removing users from the system, ensuring that appropriate user authentication mechanisms are in place, and setting up the permissions in the system so that users only have access to the resources they need.
2. System software deployment and maintenance involves installing system software and middleware and configuring these properly so that security vulnerabilities are avoided. It also involves updating this software regularly with new versions or patches, which repair security problems that have been discovered.

3. Attack monitoring, detection, and recovery involves monitoring the system for unauthorized access, detecting and putting in place strategies for resisting attacks, and organizing backups of programs and data so that normal operation can be resumed after an external attack.

Operational security is primarily a human and social issue. It focuses on ensuring that the people using the system do not behave in such a way that system security is compromised. For example, users may leave themselves logged on to a system while it is unattended. An attacker can then easily get access to the system. Users often behave in an insecure way to help them do their jobs more effectively, and they have good reason to behave in an insecure way. A challenge for operational security is to raise awareness of security issues and to find the right balance between security and system effectiveness.

The term *cybersecurity* is now commonly used in discussions of system security. Cybersecurity is a very wide-ranging term that covers all aspects of the protection of citizens, businesses, and critical infrastructures from threats that arise from their use of computers and the Internet. Its scope includes all system levels from hardware and networks through application systems to mobile devices that may be used to access these systems. I discuss general cybersecurity issues, including infrastructure security, in Chapter 14, which covers resilience engineering.

In this chapter, I focus on issues of application security engineering—security requirements, design for security, and security testing. I don't cover general security techniques that may be used, such as encryption, and access control mechanisms or attack vectors, such as viruses and worms. General textbooks on computer security (Pfleeger and Pfleeger 2007; Anderson 2008; Stallings and Brown 2012) discuss these techniques in detail.

13.1 Security and dependability

Security is a system attribute that reflects the ability of the system to protect itself from malicious internal or external attacks. These external attacks are possible because most computers and mobile devices are networked and are therefore accessible by outsiders. Examples of attacks might be the installation of viruses and Trojan horses, unauthorized use of system services, or unauthorized modification of a system or its data.

If you really want a system to be as secure as possible, it is best not to connect it to the Internet. Then, your security problems are limited to ensuring that authorized users do not abuse the system and to controlling the use of devices such as USB drives. In practice, however, networked access provides huge benefits for most systems, so disconnecting from the Internet is not a viable security option.

For some systems, security is the most important system dependability attribute. Military systems, systems for electronic commerce, and systems that involve the processing and interchange of confidential information must be designed so that

Term	Definition
Asset	Something of value that has to be protected. The asset may be the software system itself or the data used by that system.
Attack	An exploitation of a system's vulnerability where an attacker has the goal of causing some damage to a system asset or assets. Attacks may be from outside the system (external attacks) or from authorized insiders (insider attacks).
Control	A protective measure that reduces a system's vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system.
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
Threat	Circumstances that have potential to cause loss or harm. You can think of a threat as a system vulnerability that is subjected to an attack.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.

Figure 13.2 Security terminology

Unauthorized access to the Mentcare system

Clinic staff log on to the Mentcare system using a username and password. The system requires passwords to be at least eight letters long but allows any password to be set without further checking. A criminal finds out that a well-paid sports star is receiving treatment for mental health problems. He would like to gain illegal access to information in this system so that he can blackmail the star.

By posing as a concerned relative and talking with the nurses in the mental health clinic, he discovers how to access the system and personal information about the nurses and their families. By checking name badges, he discovers the names of some of the people allowed access. He then attempts to log on to the system by using these names and systematically guessing possible passwords, such as the names of the nurses' children.

Figure 13.3 A security story for the Mentcare system

they achieve a high level of security. If an airline reservation system is unavailable, for example, this causes inconvenience and some delays in issuing tickets. However, if the system is insecure, then an attacker could delete all bookings and it would be practically impossible for normal airline operations to continue.

As with other aspects of dependability, a specialized terminology is associated with security (Pfleeger and Pfleeger 2007). This terminology is explained in Figure 13.2. Figure 13.3 is a security story from the Mentcare system that I use to illustrate some of these terms. Figure 13.4 takes the security concepts defined in Figure 13.2 and shows how they apply to this security story.

System vulnerabilities may arise because of requirements, design, or implementation problems, or they may stem from human, social, or organizational failings. People may choose easy-to-guess passwords or write down their passwords in places where they can be found. System administrators make errors in setting up access control or configuration files, and users don't install or use protection software. However, we cannot simply class these problems as human errors. User mistakes or omissions often reflect poor systems design decisions that require, for example, frequent password changes (so that users write down their passwords) or complex configuration mechanisms.

Term	Example
Asset	The record of each patient who is receiving or has received treatment.
Attack	An impersonation of an authorized user.
Control	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Threat	An unauthorized user will gain access to the system by guessing the credentials (login name and password) of an authorized user.
Vulnerability	Authentication is based on a password system that does not require strong passwords. Users can then set easily guessable passwords.

Figure 13.4 Examples of security terminology

Four types of security threats may arise:

1. Interception threats that allow an attacker to gain access to an asset. So, a possible threat to the Mentcare system might be a situation where an attacker gains access to the records of an individual patient.
2. Interruption threats that allow an attacker to make part of the system unavailable. Therefore, a possible threat might be a denial-of-service attack on a system database server.
3. Modification threats that allow an attacker to tamper with a system asset. In the Mentcare system, a modification threat would be where an attacker alters or destroys a patient record.
4. Fabrication threats that allow an attacker to insert false information into a system. This is perhaps not a credible threat in the Mentcare system but would certainly be a threat in a banking system, where false transactions might be added to the system that transfers money to the perpetrator's bank account.

The controls that you might put in place to enhance system security are based on the fundamental notions of avoidance, detection, and recovery:

1. *Vulnerability avoidance* Controls that are intended to ensure that attacks are unsuccessful. The strategy here is to design the system so that security problems are avoided. For example, sensitive military systems are not connected to the Internet so that external access is more difficult. You should also think of encryption as a control based on avoidance. Any unauthorized access to encrypted data means that the attacker cannot read the encrypted data. It is expensive and time consuming to crack strong encryption.
2. *Attack detection and neutralization* Controls that are intended to detect and repel attacks. These controls involve including functionality in a system that monitors its operation and checks for unusual patterns of activity. If these

attacks are detected, then action may be taken, such as shutting down parts of the system or restricting access to certain users.

3. *Exposure limitation and recovery* Controls that support recovery from problems. These can range from automated backup strategies and information "mirroring" through to insurance policies that cover the costs associated with a successful attack on the system.

Security is closely related to the other dependability attributes of reliability, availability, safety, and resilience:

1. *Security and reliability* If a system is attacked and the system or its data are corrupted as a consequence of that attack, then this may induce system failures that compromise the reliability of the system.

   Errors in the development of a system can lead to security loopholes. If a system does not reject unexpected inputs or if array bounds are not checked, then attackers can exploit these weaknesses to gain access to the system. For example, failure to check the validity of an input may mean that an attacker can inject and execute malicious code.

2. *Security and availability* A common attack on a web-based system is a denial-of-service attack, where a web server is flooded with service requests from a range of different sources. The aim of this attack is to make the system unavailable. A variant of this attack is where a profitable site is threatened with this type of attack unless a ransom is paid to the attackers.

3. *Security and safety* Again, the key problem is an attack that corrupts the system or its data. Safety checks are based on the assumption that we can analyze the source code of safety-critical software and that the executing code is a completely accurate translation of that source code. If this is not the case, because an attacker has changed the executing code, safety-related failures may be induced and the safety case made for the software is invalid.

   Like safety, we cannot assign a numeric value to the security of a system, nor can we exhaustively test the system for security. Both safety and security can be thought of as "negative" or "shall not" characteristics in that they are concerned with things that should not happen. As we can never prove a negative, we can never prove that a system is safe or secure.

4. *Security and resilience* Resilience, covered in Chapter 14, is a system characteristic that reflects its ability to resist and recover from damaging events. The most probable damaging event on networked software systems is a cyberattack of some kind, so most of the work now done in resilience is aimed at deterring, detecting, and recovering from such attacks.

Security has to be maintained if we are to create reliable, available, and safe software-intensive systems. It is not an add-on, which can be added later but has to be considered at all stages of the development life cycle from early requirements to system operation.

13.2 Security and organizations

Building secure systems is expensive and uncertain. It is impossible to predict the costs of a security failure, so companies and other organizations find it difficult to judge how much they should spend on system security. In this respect, security and safety are different. There are laws that govern workplace and operator safety, and developers of safety-critical systems have to comply with these irrespective of the costs. They may be subject to legal action if they use an unsafe system. However, unless a security failure discloses personal information, there are no laws that prevent an insecure system from being deployed.

Companies assess the risks and losses that may arise from certain types of attacks on system assets. They may then decide that it is cheaper to accept these risks rather than build a secure system that can deter or repel the external attacks. Credit card companies apply this approach to fraud prevention. It is usually possible to introduce new technology to reduce credit card fraud. However, it is often cheaper for these companies to compensate users for their losses due to fraud than to buy and deploy fraud-reduction technology.

Security risk management is therefore a business rather than a technical issue. It has to take into account the financial and reputational losses from a successful system attack as well as the costs of security procedures and technologies that may reduce these losses. For risk management to be effective, organizations should have a documented information security policy that sets out:

1. *The assets that must be protected* It does not necessarily make sense to apply stringent security procedures to all organizational assets. Many assets are not confidential, and a company can improve its image by making these assets freely available. The costs of maintaining the security of information that is in the public domain are much less than the costs of keeping confidential information secure.

2. *The level of protection that is required for different types of assets* Not all assets need the same level of protection. In some cases (e.g., for sensitive personal information), a high level of security is required; for other information, the consequences of loss may be minor, so a lower level of security is adequate. Therefore, some information may be made available to any authorized and logged-in user; other information may be much more sensitive and only available to users in certain roles or positions of responsibility.

3. *The responsibilities of individual users, managers, and the organization* The security policy should set out what is expected of users—for example, use strong passwords, log out of computers, and lock offices. It also defines what users can expect from the company, such as backup and information-archiving services, and equipment provision.

4. *Existing security procedures and technologies that should be maintained* For reasons of practicality and cost, it may be essential to continue to use existing approaches to security even where these have known limitations. For example,

a company may require the use of a login name/password for authentication, simply because other approaches are likely to be rejected by users.

Security policies often set out general information access strategies that should apply across the organization. For example, an access strategy may be based on the clearance or seniority of the person accessing the information. Therefore, a military security policy may state: "Readers may only examine documents whose classification is the same as or below the reader's vetting level." This means that if a reader has been vetted to a "secret" level, he or she may access documents that are classed as secret, confidential, or open but not documents classed as top secret.

The point of security policies is to inform everyone in an organization about security, so these should not be long and detailed technical documents. From a security engineering perspective, the security policy defines, in broad terms, the security goals of the organization. The security engineering process is concerned with implementing these goals.

13.2.1 Security risk assessment

Security risk assessment and management are organizational activities that focus on identifying and understanding the risks to information assets (systems and data) in the organization. In principle, an individual risk assessment should be carried out for all assets; in practice, however, this may be impractical if a large number of existing systems and databases need to be assessed. In those situations, a generic assessment may be applied to all of them. However, individual risk assessments should be carried out for new systems.

Risk assessment and management is an organizational activity rather than a technical activity that is part of the software development life cycle. The reason for this is that some types of attack are not technology-based but rather rely on weaknesses in more general organizational security. For example, an attacker may gain access to equipment by pretending to be an accredited engineer. If an organization has a process to check with the equipment supplier that an engineer's visit is planned, this can deter this type of attack. This approach is much simpler than trying to address the problem using a technological solution.

When a new system is to be developed, security risk assessment and management should be a continuing process throughout the development life cycle from initial specification to operational use. The stages of risk assessment are:

1. *Preliminary risk assessment* The aim of this initial risk assessment is to identify generic risks that are applicable to the system and to decide if an adequate level of security can be achieved at a reasonable cost. At this stage, decisions on the detailed system requirements, the system design, or the implementation technology have not been made. You don't know of potential technology vulnerabilities or the controls that are included in reused system components or middleware. The risk assessment should therefore focus on the identification and analysis of high-level risks to the system. The outcomes of the risk assessment process are used to help identify security requirements.

2. *Design risk assessment* This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of the assessment may lead to changes to the security requirements and the addition of new requirements. Known and potential vulnerabilities are identified, and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.

3. *Operational risk assessment* This risk assessment process focuses on the use of the system and the possible risks that can arise. For example, when a system is used in an environment where interruptions are common, a security risk is that a logged-in user leaves his or her computer unattended to deal with a problem. To counter this risk, a timeout requirement may be specified so that a user is automatically logged out after a period of inactivity.

   Operational risk assessment should continue after a system has been installed to take account of how the system is used and proposals for new and changed requirements. Assumptions about the operating requirement made when the system was specified may be incorrect. Organizational changes may mean that the system is used in different ways from those originally planned. These changes lead to new security requirements that have to be implemented as the system evolves.

13.3 Security requirements

The specification of security requirements for systems has much in common with the specification of safety requirements. You cannot specify safety or security requirements as probabilities. Like safety requirements, security requirements are often "shall not" requirements that define unacceptable system behavior rather than required system functionality.

However, security is a more challenging problem than safety, for a number of reasons:

1. When considering safety, you can assume that the environment in which the system is installed is not hostile. No one is trying to cause a safety-related incident. When considering security, you have to assume that attacks on the system are deliberate and that the attacker may have knowledge of system weaknesses.

2. When system failures occur that pose a risk to safety, you look for the errors or omissions that have caused the failure. When deliberate attacks cause system failure, finding the root cause may be more difficult as the attacker may try to conceal the cause of the failure.

3. It is usually acceptable to shut down a system or to degrade system services to avoid a safety-related failure. However, attacks on a system may be denial-of-service attacks, which are intended to compromise system availability. Shutting down the system means that the attack has been successful.

4. Safety-related events are accidental and are not created by an intelligent adversary. An attacker can probe a system's defenses in a series of attacks, modifying the attacks as he or she learns more about the system and its responses.

These distinctions mean that security requirements have to be more extensive than safety requirements. Safety requirements lead to the generation of functional system requirements that provide protection against events and faults that could cause safety-related failures. These requirements are mostly concerned with checking for problems and taking actions if these problems occur. By contrast, many types of security requirements cover the different threats faced by a system.

Firesmith (Firesmith 2003) identified 10 types of security requirements that may be included in a system specification:

1. Identification requirements specify whether or not a system should identify its users before interacting with them.
2. Authentication requirements specify how users are identified.
3. Authorization requirements specify the privileges and access permissions of identified users.
4. Immunity requirements specify how a system should protect itself against viruses, worms, and similar threats.
5. Integrity requirements specify how data corruption can be avoided.
6. Intrusion detection requirements specify what mechanisms should be used to detect attacks on the system.
7. Nonrepudiation requirements specify that a party in a transaction cannot deny its involvement in that transaction.
8. Privacy requirements specify how data privacy is to be maintained.
9. Security auditing requirements specify how system use can be audited and checked.
10. System maintenance security requirements specify how an application can prevent authorized changes from accidentally defeating its security mechanisms.

Of course, you will not see all of these types of security requirements in every system. The particular requirements depend on the type of system, the situation of use, and the expected users.

Preliminary risk assessment and analysis aim to identify the generic security risks for a system and its associated data. This risk assessment is an important input to the security requirements engineering process. Security requirements can be proposed to support the general risk management strategies of avoidance, detection and mitigation.

1. Risk avoidance requirements set out the risks that should be avoided by designing the system so that these risks simply cannot arise.

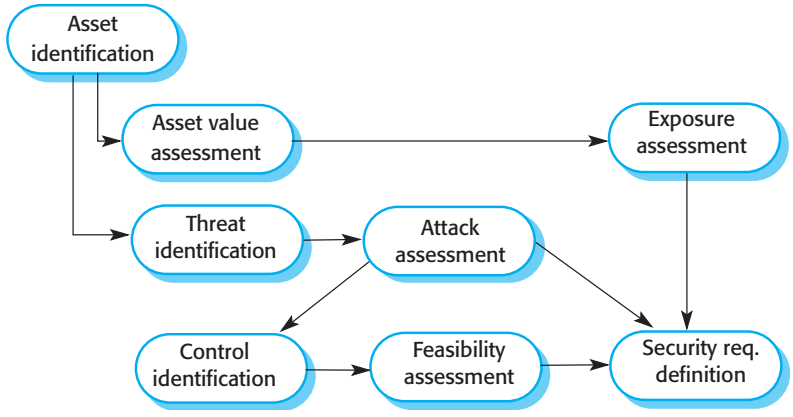


Figure 13.5 The preliminary risk assessment process for security requirements

2. Risk detection requirements define mechanisms that identify the risk if it arises and neutralize the risk before losses occur.
3. Risk mitigation requirements set out how the system should be designed so that it can recover from and restore system assets after some loss has occurred.

A risk-driven security requirements process is shown in Figure 13.5. The process stages are:

1. *Asset identification,* where the system assets that may require protection are identified. The system itself or particular system functions may be identified as assets as well as the data associated with the system.
2. *Asset value assessment,* where you estimate the value of the identified assets.
3. *Exposure assessment,* where you assess the potential losses associated with each asset. This process should take into account direct losses such as the theft of information, the costs of recovery, and the possible loss of reputation.
4. *Threat identification,* where you identify the threats to system assets.
5. *Attack assessment,* where you decompose each threat into attacks that might be made on the system and the possible ways in which these attacks may occur. You may use attack trees (Schneier 1999) to analyze the possible attacks. These are similar to fault trees, (Chapter 12) as you start with a threat at the root of the tree and then identify possible causal attacks and how these might be made.
6. *Control identification,* where you propose the controls that might be put in place to protect an asset. The controls are the technical mechanisms, such as encryption, that you can use to protect assets.
7. *Feasibility assessment,* where you assess the technical feasibility and the costs of the proposed controls. It is not worth having expensive controls to protect assets that don't have a high value.

Asset	Value	Exposure
The information system	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
The patient database	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
An individual patient record	Normally low, although may be high for specific high-profile patients	Low direct losses but possible loss of reputation.

Figure 13.6 Asset analysis in a preliminary risk assessment report for the Mentcare system

8. *Security requirements definition,* where knowledge of the exposure, threats, and control assessments is used to derive system security requirements. These requirements may apply to the system infrastructure or the application system.

The Mentcare patient management system is a security-critical system. Figures 13.6 and 13.7 are fragments of a report that documents the risk analysis of that software system. Figure 13.6 is an asset analysis that describes the assets in the system and their value. Figure 13.7 shows some of the threats that a system may face.

Once a preliminary risk assessment has been completed, then requirements can be proposed that aim to avoid, detect, and mitigate risks to the system. However, creating these requirements is not a formulaic or automated process. It requires inputs from both engineers and domain experts to suggest requirements based on their understanding of the risk analysis and the functional requirements of the software system. Some examples of the Mentcare system security requirements and associated risks are:

1. Patient information shall be downloaded, at the start of a clinic session, from the database to a secure area on the system client.

   Risk: Damage from denial-of-service attack. Maintaining local copies means that access is still possible.

2. All patient information on the system client shall be encrypted.

   Risk: External access to patient records. If data is encrypted, then attacker must have access to the encryption key to discover patient information.

3. Patient information shall be uploaded to the database when a clinic session is over and deleted from the client computer.

   Risk: External access to patience records through stolen laptop.

4. A log of all changes made to the system database and the initiator of these changes shall be maintained on a separate computer from the database server.

   Risk: Insider or external attacks that corrupt current data. A log should allow up-to-date records to be re-created from a backup.

Threat	Probability	Control	Feasibility
An unauthorized user gains access as system manager and makes system unavailable	Low	Only allow system management from specific locations that are physically secure.	Low cost of implementation, but care must be taken with key distribution and to ensure that keys are available in the event of an emergency.
An unauthorized user gains access as system user to confidential information	High	Require all users to authenticate themselves using a biometric mechanism.	Technically feasible but high- cost solution. Possible user resistance.
		Log all changes to patient information to track system usage.	Simple and transparent to implement and also supports recovery.

Figure 13.7 Threat and control analysis in a preliminary risk assessment report

The first two requirements are related—patient information is downloaded to a local machine, so that consultations may continue if the patient database server is attacked or becomes unavailable. However, this information must be deleted so that later users of the client computer cannot access the information. The fourth requirement is a recovery and auditing requirement. It means that changes can be recovered by replaying the change log and that it is possible to discover who has made the changes. This accountability discourages misuse of the system by authorized staff.

13.3.1 Misuse cases

The derivation of security requirements from a risk analysis is a creative process involving engineers and domain experts. One approach that has been developed to support this process for users of the UML is the idea of misuse cases (Sindre and Opdahl 2005). Misuse cases are scenarios that represent malicious interactions with a system. You can use these scenarios to discuss and identify possible threats and, therefore also determine the system's security requirements. They can be used alongside use cases when deriving the system requirements (Chapters 4 and 5).

Misuse cases are associated with use case instances and represent threats or attacks associated with these use cases. They may be included in a use case diagram but should also have a more complete and detailed textual description. In Figure 13.8, I have taken the use cases for a medical receptionist using the Mentcare system and have added misuse cases. These are normally represented as black ellipses.

As with use cases, misuse cases can be described in several ways. I think that it is most helpful to describe them as a supplement to the original use case description. I also think it is best to have a flexible format for misuse cases as different types of attack have to be described in different ways. Figure 13.9 shows the original description of the Transfer Data use case (Figure 5.4), with the addition of a misuse case description.

The problem with misuse cases mirrors the general problem of use cases, which is that interactions between end-users and a system do not capture all of the system

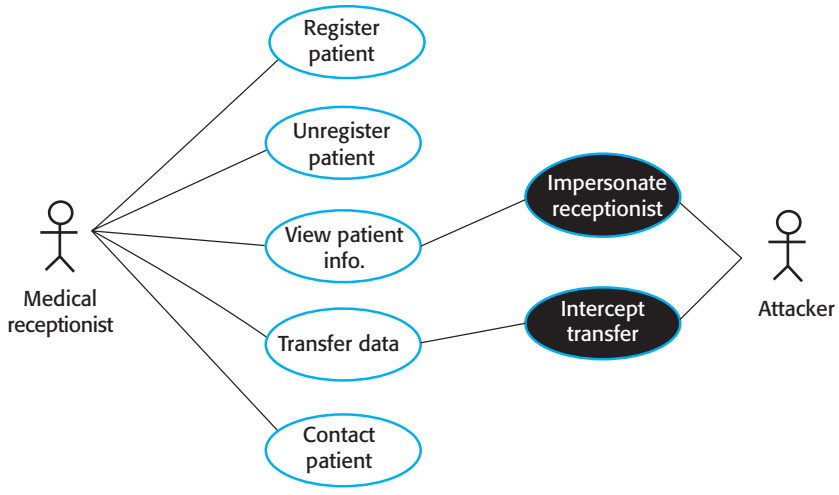


Figure 13.8 Misuse cases

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Mentcare system: Intercept transfer (Misuse case)	
Actors	Medical receptionist, Patient records system (PRS), Attacker
Description	A receptionist transfers data from his or her PC to the Mentcare system on the server. An attacker intercepts the data transfer and takes a copy of that data.
Data (assets)	Patient's personal information, treatment summary
Attacks	A network monitor is added to the system, and packets from the receptionist to the server are intercepted. A spoof server is set up between the receptionist and the database server so that receptionist believes they are interacting with the real system.
Mitigations	All networking equipment must be maintained in a locked room. Engineers accessing the equipment must be accredited. All data transfers between the client and server must be encrypted. Certificate-based client–server communication must be used.
Requirements	All communications between the client and the server must use the Secure Socket Layer (SSL). The https protocol uses certificate-based authentication and encryption.

Figure 13.9 Misuse case descriptions

requirements. Misuse cases can be used as part of the security requirements engineering process, but you also need to consider risks that are associated with system stakeholders who do not interact directly with the system.

13.4 Secure systems design

It is very difficult to add security to a system after it has been implemented. Therefore, you need to take security issues into account during the systems design process and make design choices that enhance the security of a system. In this section, I focus on two application-independent issues relevant to secure systems design:

1. *Architectural design*—how do architectural design decisions affect the security of a system?
2. *Good practice*—what is accepted good practice when designing secure systems?

Of course, these are not the only design issues that are important for security. Every application is different, and security design also has to take into account the purpose, criticality, and operational environment of the application. For example, if you are designing a military system, you need to adopt their security classification model (secret, top secret, etc.) If you are designing a system that maintains personal information, you may have to take into account data protection legislation that places restrictions on how data is managed.

Using redundancy and diversity, which is essential for dependability, may mean that a system can resist and recover from attacks that target specific design or implementation characteristics. Mechanisms to support a high level of availability may help the system to recover from denial-of-service attacks, where the aim of an attacker is to bring down the system and stop it from working properly.

Designing a system to be secure inevitably involves compromises. It is usually possible to design multiple security measures into a system that will reduce the chances of a successful attack. However, these security measures may require additional computation and so affect the overall performance of the system. For example, you can reduce the chances of confidential information being disclosed by encrypting that information. However, this means that users of the information have to wait for it to be decrypted, which may slow down their work.

There are also tensions between security and usability—another emergent system property. Security measures sometimes require the user to remember and provide additional information (e.g., multiple passwords). However, sometimes users forget this information, so the additional security means that they can't use the system.

System designers have to find a balance between security, performance, and usability. This depends on the type of system being developed, the expectations of its users, and its operational environment. For example, in a military system, users are familiar with high-security systems and so accept and follow processes that require frequent checks. In a system for stock trading, where speed is essential, interruptions of operation for security checks would be completely unacceptable.



Denial-of-service attacks

Denial-of-service attacks attempt to bring down a networked system by bombarding it with a huge number of service requests, usually from hundreds of attacking systems. These place a load on the system for which it was not designed and they exclude legitimate requests for system service. Consequently, the system may become unavailable either because it crashes with the heavy load or has to be taken offline by system managers to stop the flow of requests.

http://software-engineering-book.com/web/denial-of-service/

13.4.1 Design risk assessment

Security risk assessment during requirements engineering identifies a set of high-level security requirements for a system. However, as the system is designed and implemented, architectural and technology decisions made during the system design process influence the security of a system. These decisions generate new design requirements and may mean that existing requirements have to change.

System design and the assessment of design-related risks are interleaved processes (Figure 13.10). Preliminary design decisions are made, and the risks associated with these decisions are assessed. This assessment may lead to new requirements to mitigate the risks that have been identified or design changes to reduce these risks. As the system design evolves and is developed in more detail, the risks are reassessed and the results are fed back to the system designers. The design risk assessment process ends when the design is complete and the remaining risks are acceptable.

When assessing risks during design and implementation, you have more information about what needs to be protected, and you also will know something about the vulnerabilities in the system. Some of these vulnerabilities will be inherent in the design choices made. For example, an inherent vulnerability in password-based authentication is that an authorized user reveals their password to an unauthorized user. So, if password-based authentication is used, the risk assessment process may suggest new requirements to mitigate the risk. For example, there may be a requirement for multifactor authentication where users must authenticate themselves using some personal knowledge as well as a password.

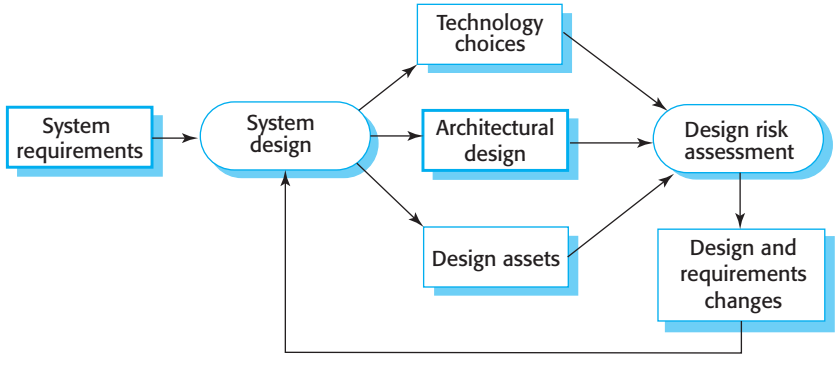


Figure 13.10 Interleaved design and risk assessment

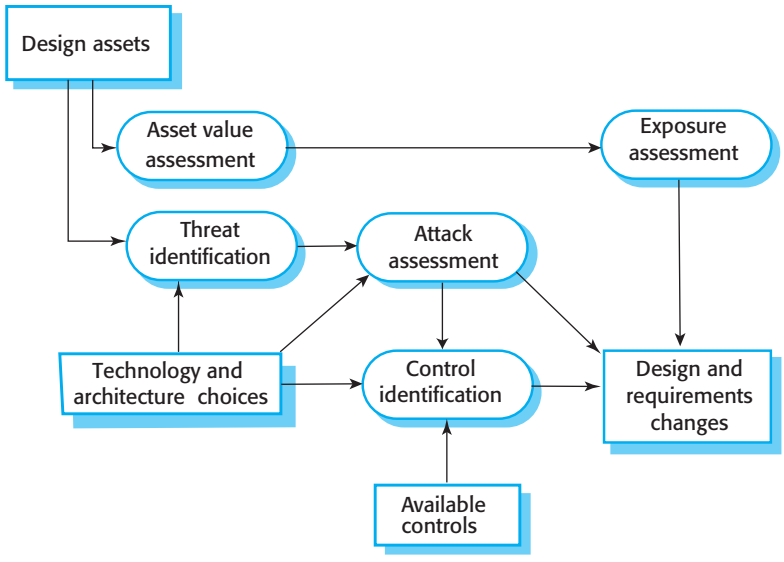


Figure 13.11 Design risk assessment

Figure 13.11 is a model of the design risk assessment process. The key difference between preliminary risk analysis and design risk assessment is that, at the design stage, you now have information about information representation and distribution and the database organization for the high-level assets that have to be protected. You also know about important design decisions such as the software to be reused, infrastructure controls and protection, and so forth. Based on this information, your assessment can identify changes to the security requirements and the system design to provide additional protection for the important system assets.

Two examples from the Mentcare system illustrate how protection requirements are influenced by decisions on information representation and distribution:

1. You may make a design decision to separate personal patient information and information (design assets) about treatments received, with a key linking these records. The treatment information is technical and so much less sensitive than the personal patient information. If the key is protected, then an attacker will only be able to access routine information, without being able to link this to an individual patient.

2. Assume that, at the beginning of a session, a design decision is made to copy patient records to a local client system. This allows work to continue if the server is unavailable. It makes it possible for a healthcare worker to access patient records from a laptop, even if no network connection is available. However, you now have two sets of records to protect and the client copies are subject to additional risks, such as theft of the laptop computer. You therefore have to think about what controls should be used to reduce risk. You may therefore include a requirement that client records held on laptops or other personal computers may have to be encrypted.

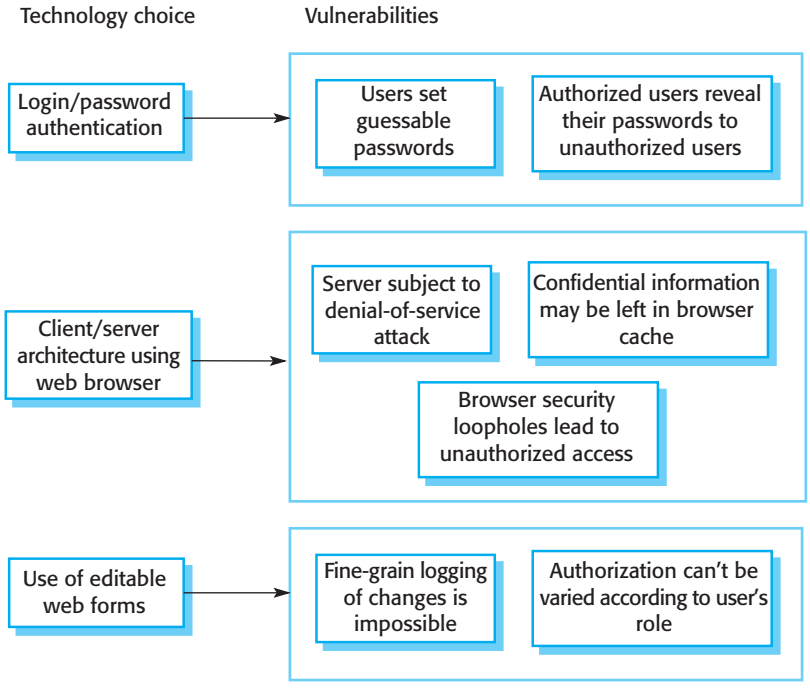


Figure 13.12 Vulnerabilities associated with technology choices

To illustrate how decisions on development technologies influence security, assume that the health care provider has decided to build a Mentcare system using an off-the-shelf information system for maintaining patient records. This system has to be configured for each type of clinic in which it is used. This decision has been made because it appears to offer the most extensive functionality for the lowest development cost and fastest deployment time.

When you develop an application by reusing an existing system, you have to accept the design decisions made by the developers of that system. Let us assume that some of these design decisions are:

1. System users are authenticated using a login name/password combination. No other authentication method is supported.
2. The system architecture is client–server, with clients accessing data through a standard web browser on a client computer.
3. Information is presented to users as an editable web form. They can change information in place and upload the revised information to the server.

For a generic system, these design decisions are perfectly acceptable, but design risk assessment shows that they have associated vulnerabilities. Examples of these possible vulnerabilities are shown in Figure 13.12.

Once vulnerabilities have been identified, you then have to decide what steps you can take to reduce the associated risks. This will often involve making decisions

about additional system security requirements or the operational process of using the system. Examples of these requirements might be:

1. A password checker program shall be made available and shall be run daily to check all user passwords. User passwords that appear in the system dictionary shall be identified, and users with weak passwords shall be reported to system administrators.
2. Access to the system shall only be allowed to client computers that have been approved and registered with the system administrators.
3. Only one approved web browser shall be installed on client computers.

As an off-the-shelf system is used, it isn't possible to include a password checker in the application system itself, so a separate system must be used. Password checkers analyze the strength of user passwords when they are set up and notify users if they have chosen weak passwords. Therefore, vulnerable passwords can be identified reasonably quickly after they have been set up, and action can then be taken to ensure that users change their password.

The second and third requirements mean that all users will always access the system through the same browser. You can decide what is the most secure browser when the system is deployed and install that on all client computers. Security updates are simplified because there is no need to update different browsers when security vulnerabilities are discovered and fixed.

The process model shown in Figure 13.10 assumes a design process where the design is developed to a fairly detailed level before implementation begins. This is not the case for agile processes where the design and the implementation are developed together, with the code refactored as the design is developed. Frequent delivery of system increments does not allow time for a detailed risk assessment, even if information on assets and technology choices is available.

The issues surrounding security and agile development have been widely discussed (Lane 2010; Schoenfield 2013). So far, the issue has not really been resolved—some people think that a fundamental conflict exists between security and agile development, and others believe that this conflict can be resolved using security-focused stories (Safecode 2012). This remains an outstanding problem for developers of agile methods. Meanwhile, many security-conscious companies refuse to use agile methods because they conflict with their security and risk analysis policies.

13.4.2 Architectural design

Software architecture design decisions can have profound effects on the emergent properties of a software system. If an inappropriate architecture is used, it may be very difficult to maintain the confidentiality and integrity of information in the system or to guarantee a required level of system availability.

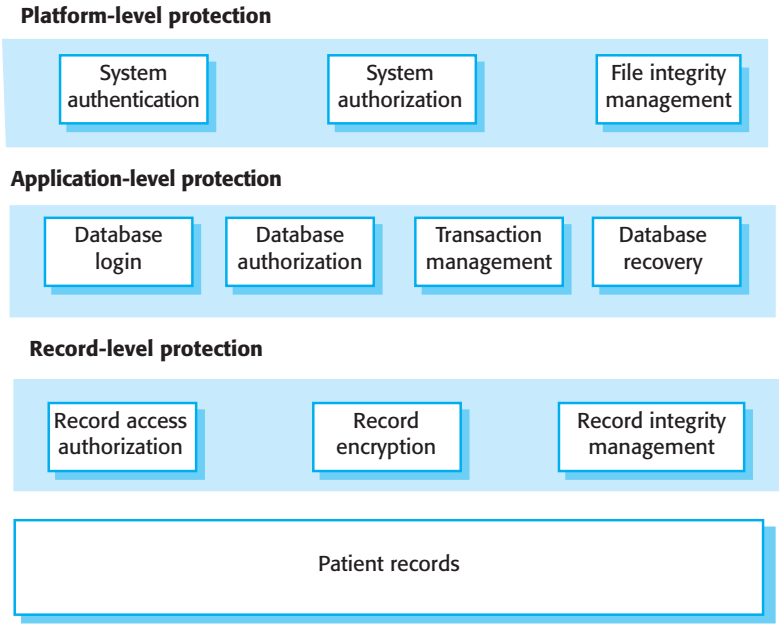


Figure 13.13 A layered protection architecture

In designing a system architecture that maintains security, you need to consider two fundamental issues:

1. *Protection*—how should the system be organized so that critical assets can be protected against external attack?
2. *Distribution*—how should system assets be distributed so that the consequences of a successful attack are minimized?

These issues are potentially conflicting. If you put all your assets in one place, then you can build layers of protection around them. As you only have to build a single protection system, you may be able to afford a strong system with several protection layers. However, if that protection fails, then all your assets are compromised. Adding several layers of protection also affects the usability of a system, so it may mean that it is more difficult to meet system usability and performance requirements.

On the other hand, if you distribute assets, they are more expensive to protect because protection systems have to be implemented for each distributed asset. Typically, then, you cannot afford to implement as many protection layers. The chances are greater that the protection will be breached. However, if this happens, you don't suffer a total loss. It may be possible to duplicate and distribute information assets so that if one copy is corrupted or inaccessible, then the other copy can be used. However, if the information is confidential, keeping additional copies increases the risk that an intruder will gain access to this information.

For the Mentcare system, a client–server architecture with a shared central database is used. To provide protection, the system has a layered architecture with the

critical protected assets at the lowest level in the system. Figure 13.13 illustrates this multilevel system architecture in which the critical assets to be protected are the records of individual patients.

To access and modify patient records, an attacker has to penetrate three system layers:

1. *Platform-level protection.* The top level controls access to the platform on which the patient record system runs. This usually involves a user signing-on to a particular computer. The platform will also normally include support for maintaining the integrity of files on the system, backups, and so on.
2. *Application-level protection.* The next protection level is built into the application itself. It involves a user accessing the application, being authenticated, and getting authorization to take actions such as viewing or modifying data. Application-specific integrity management support may be available.
3. *Record-level protection.* This level is invoked when access to specific records is required, and involves checking that a user is authorized to carry out the requested operations on that record. Protection at this level might also involve encryption to ensure that records cannot be browsed using a file browser. Integrity checking using, for example, cryptographic checksums can detect changes that have been made outside the normal record update mechanisms.

The number of protection layers that you need in any particular application depends on the criticality of the data. Not all applications need protection at the record level, and, therefore, coarser-grain access control is more commonly used. To achieve security, you should not allow the same user credentials to be used at each level. Ideally, if you have a password-based system, then the application password should be different from both the system password and the record-level password. However, multiple passwords are difficult for users to remember, and they find repeated requests to authenticate themselves irritating. Therefore, you often have to compromise on security in favor of system usability.

If protection of data is a critical requirement, then a centralized client–server architecture is usually the most effective security architecture. The server is responsible for protecting sensitive data. However, if the protection is compromised, then the losses associated with an attack are high, as all data may be lost or damaged. Recovery costs may also be high (e.g., all user credentials may have to be reissued). Centralized systems are also more vulnerable to denial-of-service attacks, which overload the server and make it impossible for anyone to access the system database.

If the consequences of a server breach are high, you may decide to use an alternative distributed architecture for the application. In this situation, the system's assets are distributed across a number of different platforms, with separate protection mechanisms used for each of these platforms. An attack on one node might mean that some assets are unavailable, but it would still be possible to provide some system services. Data can be replicated across the nodes in the system so that recovery from attacks is simplified.

Figure 13.14 illustrates the architecture of a banking system for trading in stocks and funds on the New York, London, Frankfurt, and Hong Kong markets. The system

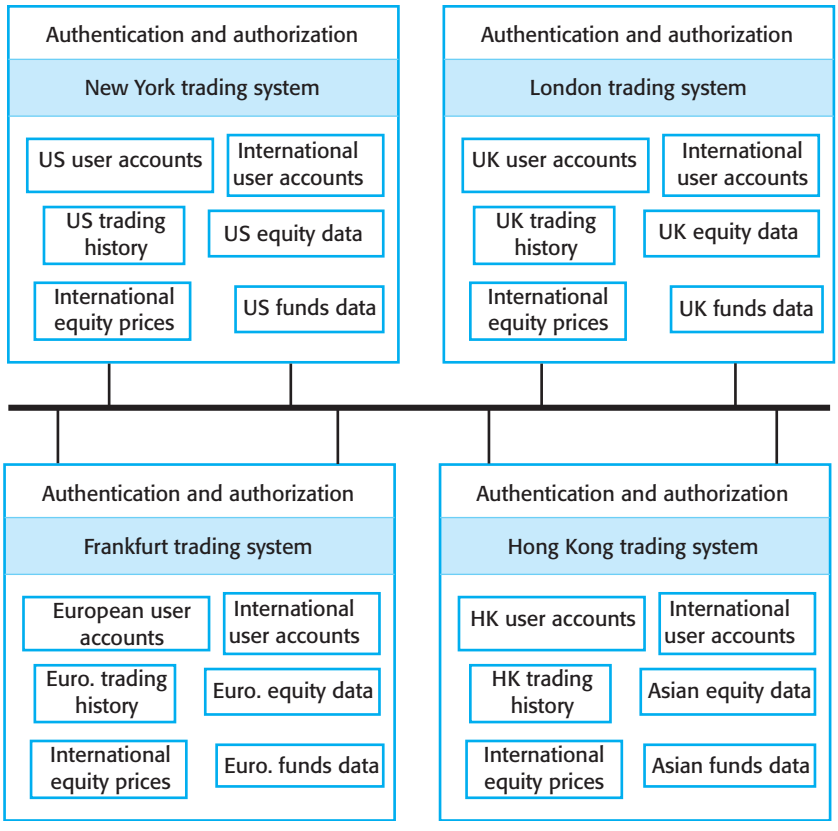


Figure 13.14 Distributed assets in an equity trading system

is distributed so that data about each market is maintained separately. Assets required to support the critical activity of equity trading (user accounts and prices) are replicated and available on all nodes. If a node of the system is attacked and becomes unavailable, the critical activity of equity trading can be transferred to another country and so can still be available to users.

I have already discussed the problem of finding a balance between security and system performance. A problem of secure system design is that in many cases, the architectural style that is best for the security requirements may not be the best one for meeting the performance requirements. For example, say an application has an absolute requirement to maintain the confidentiality of a large database and another requirement for very fast access to that data. A high-level of protection suggests that layers of protection are required, which means that there must be communications between the system layers. This has an inevitable performance overhead and so will slow down access to the data.

If an alternative architecture is used, then implementing protection and guaranteeing confidentiality may be more difficult and expensive. In such a situation, you have to discuss the inherent conflicts with the customer who is paying for the system and agree on how these conflicts are to be resolved.

13.4.3 Design guidelines

There are no easy ways to ensure system security. Different types of systems require different technical measures to achieve a level of security that is acceptable to the system owner. The attitudes and requirements of different groups of users profoundly affect what is and is not acceptable. For example, in a bank, users are likely to accept a higher level of security, and hence more intrusive security procedures than, say, in a university.

However, some general guidelines have wide applicability when designing system security solutions. These guidelines encapsulate good design practice for secure systems engineering. General design guidelines for security, such as those discussed, below, have two principal uses:

1. They help raise awareness of security issues in a software engineering team. Software engineers often focus on the short-term goal of getting the software working and delivered to customers. It is easy for them to overlook security issues. Knowledge of these guidelines can mean that security issues are considered when software design decisions are made.
2. They can be used as a review checklist that can be used in the system validation process. From the high-level guidelines discussed here, more specific questions can be derived that explore how security has been engineered into a system.

Security guidelines are sometimes very general principles such as "Secure the weakest link in a system," "Keep it simple," and "Avoid security through obscurity." I think these general guidelines are too vague to be of real use in the design process. Consequently, I have focused here on more specific design guidelines. The 10 design guidelines, summarized in Figure 13.15, have been taken from different sources (Schneier 2000; Viega and McGraw 2001; Wheeler 2004).

Guideline 1: Base security decisions on an explicit security policy

An organizational security policy is a high-level statement that sets out fundamental security conditions for an organization. It defines the "what" of security rather than the "how." so the policy should not define the mechanisms to be used to provide and enforce security. In principle, all aspects of the security policy should be reflected in the system requirements. In practice, especially if agile development is used, this is unlikely to happen.

Designers should use the security policy as a framework for making and evaluating design decisions. For example, say you are designing an access control system for the Mentcare system. The hospital security policy may state that only accredited clinical staff may modify electronic patient records. This leads to requirements to check the accreditation of anyone attempting to modify the system and to reject modifications from unaccredited people.

The problem that you may face is that many organizations do not have an explicit systems security policy. Over time, changes may have been made to systems in response to identified problems, but with no overarching policy document to guide the evolution of a system. In such situations, you need to work out and document the policy from examples and confirm it with managers in the company.

Design guidelines for security	
1	Base security decisions on an explicit security policy
2	Use defense in depth
3	Fail securely
4	Balance security and usability
5	Log user actions
6	Use redundancy and diversity to reduce risk
7	Specify the format of system inputs
8	Compartmentalize your assets
9	Design for deployment
10	Design for recovery

Figure 13.15 Design guidelines for secure systems engineering

Guideline 2: Use defense in depth

In any critical system, it is good design practice to try to avoid a single point of failure. That is, a single failure in part of the system should not result in an overall systems failure. In security terms, this means that you should not rely on a single mechanism to ensure security; rather, you should employ several different techniques. This concept is sometimes called "defense in depth."

An example of defense in depth is multifactor authentication. For example, if you use a password to authenticate users to a system, you may also include a challenge/response authentication mechanism where users have to pre-register questions and answers with the system. After they have input their login credentials, they must then answer questions correctly before being allowed access.

Guideline 3: Fail securely

System failures are inevitable in all systems, and, in the same way that safety-critical systems should always fail-safe; security-critical systems should always "fail-secure." When the system fails, you should not use fallback procedures that are less secure than the system itself. Nor should system failure mean that an attacker can access data that would not normally be allowed.

For example, in the Mentcare system, I suggested a requirement that patient data should be downloaded to a system client at the beginning of a clinic session. This speeds up access and means that access is possible if the server is unavailable. Normally, the server deletes this data at the end of the clinic session. However, if the server has failed, then it is possible that the information on the client will be maintained. A fail-secure approach in those circumstances is to encrypt all patient data stored on the client. This means that an unauthorized user cannot read the data.

Guideline 4: Balance security and usability

The demands of security and usability are often contradictory. To make a system secure, you have to introduce checks that users are authorized to use the system and

that they are acting in accordance with security policies. All of these inevitably make demands on users—they may have to remember login names and passwords, only use the system from certain computers, and so on. These mean that it takes users more time to get started with the system and use it effectively. As you add security features to a system, it usually becomes more difficult to use. I recommend Cranor and Garfinkel's book (Cranor and Garfinkel 2005), which discusses a wide range of issues in the general area of security and usability.

There comes a point when it is counterproductive to keep adding on new security features at the expense of usability. For example, if you require users to input multiple passwords or to change their passwords to impossible to remember character strings at frequent intervals, they will simply write down these passwords. An attacker (especially an insider) may then be able to find the passwords that have been written down and gain access to the system.

Guideline 5: Log user actions

If it is practically possible to do so, you should always maintain a log of user actions. This log should, at least, record who did what, the assets used and the time and date of the action. If you maintain this as a list of executable commands, you can replay the log to recover from failures. You also need tools that allow you to analyze the log and detect potentially anomalous actions. These tools can scan the log and find anomalous actions, and thus help detect attacks and trace how the attacker gained access to the system.

Apart from helping recover from failure, a log of user actions is useful because it acts as a deterrent to insider attacks. If people know that their actions are being logged, then they are less likely to do unauthorized things. This is most effective for casual attacks, such as a nurse looking up patient records of neighbors, or for detecting attacks where legitimate user credentials have been stolen through social engineering. Of course, this approach is not foolproof, as technically skilled insiders may also be able to access and change the log.

Guideline 6: Use redundancy and diversity to reduce risk

Redundancy means that you maintain more than one version of software or data in a system. Diversity, when applied to software, means that the different versions should not rely on the same platform or be implemented using the same technologies. Therefore, platform or technology vulnerabilities will not affect all versions and so will lead to a common failure.

I have already discussed examples of redundancy—maintaining patient information on both the server and the client, first in the Mentcare system and then in the distributed equity trading system shown in Figure 13.14. In the patient records system, you could use diverse operating systems on the client and the server (e.g., Linux on the server, Windows on the client). This ensures that an attack based on an operating system vulnerability will not affect both the server and the client. Of course, running multiple operating systems leads to higher systems management costs. You have to trade off security benefits against this increased cost.

Guideline 7: Specify the format of system inputs

A common attack on a system involves providing the system with unexpected inputs that cause it to behave in an unanticipated way. These inputs may simply cause a system crash, resulting in a loss of service, or the inputs could be made up of malicious code that is executed by the system. Buffer overflow vulnerabilities, first demonstrated in the Internet worm (Spafford 1989) and commonly used by attackers, may be triggered using long input strings. So-called SQL poisoning, where a malicious user inputs an SQL fragment that is interpreted by a server, is another fairly common attack.

You can avoid many of these problems if you specify the format and structure of the system inputs that are expected. This specification should be based on your knowledge of the expected system inputs. For example, if a surname is to be input, you might specify that all characters must be alphabetic with no numbers or punctuation (apart from a hyphen) allowed. You might also limit the length of the name. For example, no one has a family name with more than 40 characters, and no addresses are more than 100 characters long. If a numeric value is expected, no alphabetic characters should be allowed. This information is then used in input checks when the system is implemented.

Guideline 8: Compartmentalize your assets

Compartmentalizing means that you should not provide users with access to all information in a system. Based on a general "need to know" security principle, you should organize the information in a system into compartments. Users should only have access to the information that they need for their work, rather than to all of the information in a system. This means that the effects of an attack that compromises an individual user account may be contained. Some information may be lost or damaged, but it is unlikely that all of the information in the system will be affected.

For example, the Mentcare system could be designed so that clinic staff will normally only have access to the records of patients who have an appointment at their clinic. They should not normally have access to all patient records in the system. Not only does this limit the potential loss from insider attacks, but it also means that if an intruder steals their credentials, then they cannot damage all patient records.

Having said this, you also may have to have mechanisms in the system to grant unexpected access—say to a patient who is seriously ill and requires urgent treatment without an appointment. In those circumstances, you might use some alternative secure mechanism to override the compartmentalization in the system. In such situations, where security is relaxed to maintain system availability, it is essential that you use a logging mechanism to record system usage. You can then check the logs to trace any unauthorized use.

Guideline 9: Design for deployment

Many security problems arise because the system is not configured correctly when it is deployed in its operational environment. Deployment means installing the software

on the computers where it will execute and setting software parameters to reflect the execution environment and the preferences of the system user. Mistakes such as forgetting to turn off debugging facilities or forgetting to change the default administration password can introduce vulnerabilities into a system.

Good management practice can avoid many security problems that arise from configuration and deployment mistakes. However, software designers have the responsibility to "design for deployment." You should always provide support for deployment that reduces the chances of users and system administrators making mistakes when configuring the software.

I recommend four ways to incorporate deployment support in a system:

1. *Include support for viewing and analyzing configurations* You should always include facilities in a system that allow administrators or permitted users to examine the current configuration of the system.
2. *Minimize default privileges* You should design software so that the default configuration of a system provides minimum essential privileges.
3. *Localize configuration settings* When designing system configuration support, you should ensure that everything in a configuration that affects the same part of a system is set up in the same place.
4. *Provide easy ways to fix security vulnerabilities* You should include straightforward mechanisms for updating the system to repair security vulnerabilities that have been discovered.

Deployment issues are less of a problem than they used to be as more and more software does not require client installation. Rather, the software runs as a service and is accessed through a web browser. However, server software is still vulnerable to deployment errors and omissions, and some types of system require dedicated software running on the user's computer.

Guideline 10: Design for recovery

Irrespective of how much effort you put into maintaining systems security, you should always design your system with the assumption that a security failure could occur. Therefore, you should think about how to recover from possible failures and restore the system to a secure operational state. For example, you may include a backup authentication system in case your password authentication is compromised.

For example, say an unauthorized person from outside the clinic gains access to the Mentcare system and you don't know how that person obtained a valid login/password combination. You need to re-initialize the authentication system and not just change the credentials used by the intruder. This is essential because the intruder may also have gained access to other user passwords. You need, therefore, to ensure that all authorized users change their passwords. You also must ensure that the unauthorized person does not have access to the password-changing mechanism.

You therefore have to design your system to deny access to everyone until they have changed their password and to email all users asking them to make the change. You need an alternative mechanism to authenticate real users for password change, assuming that their chosen passwords may not be secure. One way of doing this is to use a challenge/response mechanism, where users have to answer questions for which they have pre-registered answers. This is only invoked when passwords are changed, allowing for recovery from the attack with relatively little user disruption.

Designing for recoverability is an essential element of building resilience into systems. I cover this topic in more detail in Chapter 14.

13.4.4 Secure systems programming

Secure system design means designing security into an application system. However, as well as focusing on security at the design level, it is also important to consider security when programming a software system. Many successful attacks on software rely on program vulnerabilities that were introduced when the program was developed.

The first widely known attack on Internet-based systems happened in 1988 when a worm was introduced into Unix systems across the network (Spafford 1989). This took advantage of a well-known programming vulnerability. If systems are programmed in C, there is no automatic array bound checking. An attacker can include a long string with program commands as an input, and this overwrites the program stack and can cause control to be transferred to malicious code. This vulnerability has been exploited in many other systems programmed in C or C++ since then.

This example illustrates two important aspects of secure systems programming:

1. Vulnerabilities are often language-specific. Array bound checking is automatic in languages such as Java, so this is not a vulnerability that can be exploited in Java programs. However, millions of programs are written in C and C++ as these allow for the development of more efficient software. Thus. simply avoiding the use of these languages is not a realistic option.
2. Security vulnerabilities are closely related to program reliability. The above example caused the program concerned to crash, so actions taken to improve program reliability can also improve system security.

In Chapter 11, I introduced programming guidelines for dependable system programming. These are shown in Figure 13.16. These guidelines also help improve the security of a program as attackers focus on program vulnerabilities to gain access to a system. For example, an SQL poisoning attack is based on the attacker filling in a form with SQL commands rather than the text expected by the system. These can corrupt the database or release confidential information. You can completely avoid this problem if you implement input checks (Guideline 2) based on the expected format and structure of the inputs.

Dependable programming guidelines

1. Limit the visibility of information in a program.
2. Check all inputs for validity.
3. Provide a handler for all exceptions.
4. Minimize the use of error-prone constructs.
5. Provide restart capabilities.
6. Check array bounds.
7. Include timeouts when calling external components.
8. Name all constants that represent real-world values.

Figure 13.16 Dependable programming guidelines

13.5 Security testing and assurance

The assessment of system security is increasingly important so that we can be confident that the systems we use are secure. The verification and validation processes for web-based systems should therefore focus on security assessment, where the ability of the system to resist different types of attack is tested. However, as Anderson explains (Anderson 2008), this type of security assessment is very difficult to carry out. Consequently, systems are often deployed with security loopholes. Attackers use these vulnerabilities to gain access to the system or to cause damage to the system or its data.

Fundamentally, security testing is difficult for two reasons:

1. Security requirements, like some safety requirements, are "shall not" requirements. That is, they specify what should not happen rather than system functionality or required behavior. It is not usually possible to define this unwanted behavior as simple constraints to be checked by the system.

   If resources are available, you can demonstrate, in principle at least, that a system meets its functional requirements. However, it is impossible to prove that a system does not do something. Irrespective of the amount of testing, security vulnerabilities may remain in a system after it has been deployed.

   You may, of course, generate functional requirements that are designed to guard the system against some known types of attack. However, you cannot derive requirements for unknown or unanticipated types of attack. Even in systems that have been in use for many years, an ingenious attacker can discover a new attack and can penetrate what was thought to be a secure system.

2. The people attacking a system are intelligent and are actively looking for vulnerabilities that they can exploit. They are willing to experiment with the system and to try things that are far outside normal activity and system use. For example, in a surname field they may enter 1000 characters with a mixture of letters, punctuation, and numbers simply to see how the system responds.

   Once they find a vulnerability, they publicize it and so increase the number of possible attackers. Internet forums have been set up to exchange information about system vulnerabilities. There is also a thriving market in malware where

Security checklist
1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that passwords are "strong"? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.
5. Are inputs from the system's environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Figure 13.17 Examples of entries in a security checklist

attackers can get access to kits that help them easily develop malware such as worms and keystroke loggers.

Attackers may try to discover the assumptions made by system developers and then challenge these assumptions to see what happens. They are in a position to use and explore a system over a period of time and analyze it using software tools to discover vulnerabilities that they may be able to exploit. They may, in fact, have more time to spend on looking for vulnerabilities than system test engineers, as testers must also focus on testing the system.

You may use a combination of testing, tool-based analysis, and formal verification to check and analyze the security of an application system:

1. *Experience-based testing* In this case, the system is analyzed against types of attack that are known to the validation team. This may involve developing test cases or examining the source code of a system. For example, to check that the system is not susceptible to the well-known SQL poisoning attack, you might test the system using inputs that include SQL commands. To check that buffer overflow errors will not occur, you can examine all input buffers to see if the program is checking that assignments to buffer elements are within bounds.

   Checklists of known security problems may be created to assist with the process. Figure 13.17 gives some examples of questions that might be used to drive experience-based testing. Checks on whether design and programming guidelines for security have been followed may also be included in a security problem checklist.

2. *Penetration testing* This is a form of experience-based testing where it is possible to draw on experience from outside the development team to test an application system. The penetration testing teams are given the objective of breaching the system security. They simulate attacks on the system and use their ingenuity to discover new ways to compromise the system security. Penetration testing team

members should have previous experience with security testing and finding security weaknesses in systems.

3. *Tool-based analysis* In this approach, security tools such as password checkers are used to analyze the system. Password checkers detect insecure passwords such as common names or strings of consecutive letters. This approach is really an extension of experience-based validation, where experience of security flaws is embodied in the tools used. Static analysis is, of course, another type of tool-based analysis, which has become increasingly used.

   Tool-based static analysis (Chapter 12) is a particularly useful approach to security checking. A static analysis of a program can quickly guide the testing team to areas of a program that may include errors and vulnerabilities. Anomalies revealed in the static analysis can be directly fixed or can help identify tests that need to be done to reveal whether or not these anomalies actually represent a risk to the system. Microsoft uses static analysis routinely to check its software for possible security vulnerabilities (Jenney 2013). Hewlett-Packard offers a tool called Fortify (Hewlett-Packard 2012) specifically designed for checking Java programs for security vulnerabilities.

4. *Formal verification* I have discussed the use of formal program verification in Chapters 10 and 12. Essentially, this involves making formal, mathematical arguments that demonstrate that a program conforms to its specification. Hall and Chapman (Hall and Chapman 2002) demonstrated the feasibility of proving that a system met its formal security requirements more than 10 years ago, and there have been a number of other experiments since then. However, as in other areas, formal verification for security is not widely used. It requires specialist expertise and is unlikely to be as cost-effective as static analysis.

Security testing takes a long time, and, usually, the time available to the testing team is limited. This means that you should adopt a risk-based approach to security testing and focus on what you think are the most significant risks faced by the system. If you have an analysis of the security risks to the system, these can be used to drive the testing process. As well as testing the system against the security requirements derived from these risks, the test team should also try to break the system by adopting alternative approaches that threaten the system assets.

KEY POINTS

- Security engineering focuses on how to develop and maintain software systems that can resist malicious attacks intended to damage a computer-based system or its data.
- Security threats can be threats to the confidentiality, integrity, or availability of a system or its data.

- Security risk management involves assessing the losses that might ensue from attacks on a system, and deriving security requirements that are aimed at eliminating or reducing these losses.
- To specify security requirements, you should identify the assets that are to be protected and define how security techniques and technology should be used to protect these assets.
- Key issues when designing a secure systems architecture include organizing the system structure to protect key assets and distributing the system assets to minimize the losses from a successful attack.
- Security design guidelines sensitize system designers to security issues that they may not have considered. They provide a basis for creating security review checklists.
- Security validation is difficult because security requirements state what should not happen in a system, rather than what should. Furthermore, system attackers are intelligent and may have more time to probe for weaknesses than is available for security testing.

FURTHER READING

Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd ed. This is a thorough and comprehensive discussion of the problems of building secure systems. The focus is on systems rather than software engineering, with extensive coverage of hardware and networking, with excellent examples drawn from real system failures. (R. Anderson, John Wiley & Sons, 2008) http://www.cl.cam.ac.uk/~rja14/book.html

24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. I think this is one of the best practical books on secure systems programming. The authors discuss the most common programming vulnerabilities and describe how they can be avoided in practice. (M. Howard, D. LeBlanc, and J. Viega, McGraw-Hill, 2009).

Computer Security: Principles and Practice. This is a good general text on computer security issues. It covers security technology, trusted systems, security management, and cryptography. (W. Stallings and L. Brown, Addison-Wesley, 2012).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/security-and-resilience/

EXERCISES

13.1. Describe the security dimensions and security levels that have to be considered in secure systems engineering.

13.2. For the Mentcare system, suggest an example of an asset, an exposure, a vulnerability, an attack, a threat, and a control, in addition to those discussed in this chapter.

13.3. Explain why security is considered a more challenging problem than safety in a system.

13.4. Extend the table in Figure 13.7 to identify two further threats to the Mentcare system, along with associated controls. Use these as a basis for generating software security requirements that implement the proposed controls.

13.5. Explain, using an analogy drawn from a non-software engineering context, why a layered approach to asset protection should be used.

13.6. Explain why it is important to log user actions in the development of secure systems.

13.7. For the equity trading system discussed in Section 13.4.2, whose architecture is shown in Figure 13.14, suggest two further plausible attacks on the system and propose possible strategies that could counter these attacks.

13.8. Explain why it is important when writing secure systems to validate all user inputs to check that these have the expected format.

13.9. Suggest how you would go about validating a password protection system for an application that you have developed. Explain the function of any tools that you think may be useful.

13.10. The Mentcare system has to be secure against attacks that might reveal confidential patient information. Suggest three possible attacks against this system that might occur. Using this information, extend the checklist in Figure 13.17 to guide testers of the Mentcare system.

REFERENCES

Anderson, R. 2008. *Security Engineering, 2nd ed.* Chichester, UK: John Wiley & Sons.

Cranor, L. and S. Garfinkel. 2005. *Designing Secure Systems That People Can Use.* Sebastopol, CA: O'Reilly Media Inc.

Firesmith, D. G. 2003. "Engineering Security Requirements." *Journal of Object Technology* 2 (1): 53–68. http://www.jot.fm/issues/issue_2003_01/column6

Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18–25. doi:10.1109/52.976937.

Hewlett-Packard. 2012. "Securing Your Enterprise Software: Hp Fortify Code Analyzer." http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-2455ENW&cc=us&lc=en

Jenney, P. 2013. "Static Analysis Strategies: Success with Code Scanning." http://msdn.microsoft.com/en-us/security/gg615593.aspx

Lane, A. 2010. "Agile Development and Security." https://securosis.com/blog/agile-development-and-security

Pfleeger, C. P., and S. L. Pfleeger. 2007. *Security in Computing, 4th ed.* Boston: Addison-Wesley.

Safecode. 2012. "Practical Security Stories and Security Tasks for Agile Development Environments." http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf

Schneier, B. 1999. "Attack Trees." *Dr Dobbs Journal* 24 (12): 1–9. https://www.schneier.com/paper-attacktrees-ddj-ft.html

———. 2000. *Secrets and Lies: Digital Security in a Networked World.* New York: John Wiley & Sons.

Schoenfield, B. 2013. "Agile and Security: Enemies for Life?" http://brookschoenfield.com/?p=151

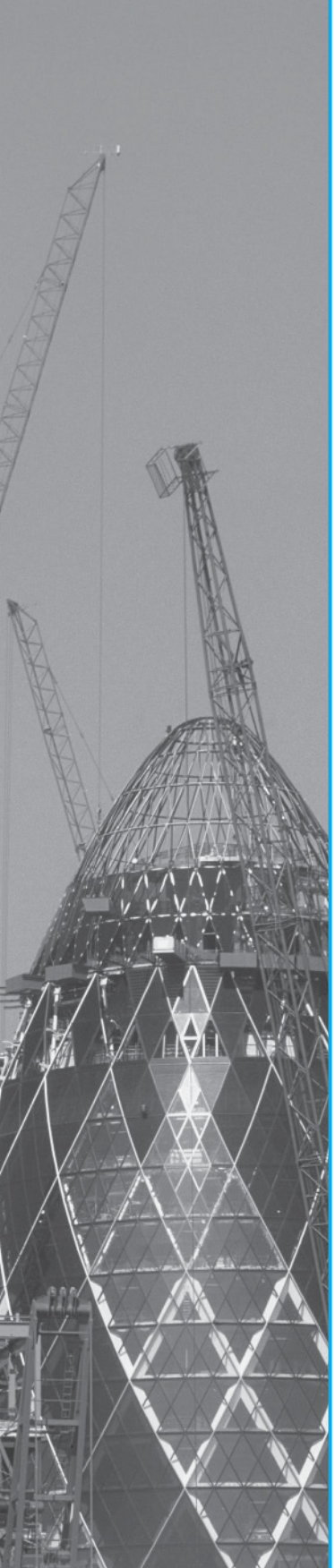
Sindre, G., and A. L. Opdahl. 2005. "Eliciting Security Requirements through Misuse Cases." *Requirements Engineering* 10 (1): 34–44. doi:10.1007/s00766-004-0194-4.

Spafford, E. 1989. "The Internet Worm: Crisis and Aftermath." *Comm ACM* 32 (6): 678–687. doi:10.1145/63526.63527.

Stallings, W., and L. Brown. 2012. *Computer Security: Principles, d Practice.* (*2nd ed.*) Boston: Addison-Wesley.

Viega, J., and G. McGraw. 2001. *Building Secure Software.* Boston: Addison-Wesley.

Wheeler, D. A. 2004. *Secure Programming for Linux and Unix.* Self-published. http://www.dwheeler.com/secure-programs/



14
Resilience engineering

Objectives

The objective of this chapter is to introduce the idea of resilience engineering where systems are designed to withstand adverse external events such as operator errors and cyberattacks. When you have read this chapter, you will:

- understand the differences between resilience, reliability, and security and why resilience is important for networked systems;
- be aware of the fundamental issues in building resilient systems, namely, recognition of problems, resistance to failures and attacks, recovery of critical services, and system reinstatement;
- understand why resilience is a sociotechnical rather than a technical issue and the role of system operators and managers in providing resilience;
- have been introduced to a system design method that supports resilience.

Contents

14.1 Cybersecurity
14.2 Sociotechnical resilience
14.3 Resilient systems design

In April 1970, the Apollo 13 manned mission to the moon suffered a catastrophic failure. An oxygen tank exploded in space, resulting in a serious loss of atmospheric oxygen and oxygen for the fuel cells that powered the spacecraft. The situation was life threatening, with no possibility of rescue. There were no contingency plans for this situation. However, by using equipment in unintended ways and by adapting standard procedures, the combined efforts of the spacecraft crew and ground staff worked around the problems. The spacecraft was brought back to earth safely, and all the crew survived. The overall system (people, equipment, and processes) was *resilient*. It adapted to cope with and recover from the failure.

I introduced the idea of resilience in Chapter 10, as one of the fundamental attributes of system dependability. I defined resilience in Chapter 10 as:

> *The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.*

This is not a "standard" definition of resilience—different authors such as Laprie (Laprie 2008) and Hollnagel (Hollnagel 2006) propose general definitions based on the ability of a system to withstand change. That is, a resilient system is one that can operate successfully when some of the fundamental assumptions made by the system designers no longer hold.

For example, an initial design assumption may be that users will make mistakes but will not deliberately seek out system vulnerabilities to be exploited. If the system is used in an environment where it may be subject to cyberattacks, this is no longer true. A resilient system can cope with the environmental change and can continue to operate successfully.

While these definitions are more general, my definition of resilience is closer to how the term is now used in practice by governments and industry. It embeds three essential ideas:

1. The idea that some of the services offered by a system are critical services whose failure could have serious human, social, or economic effects.
2. The idea that some events are disruptive and can affect the ability of a system to deliver its critical services.
3. The idea that resilience is a judgment—there are no resilience metrics, and resilience cannot be measured. The resilience of a system can only be assessed by experts, who can examine the system and its operational processes.

Fundamental work on system resilience started in the safety-critical systems community, where the aim was to understand what factors led to accidents being avoided and survived. However, the increasing number of cyberattacks on networked systems has meant that resilience is now often seen as a security issue. It is essential to build systems that can withstand malicious cyberattacks and continue to deliver services to their users.

Obviously, resilience engineering is closely related to reliability and security engineering. The aim of reliability engineering is to ensure that systems do not fail. A system failure is an externally observable event, which is often a consequence of a fault in the system. Therefore, techniques such as fault avoidance and fault tolerance, as discussed in Chapter 11, have been developed to reduce the number of system faults and to trap faults before they lead to system failure.

In spite of our best efforts, faults will always be present in a large, complex system, and they may lead to system failure. Delivery schedules are short, and testing budgets are limited. Development teams are working under pressure, and it is practically impossible to detect all of the faults and security vulnerabilities in a software system. We are building systems that are so complex (see Chapter 19) that we cannot possibly understand all of the interactions between the system components. Some of these interactions may be a trigger for overall system failure.

Resilience engineering does not focus on avoiding failure but rather on accepting the reality that failures will occur. It makes two important assumptions:

1. Resilience engineering assumes that it is impossible to avoid system failures and so is concerned with limiting the costs of these failures and recovering from them.
2. Resilience engineering assumes that good reliability engineering practices have been used to minimize the number of technical faults in a system. It therefore places more emphasis on limiting the number of system failures that arise from external events such as operator errors or cyberattacks.

In practice, technical system failures are often triggered by events that are external to the system. These events may involve operator actions or user errors that are unexpected. Over the last few years, however, as the number of networked systems has increased, these events have often been cyberattacks. In a cyberattack, a malicious person or group tries to damage the system or to steal confidential information. These are now more significant than user or operator errors as a potential source of system failure.

Because of the assumption that failures will inevitably occur, resilience engineering is concerned with both the immediate recovery from failure to maintain critical services and the longer-term reinstatement of all system services. As I discuss in Section 14.3, this means that system designers have to include system features to maintain the state of the system's software and data. In the event of a failure, essential information may then be restored.

Four related resilience activities are involved in the detection of and recovery from system problems:

1. *Recognition* The system or its operators should be able to recognize the symptoms of a problem that may lead to system failure. Ideally, this recognition should be possible before the failure occurs.
2. *Resistance* If the symptoms of a problem or signs of a cyberattack are detected early, then resistance strategies may be invoked that reduce the probability that the system will fail. These resistance strategies may focus on isolating critical parts of the system so that they are unaffected by problems elsewhere. Resistance includes

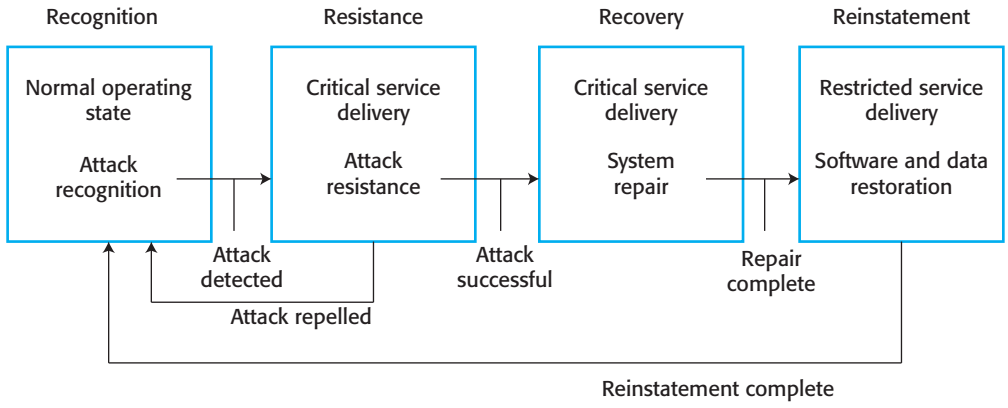


Figure 14.1 Resilience activities

proactive resistance where defenses are included in a system to trap problems and reactive resistance where actions are taken when a problem is discovered.

3. *Recovery* If a failure occurs, the aim of the recovery activity is to ensure that critical system services are restored quickly so that system users are not seriously affected by the failure.

4. *Reinstatement* In this final activity, all of the system services are restored, and normal system operation can continue.

These activities lead to changes to the system state as shown in Figure 14.1, which shows the state changes in the system in the event of a cyberattack. In parallel with normal system operation, the system monitors network traffic for possible cyberattacks. In the event of a cyberattack, the system moves to a resistance state in which normal services may be restricted.

If resistance successfully repels the attack, normal service is resumed. Otherwise, the system moves to a recovery state where only critical services are available. Repairs to the damage caused by the cyberattack are carried out. Finally, when repairs are complete, the system moves to a reinstatement state. In this state, the system's services are incrementally restored. Finally, when all restoration is complete, normal service is resumed.

As the Apollo 13 example illustrates, resilience cannot be "programmed in" to a system. It is impossible to anticipate everything that might go wrong and every context where problems might arise. The key to resilience, therefore, is flexibility and adaptability. As I discuss in Section 14.2, it should be possible for system operators and managers to take actions to protect and repair the system, even if these actions are abnormal or are normally disallowed.

Increasing the resilience of a system of course has significant costs. Software may have to be purchased or modified, and additional investments made in hardware or cloud services to provide backup systems that can be used in the event of a system failure. The benefits from these costs are impossible to calculate because the losses from a failure or attack can only be calculated after the event.

Companies may therefore be reluctant to invest in resilience if they have never suffered a serious attack or associated loss. However, the increasing number of

high-profile cyberattacks that have damaged business and government systems have increased awareness of the need for resilience. It is clear that losses can be very significant, and sometimes businesses may not survive a successful cyberattack. Therefore, there is increasing investment in resilience engineering to reduce the business risks associated with system failure.

14.1 Cybersecurity

Maintaining the security of our networked infrastructure and government, business, and personal computer systems is one of the most significant problems facing our society. The ubiquity of the Internet and our dependence on computer systems have created new criminal opportunities for theft and social disruption. It is very difficult to measure the losses due to cybercrime. However, in 2013, it was estimated that losses to the global economy due to cybercrime were between $100 billion and $500 billion (InfoSecurity 2013).

As I suggested in Chapter 13, cybersecurity is a broader issue than system security engineering. Software security engineering is a primarily technical activity that focuses on techniques and technologies to ensure that application systems are secure. Cybersecurity is a sociotechnical concern. It covers all aspects of ensuring the protection of citizens, businesses, and critical infrastructures from threats that arise from their use of computers and the Internet. While technical issues are important, technology on its own cannot guarantee security. Factors that contribute to cybersecurity failures include:

- organizational ignorance of the seriousness of the problem,
- poor design and lax application of security procedures,
- human carelessness, and
- inappropriate trade-offs between usability and security.

Cybersecurity is concerned with all of an organization's IT assets from networks through to application systems. The vast majority of these assets are externally procured, and companies do not understand their detailed operation. Systems such as web browsers are large and complex programs, and inevitably they contain bugs that can be a source of vulnerability. The different systems in an organization are related to each other in many different ways. They may be stored on the same disk, share data, rely on common operating systems components, and so on. The organizational "system of systems" is incredibly complex. It is impossible to ensure that it is free of security vulnerabilities.

Consequently, you should generally assume that your systems are vulnerable to cyberattack and that, at some stage, a cyberattack is likely to occur. A successful cyberattack can have very serious financial consequences for businesses, so it is essential that attacks are contained and losses minimized. Effective resilience engineering at the organizational and systems levels can repel attacks and bring systems back into operation quickly and so limit the losses incurred.

In Chapter 13, where I discussed security engineering, I introduced concepts that are fundamental to resilience planning. Some of these concepts are:

1. *Assets,* which are systems and data that have to be protected. Some assets are more valuable than others and so require a higher level of protection.
2. *Threats,* which are circumstances that can cause harm by damaging or stealing organizational IT infrastructure or system assets.
3. *Attacks,* which are manifestations of a threat where an attacker aims to damage or steal IT assets, such as websites or personal data.

Three types of threats have to be considered in resilience planning:

1. *Threats to the confidentiality of assets* In this case, data is not damaged, but it is made available to people who should not have access to it. An example of a threat to confidentiality is when a credit card database held by a company is stolen, with the potential for illegal use of card information.
2. *Threats to the integrity of assets* These are threats where systems or data are damaged in some way by a cyberattack. This may involve introducing a virus or a worm into software or corrupting organizational databases.
3. *Threats to the availability of assets* These are threats that aim to deny use of assets by authorized users. The best-known example is a denial-of-service attack that aims to take down a website and so make it unavailable for external use.

These are not independent threat classes. An attacker may compromise the integrity of a user's system by introducing malware, such as a botnet component. This may then be invoked remotely as part of a distributed denial-of-service attack on another system. Other types of malware may be used to capture personal details and so allow confidential assets to be accessed.

To counter these threats, organizations should put controls in place that make it difficult for attackers to access or damage assets. It is also important to raise awareness of cybersecurity issues so that people know why these controls are important and so are less likely to reveal information to an attacker.

Examples of controls that may be used are:

1. *Authentication,* where users of a system have to show that they are authorized to access the system. The familiar login/password approach to authentication is a universally used but rather weak control.
2. *Encryption,* where data is algorithmically scrambled so that an unauthorized reader cannot access the information. Many companies now require that laptop disks are encrypted. If the computer is lost or stolen, this reduces the likelihood that the confidentiality of the information will be breached.
3. *Firewalls,* where incoming network packets are examined, then accepted or rejected according to a set of organizational rules. Firewalls can be used to

ensure that only traffic from trusted sources is allowed to pass from the external Internet into the local organizational network.

A set of controls in an organization provides a layered protection system. An attacker has to get through all of the protection layers for the attack to succeed. However, there is a trade-off between protection and efficiency. As the number of layers of protection increases, the system slows down. The protection systems consume an increasing amount of memory and processor resources, leaving less available to do useful work. The more security, the more inconvenient it is for users and the more likely that they will adopt insecure practices to increase system usability.

As with other aspects of system dependability, the fundamental means of protecting against cyberattacks depends on redundancy and diversity. Recall that redundancy means having spare capacity and duplicated resources in a system. Diversity means that different types of equipment, software, and procedures are used so that common failures are less likely to occur across a number of systems. Examples of where redundancy and diversity are valuable for cyber-resilience are:

1. For each system, copies of data and software should be maintained on separate computer systems. Shared disks should be avoided if possible. This supports recovery after a successful cyberattack (recovery and reinstatement).
2. Multi-stage diverse authentication can protect against password attacks. As well as login/password authentication, additional authentication steps may be involved that require users to provide some personal information or a code generated by their mobile device (resistance).
3. Critical servers may be overprovisioned; that is, they may be more powerful than is required to handle their expected load. The spare capacity means that attacks may be resisted without necessarily degrading the normal response of the server. Furthermore, if other servers are damaged, spare capacity is available to run their software while they are being repaired (resistance and recovery).

Planning for cybersecurity has to be based on assets and controls and the 4 Rs of resilience engineering—recognition, resistance, recovery, and reinstatement. Figure 14.2 shows a planning process that may be followed. The key stages in this process are:

1. *Asset classification* The organization's hardware, software, and human assets are examined and classified depending on how essential they are to normal operations. They may be classed as critical, important, or useful.
2. *Threat identification* For each of the assets (or at least the critical and important assets), you should identify and classify threats to that asset. In some cases, you may try to estimate the probability that a threat will arise, but such estimates are often inaccurate as you don't have enough information about potential attackers.
3. *Threat recognition* For each threat or, sometimes asset/threat pair, you should identify how an attack based on that threat might be recognized. You may

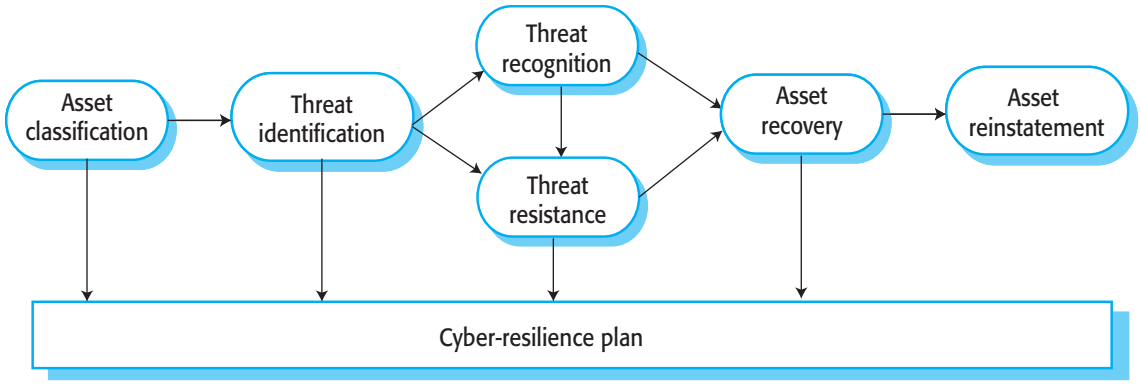


Figure 14.2 Cyber-resilience planning

decide that additional software needs to be bought or written for threat recognition or that regular checking procedures are put in place.

4. *Threat resistance* For each threat or asset/threat pair, you should identify possible resistance strategies. These either may be embedded in the system (technical strategies) or may rely on operational procedures. You may also need to think of threat neutralization strategies so that the threat does not recur.

5. *Asset recovery* For each critical asset or asset/threat pair, you should work out how that asset could be recovered in the event of a successful cyberattack. This may involve making extra hardware available or changing backup procedures to make it easier to access redundant copies of data.

6. *Asset reinstatement* This is a more general process of asset recovery where you define procedures to bring the system back into normal operation. Asset reinstatement should be concerned with all assets and not simply assets that are critical to the organization.

Information about all of these stages should be maintained in a cyber-resilience plan. This plan should be regularly updated, and, wherever possible, the strategies identified should be tested in mock attacks on the system.

Another important part of cyber-resilience planning is to decide how to support a flexible response in the event of a cyberattack. Paradoxically, resilience and security requirements often conflict. The aim of security is usually to limit privilege as far as possible so that users can only do what the security policy of the organization allows. However, to deal with problems, a user or system operator may have to take the initiative and take actions that are normally carried out by someone with a higher level of privilege.

For example, the system manager of a medical system may not normally be allowed to change the access rights of medical staff to records. For security reasons, access permissions have to be formally authorized, and two people need to be involved in making the change. This reduces the chances of system managers colluding with attackers and allowing access to confidential medical information.

Now, imagine that the system manager notices that a logged-in user is accessing a large number of records outside of normal working hours. The manager suspects

that an account has been compromised and that the user accessing the records is not actually the authorized user. To limit the damage, the user's access rights should be removed and a check then made with the authorized user to see if the accesses were actually illegal. However, the security procedures limiting the rights of system managers to change users' permissions make this impossible.

Resilience planning should take such situations into account. One way of doing so is to include an "emergency" mode in systems where normal checks are ignored. Rather than forbidding operations, the system logs what has been done and who was responsible. Therefore, the audit trail of emergency actions can be used to check that a system manager's actions were justified. Of course, there is scope for misuse here, and the existence of an emergency mode is itself a potential vulnerability. Therefore, organizations have to trade off possible losses against the benefits of adding more features to a system to support resilience.

14.2 Sociotechnical resilience

Fundamentally, resilience engineering is a sociotechnical rather than a technical activity. As I explained in Chapter 10, a sociotechnical system includes hardware, software, and people and is influenced by the culture, policies, and procedures of the organization that owns and uses the system. To design a resilient system, you have to think about sociotechnical systems design and not exclusively focus on software. Resilience engineering is concerned with adverse external events that can lead to system failure. Dealing with these events is often easier and more effective in the broader sociotechnical system.

For example, the Mentcare system maintains confidential patient data, and a possible external cyberattack may aim to steal that data. Technical safeguards such as authentication and encryption may be used to protect the data, but these are not effective if an attacker has access to the credentials of a genuine system user. You could try to solve this problem at the technical level by using more complex authentication procedures. However, these procedures annoy users and may lead to vulnerabilities as they write down authentication information. A better strategy may be to introduce organizational policies and procedures that emphasize the importance of not sharing login credentials and that tell users about easy ways to create and maintain strong passwords.

Resilient systems are flexible and adaptable so that they can cope with the unexpected. It is very difficult to create software that can adapt to cope with problems that have not been anticipated. However, as we saw from the Apollo 13 accident, people are very good at this. Therefore, to achieve resilience, you should take advantage of the fact that people are an inherent part of sociotechnical systems. Rather than try to anticipate and deal with all problems in software, you should leave some types of problem solving to the people responsible for operating and managing the software system.

To understand why you should leave some types of problem solving to people, you have to consider the hierarchy of sociotechnical systems that includes technical, software-intensive systems. Figure 14.3 shows that technical systems S1 and S2 are

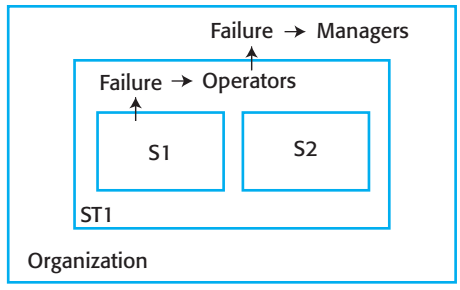


Figure 14.3 Nested technical and sociotechnical systems

part of a broader sociotechnical system ST1. That sociotechnical system includes operators who monitor the condition of S1 and S2 and who can take actions to resolve problems in these systems. If system S1 (say) fails, then the operators in ST1 may detect that failure and take recovery actions before the software failure leads to failure in the broader sociotechnical system. Operators may also invoke recovery and reinstatement procedures to get S1 back to its normal operating state.

Operational and management processes are the interface between the organization and the technical systems that are used. If these processes are well designed, they allow people to discover and to cope with technical system failures, as well as ensuring that operator errors are minimized. As I discuss in Section 14.2.2, rigid processes that are overautomated are not inherently resilient. They do not allow people to use their skills and knowledge to adapt and change processes to cope with the unexpected and deal with unanticipated failures.

The system ST1 is one of a number of sociotechnical systems in the organization. If the system operators cannot contain a technical system failure, then this may lead to a failure in the sociotechnical system ST1. Managers at the organizational level then must detect the problem and take steps to recover from it. Resilience is therefore an organizational as well as a system characteristic.

Hollnagel (Hollnagel 2010), who was an early advocate of resilience engineering, argues that it is important for organizations to study and learn from successes as well as failure. High-profile safety and security failures lead to inquiries and changes in practice and procedures. However, rather than respond to these failures, it is better to avoid them by observing how people deal with problems and maintain resilience. This good practice can then be disseminated throughout the organization. Figure 14.4 shows four characteristics that Hollnagel suggests reflect the resilience of an organization. These characteristics are:

1. *The ability to respond* Organizations have to be able to adapt their processes and procedures in response to risks. These risks may be anticipated risks, or they may be detected threats to the organization and its systems. For example, if a new security threat is detected and publicized, a resilient organization can make changes quickly so that this threat does not disrupt its operations.

2. *The ability to monitor* Organizations should monitor both their internal operations and their external environment for threats before they arise. For example, a company should monitor how its employees follow security policies.



Figure 14.4
Characteristics of resilient organizations

If potentially insecure behavior is detected, the company should respond by taking actions to understand why this has occurred and to change employee behavior.

3. *The ability to anticipate* A resilient organization should not simply focus on its current operations but should anticipate possible future events and changes that may affect its operations and resilience. These events may include technological innovations, changes in regulations or laws, and modifications in customer behavior. For example, wearable technology is starting to become available, and companies should now be thinking about how this might affect their current security policies and procedures.

4. *The ability to learn* Organizational resilience can be improved by learning from experience. It is particularly important to learn from successful responses to adverse events such as the effective resistance of a cyberattack. Learning from success allows good practice to be disseminated throughout the organization.

As Hollnagel says, to become resilient organizations have to address all of these issues to some extent. Some will focus more on one quality than others. For example, a company running a large-scale data center may focus mostly on monitoring and responsiveness. However, a digital library that manages long-term archival information may have to anticipate how future changes may affect its business as well as respond to any immediate security threats.

14.2.1 Human error

Early work on resilience engineering was concerned with accidents in safety-critical systems and with how the behavior of human operators could lead to safety-related system failures. This led to an understanding of system defenses that is equally applicable to systems that have to withstand malicious as well as accidental human actions.

We know that people make mistakes, and, unless a system is completely automated, it is inevitable that users and system operators will sometimes do the wrong thing. Unfortunately, these human errors sometimes lead to serious system failures. Reason (Reason, 2000) suggests that the problem of human error can be viewed in two ways:

1. *The person approach* Errors are considered to be the responsibility of the individual and "unsafe acts" (such as an operator failing to engage a safety barrier)

are a consequence of individual carelessness or reckless behavior. People who adopt this approach believe that human errors can be reduced by threats of disciplinary action, more stringent procedures, retraining, and so on. Their view is that the error is the fault of the individual responsible for making the mistake.

2. *The systems approach* The basic assumption is that people are fallible and will make mistakes. People make mistakes because they are under pressure from high workloads, because of poor training, or because of inappropriate system design. Good systems should recognize the possibility of human error and include barriers and safeguards that detect human errors and allow the system to recover before failure occurs. When a failure does occur, the best way to avoid its recurrence is to understand how and why the system defenses did not trap the error. Blaming and punishing the person who triggered the failure does not improve long-term system safety.

I believe that the systems approach is the right one and that systems engineers should assume that human errors will occur during system operation. Therefore, to improve the resilience of a system, designers have to think about the defenses and barriers to human error that could be part of a system. They should also think about whether these barriers should be built into the technical components of the system. If not, they could be part of the processes, procedures, and guidelines for using the system. For example, two operators may be required to check critical system inputs.

The barriers and safeguards that protect against human errors may be technical or sociotechnical. For example, code to validate all inputs is a technical defense; an approval procedure for critical system updates that needs two people to confirm the update is a sociotechnical defense. Using diverse barriers means that shared vulnerabilities are less likely and that a user error is more likely to be trapped before system failure.

In general, you should use redundancy and diversity to create a set of defensive layers (Figure 14.5), where each layer uses a different approach to deter attackers or to trap component failures or human errors. Dark blue barriers are software checks; light blue barriers are checks carried out by people.

As an example of this approach to defense in depth, some of the checks for controller errors that may be part of an air traffic control system include:

1. *A conflict alert warning as part of an air traffic control system* When a controller instructs an aircraft to change its speed or altitude, the system extrapolates its trajectory to see if it intersects with any other aircraft. If so, it sounds an alarm.

2. *Formalized recording procedures for air traffic management* The same ATC system may have a clearly defined procedure setting out how to record the control instructions that have been issued to aircraft. These procedures help controllers check if they have issued the instruction correctly and make the information visible to others for checking.

3. *Collaborative checking* Air traffic control involves a team of controllers who constantly monitor each other's work. When a controller makes a mistake, others usually detect and correct it before an incident occurs.

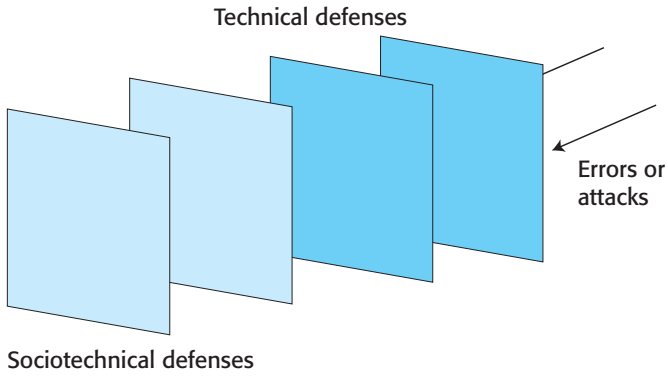


Figure 14.5 Defensive layers

Reason (Reason 2000) draws on the idea of defensive layers in a theory of how human errors lead to system failures. He introduces the so-called Swiss cheese model, which suggests that defensive layers are not solid barriers but are instead like slices of Swiss cheese. Some types of Swiss cheese, such as Emmenthal, have holes of varying sizes in them. Reason suggests that vulnerabilities, or what he calls latent conditions in the layers, are analogous to these holes.

These latent conditions are not static—they change depending on the state of the system and the people involved in system operation. To continue with the analogy, the holes change size and move around the defensive layers during system operation. For example, if a system relies on operators checking each other's work, a possible vulnerability is that both make the same mistake. This is unlikely under normal conditions so, in the Swiss cheese model, the hole is small. However, when the system is heavily loaded and the workload of both operators is high, then mistakes are more likely. The size of the hole representing this vulnerability increases.

Failure in a system with layered defenses occurs when there is some external trigger event that has the potential to cause damage. This event might be a human error (which Reason calls an active failure) or it could be a cyberattack. If all of the defensive barriers fail, then the system as a whole will fail. Conceptually, this corresponds to the holes in the Swiss cheese slices lining up, as shown in Figure 14.6.

This model suggests that different strategies can be used to increase system resilience to adverse external events:

1. Reduce the probability of the occurrence of an external event that might trigger system failures. To reduce human errors, you may introduce improved training for operators or give operators more control over their workload so that they are not overloaded. To reduce cyberattacks, you may reduce the number of people who have privileged system information and so reduce the chances of disclosure to an attacker.

2. Increase the number of defensive layers. As a general rule, the more layers that you have in a system, the less likely it is that the holes will line up and a system failure will occur. However, if these layers are not independent, then they may share a common vulnerability. Thus, the barriers are likely to have the same "hole" in the same place, so there is only a limited benefit in adding a new layer.

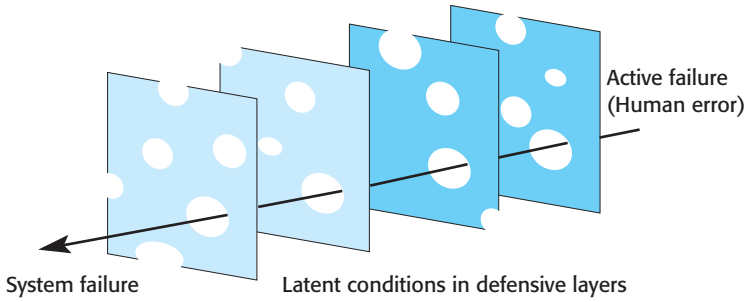


Figure 14.6 Reason's Swiss cheese model of system failure

3. Design a system so that diverse types of barriers are included. This means that the "holes" will probably be in different places, and so there is less chance of the holes lining up and failing to trap an error.

4. Minimize the number of latent conditions in a system. Effectively, this means reducing the number and size of system "holes." However, this may significantly increase systems engineering costs. Reducing the number of bugs in the system increases testing and V & V costs. Therefore, this option may not be cost-effective.

In designing a system, you need to consider all of these options and make choices about what might be the most cost-effective ways to improve the system's defenses. If you are building custom software, then using software checking to increase the number and diversity of layers may be the best option. However, if you are using off-the-shelf software, then you may have to consider how sociotechnical defenses may be added. You may decide to change training procedures to reduce the chances of problems occurring and to make it easier to deal with incidents when they arise.

14.2.2 Operational and management processes

All software systems have associated operational processes that reflect the assumptions of the designers about how these systems will be used. Some software systems, particularly those that control or are interfaced to special equipment, have trained operators who are an intrinsic part of the control system. Decisions are made during the design stage about which functions should be part of the technical system and which functions should be the operator's responsibility. For example, in an imaging system in a hospital, the operator may have the responsibility of checking the quality of the images immediately after they have been processed. This check allows the imaging procedure to be repeated if there is a problem.

Operational processes are the processes that are involved in using the system for its defined purpose. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs, and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system.

Most software systems, however, do not have trained operators but have system users, who use the system as part of their work or to support their personal interests. For personal systems, the designers may describe the expected use of the system but have no control over how users will actually behave. For enterprise IT systems, however, training may be provided for users to teach them how to use the system. Although user behavior cannot be controlled, it is reasonable to expect that they will normally follow the defined process.

Enterprise IT systems will also usually have system administrators or managers who are responsible for maintaining that system. While they are not part of the business process supported by the system, their job is to monitor the software system for errors and problems. If problems arise, system managers take action to resolve them and restore the system to its normal operational state.

In the previous section, I discussed the importance of defense in depth and the use of diverse mechanisms to check for adverse events that could lead to system failure. Operational and management processes are an important defense mechanism, and, in designing a process, you need to find a balance between efficient operation and problem management. These are often in conflict as shown in Figure 14.7 as increasing efficiency removes redundancy and diversity from a system.

Over the past 25 years, businesses have focused on so-called process improvement. To improve the efficiency of operational and management processes, companies study how their processes are enacted and look for particularly efficient and inefficient practice. Efficient practice is codified and documented, and software may be developed to support this "optimum" process. Inefficient practice is replaced by more efficient ways of doing things. Sometimes process control mechanisms are introduced to ensure that system operators and managers follow this "best practice."

The problem with process improvement is that it often makes it harder for people to cope with problems. What seems to be "inefficient" practice often arises because people maintain redundant information or share information because they know this makes it easier to deal with problems when things go wrong. For example, air traffic controllers may print flight details as well as rely on the flight database because they will then have information about flights in the air if the system database becomes unavailable.

People have a unique capability to respond effectively to unexpected situations, even when they have never had direct experience of these situations. Therefore, when things go wrong, operators and system managers can often recover the situation, although they may sometimes have to break rules and "work around" the defined process. You should therefore design operational processes to be flexible and adaptable. The operational processes should not be too constraining; they should not require operations to be done in a particular order; and the system software should not rely on a specific process being followed.

For example, an emergency service control room system is used to manage emergency calls and to initiate a response to these calls. The "normal" process of handling a call is to log the caller's details and then send a message to the appropriate emergency service giving details of the incident and the address. This procedure provides an audit trail of the actions taken. A subsequent investigation can check that the emergency call has been properly handled.

Efficient process operation	Problem management
Process optimization and control	Process flexibility and adaptability
Information hiding and security	Information sharing and visibility
Automation to reduce operator workload with fewer operators and managers	Manual processes and spare operator/manager capacity to deal with problems
Role specialization	Role sharing

Figure 14.7 Efficiency and resilience

Now imagine that this system is subject to a denial-of-service attack, which makes the messaging system unavailable. Rather than simply not responding to calls, the operators may use their personal mobile phones and their knowledge of call responders to call the emergency service units directly so that they can respond to serious incidents.

Management and provision of information are also important for resilient operation. To make a process more efficient, it may make sense to present operators with the information they need, when they need it. From a security perspective, information should not be accessible unless the operator or manager needs that information. However, a more liberal approach to information access can improve system resilience.

If operators are only presented with information that the process designer thinks they "need to know," then they may be unable to detect problems that do not directly affect their immediate tasks. When things go wrong, the system operators do not have a broad picture of what is happening in the system, so it is more difficult for them to formulate strategies for dealing with problems. If they cannot access some information in the system for security reasons, then they may be unable to stop attacks and repair the damage that has been caused.

Automating the system management process means that a single manager may be able to manage a large number of systems. Automated systems can detect common problems and take actions to recover from these problems. Fewer people are needed for system operations and management, and so costs are reduced. However, process automation has two disadvantages:

1. Automated management systems may go wrong and take incorrect actions. As problems develop, the system may take unexpected actions that make the situation worse and that cannot be understood by the system managers.

2. Problem solving is a collaborative process. If fewer managers are available, it is likely to take longer to work out a strategy to recover from a problem or cyberattack.

Therefore, process automation can have both positive and negative effects on system resilience. If the automated system works properly, it can detect problems, invoke cyberattack resistance if necessary, and start automated recovery procedures. However, if the automated system can't handle the problem, fewer people will be available to tackle the problem and the system may have been damaged by the process automation doing the wrong thing.

In an environment where there are different types of system and equipment, it may be impractical to expect all operators and managers to be able to deal with all of

the different systems. Individuals may therefore specialize so that they become expert and knowledgeable about a small number of systems. This leads to more efficient operation but has consequences for the resilience of the system.

The problem with role specialization is that there may not be anyone available at a particular time who understands the interactions between systems. Consequently, it is difficult to cope with problems if the specialist is not available. If people work with several systems, they come to understand the dependencies and relationships between them and so can tackle problems that affect more than one system. With no specialist available, it becomes much more difficult to contain the problem and repair any damage that has been caused.

You may use risk assessment, as discussed in Chapter 13, to help make decisions on the balance between process efficiency and resilience. You consider all of the risks where operator or manager intervention may be required and assess the likelihood of these risks and the extent of the possible losses that might arise. For risks that may lead to serious damage and extensive loss and for risks that are likely to occur, you should favor resilience over process efficiency.

14.3 Resilient systems design

Resilient systems can resist and recover from adverse incidents such as software failures and cyberattacks. They can deliver critical services with minimal interruptions and can quickly return to their normal operating state after an incident has occurred. In designing a resilient system, you have to assume that system failures or penetration by an attacker will occur, and you have to include redundant and diverse features to cope with these adverse events.

Designing systems for resilience involves two closely related streams of work:

1. *Identifying critical services and assets* Critical services and assets are those elements of the system that allow a system to fulfill its primary purpose. For example, the primary purpose of a system that handles ambulance dispatch in response to emergency calls is to get help to people who need it as quickly as possible. The critical services are those concerned with taking calls and dispatching ambulances to the medical emergency. Other services such as call logging and ambulance tracking are less important.

2. *Designing system components that support problem recognition, resistance, recovery, and reinstatement* For example, in an ambulance dispatch system, a watchdog timer (see Chapter 12) may be included to detect if the system is not responding to events. Operators may have to authenticate with a hardware token to resist the possibility of unauthorized access. If the system fails, calls may be diverted to another center so that the essential services are maintained. Copies of the system database and software on alternative hardware may be maintained to allow for reinstatement after an outage.

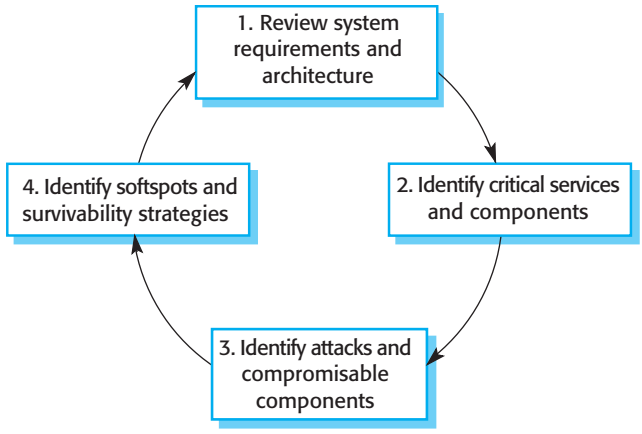


Figure 14.8 Stages in survivability analysis

The fundamental notions of recognition, resistance, and recovery were the basis of early work in resilience engineering by Ellison et al. (Ellison et al. 1999, 2002). They designed a method of analysis called survivable systems analysis. This method is used to assess vulnerabilities in systems and to support the design of system architectures and features that promote system survivability.

Survivable systems analysis is a four-stage process (Figure 14.8) that analyzes the current or proposed system requirements and architecture, identifies critical services, attack scenarios, and system "softspots," and proposes changes to improve the survivability of a system. The key activities in each of these stages are as follows:

1. *System understanding* For an existing or proposed system, review the goals of the system (sometimes called the mission objectives), the system requirements, and the system architecture.
2. *Critical service identification* The services that must always be maintained and the components that are required to maintain these services are identified.
3. *Attack simulation* Scenarios or use cases for possible attacks are identified, along with the system components that would be affected by these attacks.
4. *Survivability analysis* Components that are both essential and compromisable by an attack are identified, and survivability strategies based on resistance, recognition, and recovery are identified.

The fundamental problem with this approach to survivability analysis is that its starting point is the requirements and architecture documentation for a system. This is a reasonable assumption for defense systems (the work was sponsored by the U.S. Department of Defense), but it poses two problems for business systems:

1. It is not explicitly related to the business requirements for resilience. I believe that these are a more appropriate starting point than technical system requirements.

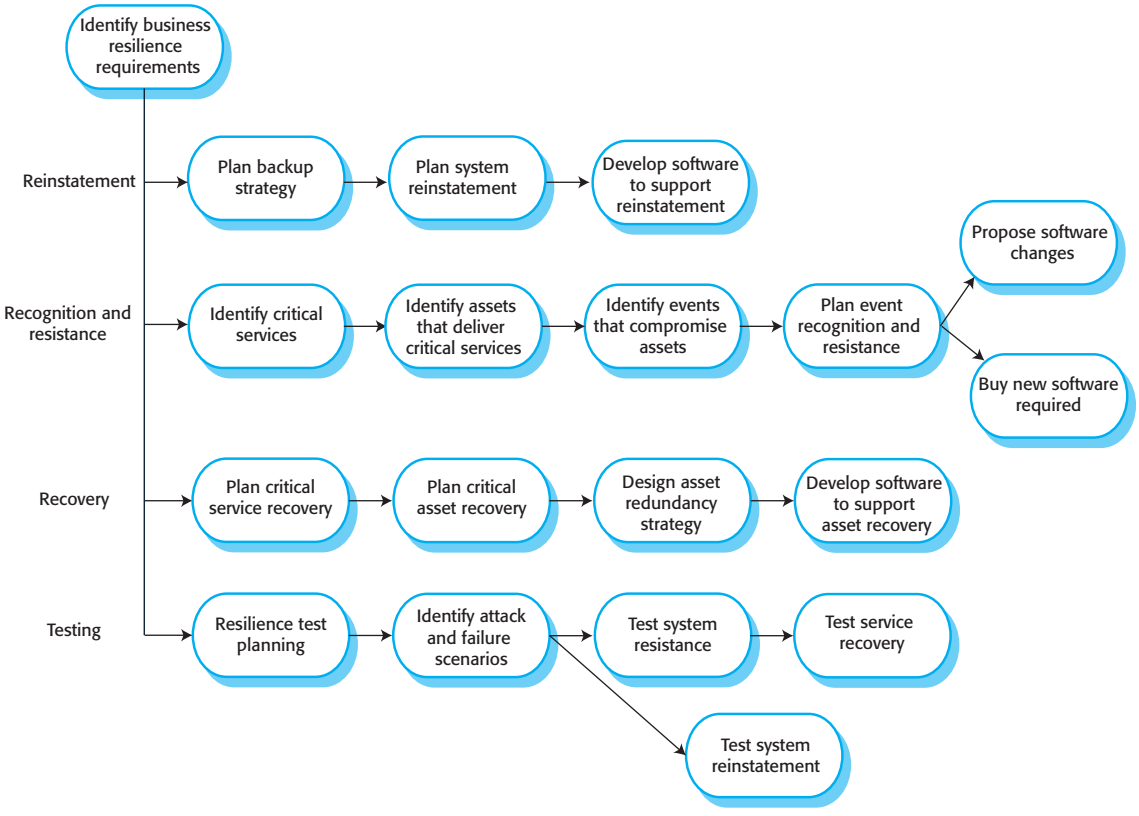


Figure 14.9
Resilience engineering

2. It assumes that there is a detailed requirements statement for a system. In fact, resilience may have to be "retrofitted" to a system where there is no complete or up-to-date requirements document. For new systems, resilience may itself be a requirement, or systems may be developed using an agile approach. The system architecture may be designed to take resilience into account.

A more general resilience engineering method, as shown in Figure 14.9, takes the lack of detailed requirements into account as well as explicitly designing recovery and reinstatement into the system. For the majority of components in a system, you will not have access to their source code and will not be able to make changes to them. Your strategy for resilience has to be designed with this limitation in mind.

There are five interrelated streams of work in this approach to resilience engineering:

1. You identify business resilience requirements. These requirements set out how the business as a whole must maintain the services that it delivers to customers and, from this, resilience requirements for individual systems are developed. Providing resilience is expensive, and it is important not to overengineer systems with unnecessary resilience support.
2. You plan how to reinstate a system or a set of systems to their normal operating state after an adverse event. This plan has to be integrated with the business's

normal backup and archiving strategy that allows recovery of information after a technical or human error. It should also be part of a wider disaster recovery strategy. You have to take account of the possibility of physical events such as fire and flooding and study how to maintain critical information in separate locations. You may decide to use cloud backups for this plan.

3. You identify system failures and cyberattacks that can compromise a system, and you design recognition and resilience strategies to cope with these adverse events.
4. You plan how to recover critical services quickly after they have been damaged or taken offline by a failure or cyberattack. This step usually involves providing redundant copies of the critical assets that provide these services and switching to these copies when required.
5. Critically, you should test all aspects of your resilience planning. This testing involves identifying failure and attack scenarios and playing these scenarios out against your system.

Maintaining the availability of critical services is the essence of resilience. Accordingly, you have to know:

- the system services that are the most critical for a business,
- the minimal quality of service that must be maintained,
- how these services might be compromised,
- how these services can be protected, and
- how you can recover quickly if the services become unavailable.

As part of the analysis of critical services, you have to identify the system assets that are essential for delivering these services. These assets may be hardware (servers, network, etc.), software, data, and people. To build a resilient system, you have to think about how to use redundancy and diversity to ensure that these assets remain available in the event of a system failure.

For all of these activities, the key to providing a rapid response and recovery plan after an adverse event is to have additional software that supports resistance, recovery, and reinstatement. This may be commercial security software or resilience support that is programmed into application systems. It may also include scripts and specially written programs that are developed for recovery and reinstatement. If you have the right support software, the processes of recovery and reinstatement can be partially automated and quickly invoked and executed after a system failure.

Resilience testing involves simulating possible system failures and cyberattacks to test whether the resilience plans that have been drawn up work as expected. Testing is essential because we know from experience that the assumptions made in resilience planning are often invalid and that planned actions do not always work. Testing for resilience can reveal these problems so that the resilience plan can be refined.

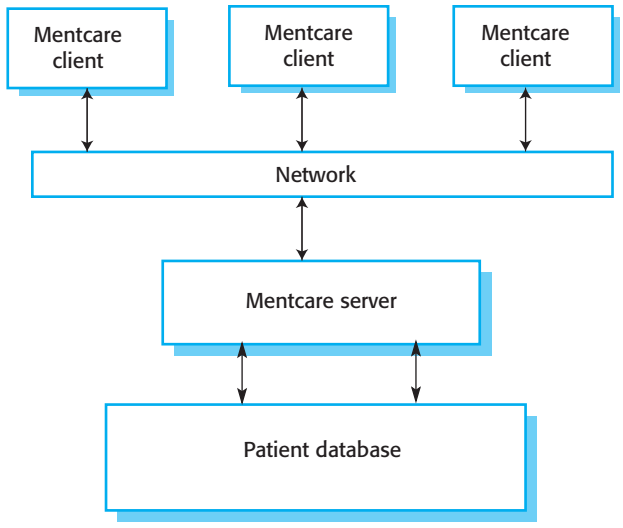


Figure 14.10 The client–server architecture of the Mentcare system

Testing can be very difficult and expensive as, obviously, the testing cannot be carried out on an operational system. The system and its environment may have to be duplicated for testing, and staff may have to be released from their normal responsibilities to work on the test system. To reduce costs, you can use "desk testing." The testing team assumes a problem has occurred and tests their reactions to it; they do not simulate that problem on a real system. While this approach can provide useful information about system resilience, it is less effective than testing in discovering deficiencies in the resilience plan.

As an example of this approach, let us look at resilience engineering for the Mentcare system. To recap, this system is used to support clinicians treating patients in a variety of locations who have mental health problems. It provides patient information and records of consultations with doctors and specialist nurses. It includes a number of checks that can flag patients who may be potentially dangerous or suicidal. Figure 14.10 shows the architecture of this system.

The system is consulted by doctors and nurses before and during a consultation, and patient information is updated after the consultation. To ensure the effectiveness of clinics, the business resilience requirements are that the critical system services are available during normal working hours, that the patient data should not be permanently damaged or lost by a system failure or cyberattack, and that patient information should not be released to unauthorized people.

Two critical services in the system have to be maintained:

1. *An information service* that provides information about a patient's current diagnosis and treatment plan.
2. *A warning service* that highlights patients who could pose a danger to others or to themselves.

Notice that the critical service is not the availability of the complete patient record. Doctors and nurses only need to go back to previous treatments occasionally,

so clinical care is not seriously affected if a full record is not available. Therefore, it is possible to deliver effective care using a summary record that only includes information about the patient and recent treatment.

The assets required to deliver these services in normal system operations are:

1. The patient record database that maintains all patient information.
2. A database server that provides access to the database for local client computers.
3. A network for client/server communications.
4. Local laptop or desktop computers used by clinicians to access patient information.
5. A set of rules that identify patients who are potentially dangerous and that can flag patient records. Client software highlights dangerous patients to system users.

To plan recognition, resistance, and recovery strategies, you need to develop a set of scenarios that anticipate adverse events that might compromise the critical services offered by the system. Examples of these adverse events are:

1. The unavailability of the database server either through a system failure, a network failure, or a denial-of-service cyberattack.
2. The deliberate or accidental corruption of the patient record database or the rules that define what is meant by a "dangerous patient."
3. Infection of client computers with malware.
4. Access to client computers by unauthorized people who gain access to patient records.

Figure 14.11 shows possible recognition and resistance strategies for these adverse events. Notice that these are not just technical approaches but also include workshops to inform system users about security issues. We know that many security breaches arise because users inadvertently reveal privileged information to an attacker and these workshops reduce the chances of this happening. I don't have space here to discuss all of the options that I identified in Figure 14.11. Instead, I focus on how the system architecture can be modified to be more resilient.

In Figure 14.11, I suggested that maintaining patient information on client computers was a possible redundancy strategy that could help maintain critical services. This leads to the modified software architecture shown in Figure 14.12. The key features of this architecture are:

1. *Summary patient records that are maintained on local client computers* The local computers can communicate directly with each other and exchange information using either the system network or, if necessary, an ad hoc network created using mobile phones. Therefore, if the database is unavailable, doctors and nurses can still access essential patient information. (resistance and recovery)
2. *A backup server to allow for main server failure* This server is responsible for taking regular snapshots of the database as backups. In the event the main server

Event	Recognition	Resistance
Server unavailability	1. Watchdog timer on client that times out if no response to client access 2. Text messages from system managers to clinical users	1. Design system architecture to maintain local copies of critical information 2. Provide peer-to-peer search across clients for patient data 3. Provide staff with smartphones that can be used to access the network in the event of server failure 4. Provide backup server
Patient database corruption	1. Record level cryptographic checksums 2. Regular auto-checking of database integrity 3. Reporting system for incorrect information	1. Replayable transaction log to update database backup with recent transactions 2. Maintenance of local copies of patient information and software to restore database from local copies and backups
Malware infection of client computers	1. Reporting system so that computer users can report unusual behavior 2. Automated malware checks on startup	1. Security awareness workshops for all system users 2. Disabling of USB ports on client computers 3. Automated system setup for new clients 4. Support access to system from mobile devices 5. Installation of security software
Unauthorized access to patient information	1. Warning text messages from users about possible intruders 2. Log analysis for unusual activity	1. Multilevel system authentication process 2. Disabling of USB ports on client computers 3. Access logging and real-time log analysis 4. Security awareness workshops for all system users

Figure 14.11 Recognition and resistance strategies for adverse events

fails, it can also act as the main server for the whole system. This provides continuity of service and recovery after a server failure (resistance and recovery).

3. *Database integrity checking and recovery software* Integrity checking runs as a background task checking for signs of database corruption. If corruption is discovered, it can automatically initiate the recovery of some or all of the data from backups. The transaction log allows these backups to be updated with details of recent changes (recognition and recovery).

To maintain the key services of patient information access and staff warning, we can make use of the inherent redundancy in a client-server system. By downloading information to the client at the start of a clinic session, the consultation can continue without server access. Only the information about the patients who are scheduled to attend consultations that day needs to be downloaded. If there is a need to access other patient information and the server is unavailable, then other client computers may be contacted using peer-to-peer communication to see if the information is available on them.

The service that provides a warning to staff of patients who may be dangerous can easily be implemented using this approach. The records of patients who may harm themselves or others are identified before the download process. When clinical staff access these records, the software can highlight the records to indicate the patients that require special care.

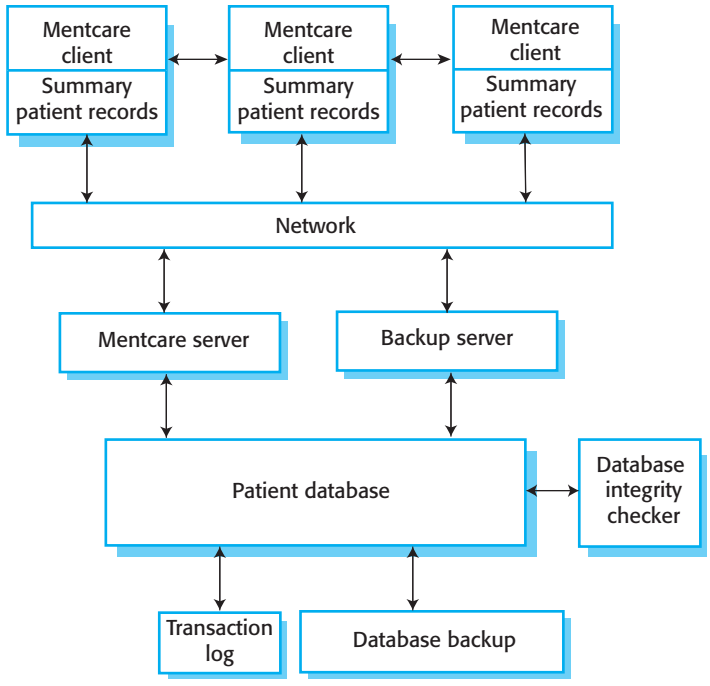


Figure 14.12 An architecture for Mentcare system resilience

The features in this architecture that support the resistance to adverse events are also useful in supporting recovery from these events. By maintaining multiple copies of information and having backup hardware available, critical system services can be quickly restored to normal operation. Because the system need only be available during normal working hours (say, 8 a.m to 6 p.m), the system can be reinstated overnight so that it is available for the following day after a failure.

As well as maintaining critical services, the other business requirements of maintaining the confidentiality and integrity of patient data must also be supported. The architecture shown in Figure 14.12 includes a backup system and explicit database integrity checking to reduce the chances that patient information is damaged accidentally or in a malicious attack. Information on client computers is also available and can be used to support recovery from data corruption or damage.

While maintaining multiple copies of data is a safeguard against data corruption, it poses a risk to confidentiality as all of these copies have to be secured. In this case, this risk can be controlled by:

1. Only downloading the summary records of patients who are scheduled to attend a clinic. This limits the number of records that could be compromised.
2. Encrypting the disk on local client computers. Attackers who do not have the encryption key cannot read the disk if they gain access to the computer.
3. Securely deleting the downloaded information at the end of a clinic session. This further reduces the chances of an attacker gaining access to confidential information.

4. Ensuring that all network transactions are encrypted. If an attacker intercepts these transactions, they cannot get access to the information.

Because of performance degradation, it is probably impractical to encrypt the entire patient database on the server. Strong authentication should therefore be used to protect this information.

KEY POINTS

- The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.
- Resilience should be based on the 4 Rs model—recognition, resistance, recovery, and reinstatement.
- Resilience planning should be based on the assumption that networked systems will be subject to cyberattacks by malicious insiders and outsiders and that some of these attacks will be successful.
- Systems should be designed with a number of defensive layers of different types. If these layers are effective, human and technical failures can be trapped and cyberattacks resisted.
- To allow system operators and managers to cope with problems, processes should be flexible and adaptable. Process automation can make it more difficult for people to cope with problems.
- Business resilience requirements should be the starting point for designing systems for resilience. To achieve system resilience, you have to focus on recognition and recovery from problems, recovery of critical services and assets, and reinstatement of the system.
- An important part of design for resilience is identifying critical services, which are those services that are essential if a system is to ensure its primary purpose. Systems should be designed so that these services are protected and, in the event of failure, recovered as quickly as possible.

FURTHER READING

"Survivable Network System Analysis: A Case Study." An excellent paper that introduces the notion of system survivability and uses a case study of a mental health record treatment system to illustrate the application of a survivability method. (R. J. Ellison, R. C. Linger, T. Longstaff, and N. R. Mead, *IEEE Software*, 16 (4), July/August 1999) http://dx.doi.org/10.1109/52.776952

Resilience Engineering in Practice: A Guidebook. This is a collection of articles and case studies on resilience engineering that takes a broad, sociotechnical systems perspective. (E. Hollnagel, J. Paries, D. W. Woods, and J. Wreathall, Ashgate Publishing Co., 2011).

"Cyber Risk and Resilience Management." This is a website with a wide range of resources on cybersecurity and resilience, including a model for resilience management. (Software Engineering Institute, 2013) https://www.cert.org/resilience/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/security-and-resilience/

EXERCISES

14.1. Explain how the complementary strategies of resistance, recognition, recovery, and reinstatement may be used to provide system resilience.

14.2. What are the types of threats that have to be considered in resilience planning? Provide examples of the controls that organizations should put in place to counter those threats.

14.3. Describe the ways in which human error can be viewed according to Reason (Reason, 2000) and the strategies that can be used to increase resilience according to the Swiss cheese model (Figure 14.6).

14.4. A hospital proposes to introduce a policy that any member of clinical staff (doctors or nurses) who takes or authorizes actions that leads to a patient being injured will be subject to criminal charges. Explain why this is a bad idea, which is unlikely to improve patient safety, and why it is likely to adversely affect the resilience of the organization.

14.5. What is survivable systems analysis and what are the key activities in each of the four stages involved in it as shown in Figure 14.8?

14.6. Explain why process inflexibility can inhibit the ability of a sociotechnical system to resist and recover from adverse events such as cyberattacks and software failure. If you have experience of process inflexibility, illustrate your answer with examples from your experience.

14.7. Suggest how the approach to resilience engineering that I proposed in Figure 14.9 could be used in conjunction with an agile development process for the software in the system. What problems might arise in using agile development for systems where resilience is important?

14.8. In Section 13.4.2, (1) an unauthorized user places malicious orders to move prices and (2) an intrusion corrupts the database of transactions that have taken place. For each of these cyberattacks, identify resistance, recognition, and recovery strategies that might be used.

14.9. In Figure 14.11, I suggested a number of adverse events that could affect the Mentcare system. Draw up a test plan for this system that sets out how you could test the ability of the Mentcare system to recognize, resist, and recover from these events.

14.10. A senior manager in a company is concerned about insider attacks from disaffected staff on the company's IT assets. As part of a resilience improvement program, she proposes that a logging system and data analysis software be introduced to capture and analyze all employee actions but that employees should not be told about this system. Discuss the ethics of both introducing a logging system and doing so without telling system users.

REFERENCES

Ellison, R. J., R. C. Linger, T. Longstaff, and N. R. Mead. 1999. "Survivable Network System Analysis: A Case Study." *IEEE Software* 16 (4): 70–77. doi:10.1109/52.776952.

Ellison, R. J., R. C. Linger, H. Lipson, N. R. Mead, and A. Moore. 2002. "Foundations of Survivable Systems Engineering." *Crosstalk: The Journal of Defense Software Engineering* 12: 10–15. http://resources.sei.cmu.edu/asset_files/WhitePaper/2002_019_001_77700.pdf

Hollnagel, E. 2006. "Resilience—the Challenge of the Unstable." In *Resilience Engineering: Concepts and Precepts*, edited by E. Hollnagel, D. D. Woods, and N.G. Leveson, 9–18.

———. 2010. "RAG—The Resilience Analysis Grid." In *Resilience Engineering in Practice*, edited by E. Hollnagel, J. Paries, D. Woods, and J. Wreathall, 275–295. Farnham, UK: Ashgate Publishing Group.

InfoSecurity. 2013. "Global Cybercrime, Espionage Costs $100–$500 Billion Per Year." http://www.infosecurity-magazine.com/view/33569/global-cybercrime-espionage-costs-100500-billion-per-year

Laprie, J-C. 2008. "From Dependability to Resilience." In *38th Int. Conf. on Dependable Systems and Networks*. Anchorage, Alaska. http://2008.dsn.org/fastabs/dsno8fastabs_laprie.pdf

Reason, J. 2000. "Human Error: Models and Management." *British Medical J.* 320: 768–770. doi:10.1136/bmj.320.7237.768.



PART 3 Advanced Software Engineering

This part of the book covers more advanced software engineering topics. I assume in these chapters that readers understand the basics of the discipline, covered in Chapters 1–9.

Chapters 15–18 focus on the dominant development paradigm for web-based information systems and enterprise systems—software reuse. Chapter 15 introduces the topic and explains the different types of reuse that are possible. I then cover the most common approach to reuse, which is the reuse of application systems. These are configured and adapted to the specific needs of each business.

Chapter 16 is concerned with the reuse of software components rather than entire software systems. In this chapter, I explain what is meant by a component and why standard component models are needed for effective component reuse. I also discuss the general process of component-based software engineering and the problems of component composition.

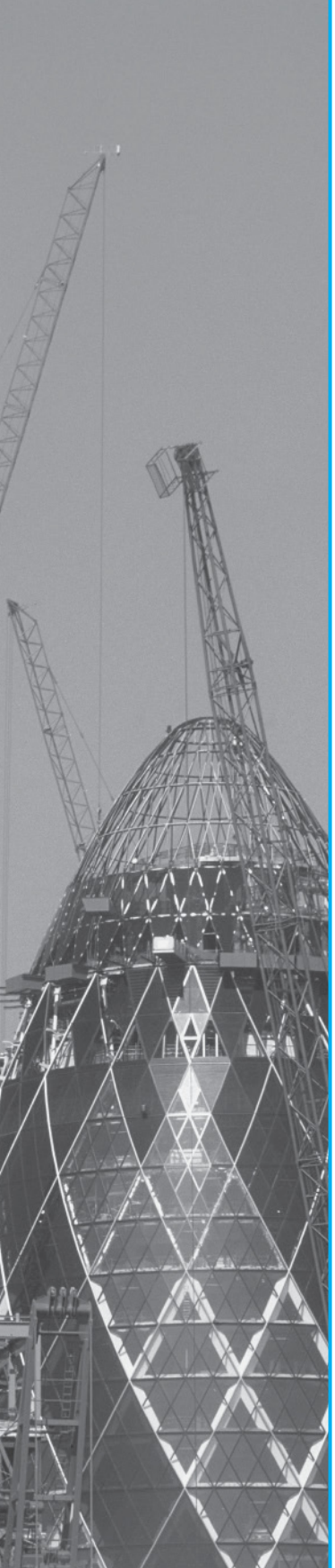
The majority of large systems are now distributed systems and Chapter 17 covers issues and problems of building distributed systems. I introduce the client-server approach as a fundamental paradigm of distributed systems engineering, and explain ways of implementing this architectural style. The final section explains software as a service–the delivery of software functionality over the Internet, which has changed the market for software products.

Chapter 18 introduces the related topic of service-oriented architectures, which link the notions of distribution and reuse. Services are reusable software components whose functionality can be accessed over the Internet. I discuss two widely-used approaches to service development namely SOAP-based and RESTful services. I explain what is involved in creating services (service engineering) and composing services to create new software systems.

The focus of Chapters 19–21 is systems engineering. In Chapter 19, I introduce the topic and explain why it is important that software engineers should understand systems engineering. I discuss the systems engineering life cycle and the importance of procurement in that life-cycle.

Chapter 20 covers systems of systems (SoS). The large systems that we will build in the 21st century will not be developed from scratch but will be created by integrating existing complex systems. I explain why an understanding of complexity is important in SoS development and discuss architectural patterns for complex systems of systems.

Most software systems are not apps or business systems but are embedded real-time systems. Chapter 21 covers this important topic. I introduce the idea of a real-time embedded system and describe architectural patterns that are used in embedded systems design. I then explain the process of timing analysis and conclude the chapter with a discussion of real-time operating systems.



15 Software reuse

Objectives

The objectives of this chapter are to introduce software reuse and to describe approaches to system development based on large-scale software reuse. When you have read this chapter, you will:

- understand the benefits and problems of reusing software when developing new systems;
- understand the concept of an application framework as a set of reusable objects and how frameworks can be used in application development;
- have been introduced to software product lines, which are made up of a common core architecture and reusable components that are configured for each version of the product;
- have learned how systems can be developed by configuring and composing off-the-shelf application software systems.

Contents

15.1 The reuse landscape
15.2 Application frameworks
15.3 Software product lines
15.4 Application system reuse

Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software. Until around 2000, systematic software reuse was uncommon, but it is now used extensively in the development of new business systems. The move to reuse-based development has been in response to demands for lower software production and maintenance costs, faster delivery of systems, and increased software quality. Companies see their software as a valuable asset. They are promoting reuse of existing systems to increase their return on software investments.

Reusable software of different kinds is now widely available. The open-source movement has meant that there is a huge code base that can be reused. This may be in the form of program libraries or entire applications. Many domain-specific application systems, such as ERP systems, are available that can be tailored and adapted to customer requirements. Some large companies provide a range of reusable components for their customers. Standards, such as web service standards, have made it easier to develop software services and reuse them across a range of applications.

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. For example:

1. *System reuse* Complete systems, which may be made up of a number of application programs, may be reused as part of a system of systems (Chapter 20).
2. *Application reuse* An application may be reused by incorporating it without change into other systems or by configuring the application for different customers. Alternatively, application families or software product lines that have a common architecture, but that are adapted to individual customer requirements, may be used to develop a new system.
3. *Component reuse* Components of an application, ranging in size from subsystems to single objects, may be reused. For example, a pattern-matching system developed as part of a text-processing system may be reused in a database management system. Components may be hosted on the cloud or on private servers and may be accessible through an application programming interface (API) as services.
4. *Object and function reuse* Software components that implement a single function, such as a mathematical function, or an object class may be reused. This form of reuse, designed around standard libraries, has been common for the past 40 years. Many libraries of functions and classes are freely available. You reuse the classes and functions in these libraries by linking them with newly developed application code. In areas such as mathematical algorithms and graphics, where specialized, expensive expertise is needed to develop efficient objects and functions, reuse is particularly cost-effective.

All software systems and components that include generic functionality are potentially reusable. However, these systems or components are sometimes so

Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Reduced process risk	The cost of existing software is already known, while the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is especially true when large software components such as subsystems are reused.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Figure 15.1 Benefits of software reuse

specific that it is very expensive to modify them for a new situation. Rather than reuse the code, however, you can reuse the ideas that are the basis of the software. This is called concept reuse.

In concept reuse you do not reuse a software component; rather, you reuse an idea, a way of working, or an algorithm. The concept that you reuse is represented in an abstract notation, such as a system model, which does not include implementation detail. It can, therefore, be configured and adapted for a range of situations. Concept reuse is embodied in approaches such as design patterns (Chapter 7), configurable system products, and program generators. When concepts are reused, the reuse process must include an activity where the abstract concepts are instantiated to create executable components.

An obvious advantage of software reuse is that overall development costs are lower. Fewer software components need to be specified, designed, implemented, and validated. However, cost reduction is only one benefit of software reuse. I have listed other advantages of reusing software in Figure 15.1.

However, there are costs and difficulties associated with reuse (Figure 15.2). There is a significant cost associated with understanding whether or not a component is suitable for reuse in a particular situation, and in testing that component to ensure its dependability. These additional costs mean that the savings in development costs may not be less than anticipated. However, the other benefits of reuse still apply.

Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood, and sometimes adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become incompatible with changes made to the system.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is more likely to be the case for tools that support embedded systems engineering than for object-oriented development tools.
"Not-invented-here" syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

Figure 15.2 Problems with software reuse

As I discussed in Chapter 2, software development processes have to be adapted to take reuse into account. In particular, there has to be a requirements refinement stage where the requirements for the system are modified to reflect the reusable software that is available. The design and implementation stages of the system may also include explicit activities to look for and evaluate candidate components for reuse.

15.1 The reuse landscape

Over the past 20 years, many techniques have been developed to support software reuse. These techniques exploit the facts that systems in the same application domain are similar and have potential for reuse, that reuse is possible at different levels from simple functions to complete applications, and that standards for reusable components facilitate reuse. Figure 15.3 shows the "reuse landscape"—different ways of implementing software reuse. Each of these approaches to reuse is briefly described in Figure 15.4.

Given this array of techniques for reuse, the key question is "which is the most appropriate technique to use in a particular situation?" Obviously, the answer to this question depends on the requirements for the system being developed, the technology

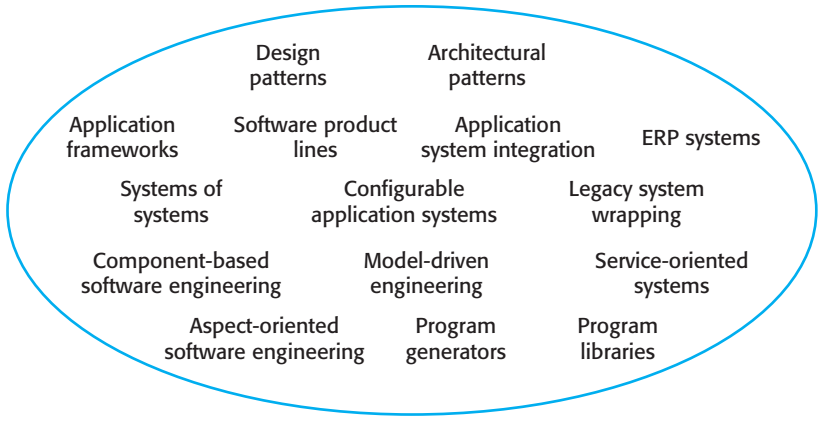


Figure 15.3 The reuse landscape

and reusable assets available, and the expertise of the development team. Key factors that you should consider when planning reuse are:

1. *The development schedule for the software* If the software has to be developed quickly, you should try to reuse complete systems rather than individual components. Although the fit to requirements may be imperfect, this approach minimizes the amount of development required.

2. *The expected software lifetime* If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications.

   Over its lifetime, you will have to adapt the system to new requirements, which will mean making changes to parts of the system. If you do not have access to the source code of the reusable components, you may prefer to avoid off-the-shelf components and systems from external suppliers. These suppliers may not be able to continue support for the reused software. You may decide that it is safer to reuse open-source systems and components (Chapter 7) as this means you can access and keep copies of the source code.

3. *The background, skills and experience of the development team* All reuse technologies are fairly complex, and you need quite a lot of time to understand and use them effectively. Therefore, you should focus your reuse effort in areas where your development team has expertise.

4. *The criticality of the software and its non-functional requirements* For a critical system that has to be certified by an external regulator you may have to create a safety or security case for the system (discussed in Chapter 12). This is difficult if you don't have access to the source code of the software. If your software has stringent performance requirements, it may be impossible to use strategies such as model-driven engineering (MDE) (Chapter 5). MDE relies on generating code from a reusable domain-specific model of a system. However, the code generators used in MDE often generate relatively inefficient code.

Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality.
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11, and 17.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in web Chapter 31.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 16.
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Legacy system wrapping	Legacy systems (Chapter 9) are "wrapped" by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models, and code is generated from these models. Described in Chapter 5.
Program generators	A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 18.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	Two or more distributed systems are integrated to create a new system. Described in Chapter 20.

Figure 15.4 Approaches that support software reuse

5. *The application domain* In many application domains, such as manufacturing and medical information systems, there are generic products that may be reused by configuring them to a local situation. This is one of the most effective approaches to reuse, and it is almost always cheaper to buy rather than build a new system.



Generator-based reuse

Generator-based reuse involves incorporating reusable concepts and knowledge into automated tools and providing an easy way for tool users to integrate specific code with this generic knowledge. This approach is usually most effective in domain-specific applications. Known solutions to problems in that domain are embedded in the generator system and selected by the user to create a new system.

http://software-engineering-book.com/web/generator-reuse/

6. *The platform on which the system will run* Some components models, such as .NET, are specific to Microsoft platforms. Similarly, generic application systems may be platform-specific, and you may only be able to reuse these if your system is designed for the same platform.

The range of available reuse techniques is such that, in most situations, there is the possibility of some software reuse. Whether or not reuse is achieved is often a managerial rather than a technical issue. Managers may be unwilling to compromise their requirements to allow reusable components to be used. They may not understand the risks associated with reuse as well as they understand the risks of original development. Although the risks of new software development may be higher, some managers may prefer known risks of development to unknown risks of reuse. To promote company-wide reuse, it may be necessary to introduce a reuse program that focuses on the creation of reusable assets and processes to facilitate reuse (Jacobsen, Griss, and Jonsson 1997).

15.2 Application frameworks

Early enthusiasts for object-oriented development suggested that one of the key benefits of using an object-oriented approach was that objects could be reused in different systems. However, experience has shown that objects are often too fine-grained and are often specialized for a particular application. It often takes longer to understand and adapt the object than to reimplement it. It has now become clear that object-oriented reuse is best supported in an object-oriented development process through larger-grain abstractions called frameworks.

As the name suggests, a framework is a generic structure that is extended to create a more specific subsystem or application. Schmidt et al. (Schmidt et al. 2004) define a framework to be

> *an integrated set of software artifacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.*[†]

Frameworks provide support for generic features that are likely to be used in all applications of a similar type. For example, a user interface framework will provide support

[†]Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." ACM Queue 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

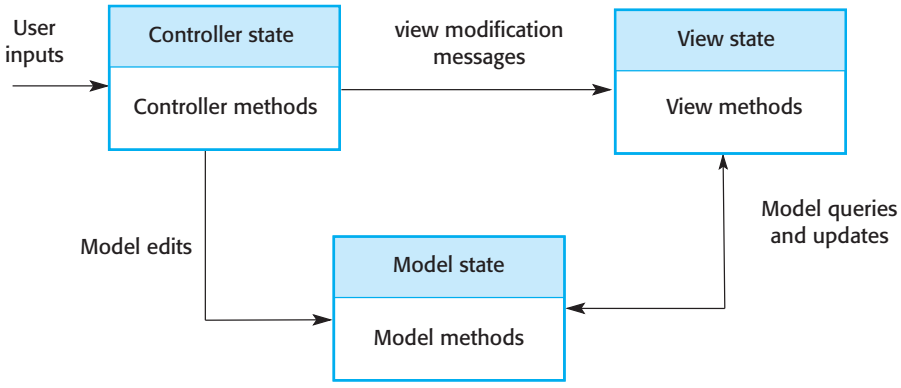


Figure 15.5 The Model-View-Controller pattern

for interface event handling and will include a set of widgets that can be used to construct displays. It is then left to the developer to specialize these by adding specific functionality for a particular application. For example, in a user interface framework, the developer defines display layouts that are appropriate to the application being implemented.

Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system. The architecture is implemented by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance and polymorphism.

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. Frameworks are available in commonly used object-oriented programming languages such as Java, C#, and C++, as well as in dynamic languages such as Ruby and Python. In fact, a framework can incorporate other frameworks, where each framework is designed to support the development of part of the application. You can use a framework to create a complete application or to implement part of an application, such as the graphical user interface.

The most widely used application frameworks are web application frameworks (WAFs), which support the construction of dynamic websites. The architecture of a WAF is usually based on the Model-View-Controller (MVC) Composite pattern shown in Figure 15.5. The MVC pattern was originally proposed in the 1980s as an approach to GUI design that allowed for multiple presentations of an object and separate styles of interaction with each of these presentations. In essence, it separates the state from its presentation so that the state may be updated from each presentation.

An MVC framework supports the presentation of data in different ways and allows interaction with each of these presentations. When the data is modified through one of the presentations, the system model is changed and the controllers associated with each view update their presentation.

Frameworks are often implementations of design patterns, as discussed in Chapter 7. For example, an MVC framework includes the Observer pattern, the Strategy pattern, the Composite pattern, and a number of others that are discussed by Gamma et al. (Gamma et al. 1995). The general nature of patterns and their use of abstract and concrete classes allow for extensibility. Without patterns, frameworks would almost certainly be impractical.

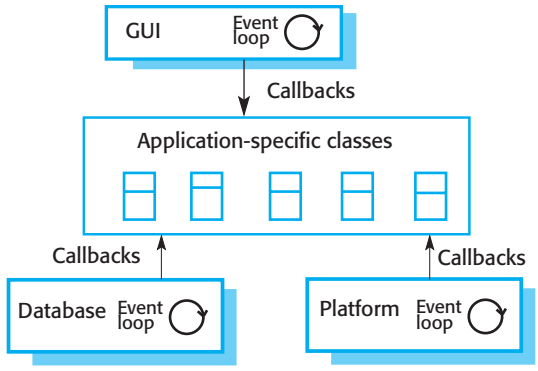


Figure 15.6 Inversion of control in frameworks

While each framework includes slightly different functionality, web application frameworks usually provide components and classes that support:

1. *Security* WAFs may include classes to help implement user authentication (login) and access control to ensure that users can only access permitted functionality in the system.
2. *Dynamic web pages* Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.
3. *Database integration* Frameworks don't usually include a database but assume that a separate database, such as MySQL, will be used. The framework may include classes that provide an abstract interface to different databases.
4. *Session management* Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
5. *User interaction* Web frameworks provide AJAX (Holdener 2008) and/or HTML5 support (Sarris 2013), which allows interactive web pages to be created. They may include classes that allow device-independent interfaces to be created, which adapt automatically to mobile phones and tablets.

To implement a system using a framework, you add concrete classes that inherit operations from abstract classes in the framework. In addition, you define "callbacks"—methods that are called in response to events recognized by the framework. The framework objects, rather than the application-specific objects, are responsible for control in the system. Schmidt et al. (Schmidt, Gokhale, and Natarajan 2004) call this "inversion of control."

In response to events from the user interface and database framework objects invoke "hook methods" that are then linked to user-provided functionality. The user-provided functionality defines how the application should respond to the event (Figure 15.6). For example, a framework will have a method that handles a mouse click from the environment. This method is called the hook method, which you must configure to call the appropriate application methods to handle the mouse click.

Fayad and Schmidt (Fayad and Schmidt 1997) discuss three other classes of framework:

1. *System infrastructure frameworks* support the development of system infrastructures such as communications, user interfaces, and compilers.
2. *Middleware integration frameworks* consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include Microsoft's .NET and Enterprise Java Beans (EJB). These frameworks provide support for standardized component models, as discussed in Chapter 16.
3. *Enterprise application frameworks* are concerned with specific application domains such as telecommunications or financial systems (Baumer et al. 1997). These embed application domain knowledge and support the development of end-user applications. These are not now widely used and have been largely superseded by software product lines.[†]

Applications that are constructed using frameworks can be the basis for further reuse through the concept of software product lines or application families. Because these applications are constructed using a framework, modifying family members to create instances of the system is often a straightforward process. It involves rewriting concrete classes and methods that you have added to the framework.

Frameworks are a very effective approach to reuse. However, they are expensive to introduce into software development processes as they are inherently complex and it can take several months to learn to use them. It can be difficult and expensive to evaluate available frameworks to choose the most appropriate one. Debugging framework-based applications is more difficult than debugging original code because you may not understand how the framework methods interact. Debugging tools may provide information about the reused framework components, which the developer does not understand.

15.3 Software product lines

When a company has to support a number of similar but not identical systems, one of the most effective approaches to reuse is to create a software product line. Hardware control systems are often developed using this approach to reuse as are domain-specific applications in areas such as logistics or medical systems. For example, a printer manufacturer has to develop printer control software, where there is a specific version of the product for each type of printer. These software versions have much in common, so it makes sense to create a core product (the product line) and adapt this for each printer type.

A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect specific customer requirements. The core system is designed so that it can be configured and adapted to

[†]Fayad, M. E., and D. C. Schmidt. 1997. "Object-Oriented Application Frameworks." Comm. ACM 40 (10): 32–38. doi:10.1145/262793.262798.

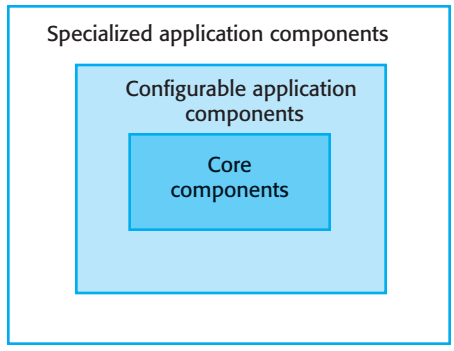


Figure 15.7 The organization of a base system for a product line

suit the needs of different customers or equipment. This may involve the configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements.

Developing applications by adapting a generic version of the application means that a high proportion of the application code is reused in each system. Testing is simplified because tests for large parts of the application may also be reused, thus reducing the overall application development time. Engineers learn about the application domain through the software product line and so become specialists who can work quickly to develop new applications.

Software product lines usually emerge from existing applications. That is, an organization develops an application and then, when a similar system is required, informally reuses code from this in the new application. The same process is used as other similar applications are developed. However, change tends to corrupt application structure so, as more new instances are developed, it becomes increasingly difficult to create a new version. Consequently, a decision to design a generic product line may then be made. This involves identifying common functionality in product instances and developing a base application, which is then used for future development.

This base application (Figure 15.7) is designed to simplify reuse and reconfiguration. Generally, a base application includes:

1. Core components that provide infrastructure support. These are not usually modified when developing a new instance of the product line.
2. Configurable components that may be modified and configured to specialize them to a new application. Sometimes it is possible to reconfigure these components without changing their code by using a built-in component configuration language.
3. Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.

Application frameworks and software product lines have much in common. They both support a common architecture and components, and require new development to create a specific version of a system. The main differences between these approaches are as follows:

1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework

code is not modified, and the possible modifications are limited to whatever is supported by the framework. Software product lines are not necessarily created using an object-oriented approach. Application components are changed, deleted, or rewritten. There are no limits, in principle at least, to the changes that can be made.

2. Most application frameworks provide general support rather than domain-specific support. For example, there are application frameworks to create web-based applications. A software product line usually embeds detailed domain and platform information. For example, there could be a software product line concerned with web-based applications for health record management.

3. Software product lines are often control applications for equipment. For example, there may be a software product line for a family of printers. This means that the product line has to provide support for hardware interfacing. Application frameworks are usually software-oriented, and they do not usually include hardware interaction components.

4. Software product lines are made up of a family of related applications, owned by the same organization. When you create a new application, your starting point is often the closest member of the application family, not the generic core application.

If you are developing a software product line using an object-oriented programming language, then you may use an application framework as a basis for the system. You create the core of the product line by extending the framework with domain-specific components using its built-in mechanisms. There is then a second phase of development where versions of the system for different customers are created. For example, you can use a web-based framework to build the core of a software product line that supports web-based help desks. This "help desk product line" may then be further specialized to provide particular types of help desk support.

The architecture of a software product line often reflects a general, application-specific architectural style or pattern. For example, consider a product-line system that is designed to handle vehicle dispatching for emergency services. Operators of this system take calls about incidents, find the appropriate vehicle to respond to the incident, and dispatch the vehicle to the incident site. The developers of such a system may market versions of it for police, fire, and ambulance services.

This vehicle dispatching system is an example of a generic resource allocation and management architecture (Figure 15.8). Resource management systems use a database of available resources and include components to implement the resource allocation policy that has been decided by the company using the system. Users interact with a resource management system to request and release resources and to ask questions about resources and their availability.

You can see how this four-layer structure may be instantiated in Figure 15.9, which shows the modules that might be included in a vehicle dispatching system product line. The components at each level in the product-line system are as follows:

1. At the interaction level, components provide an operator display interface and an interface with the communications systems used.

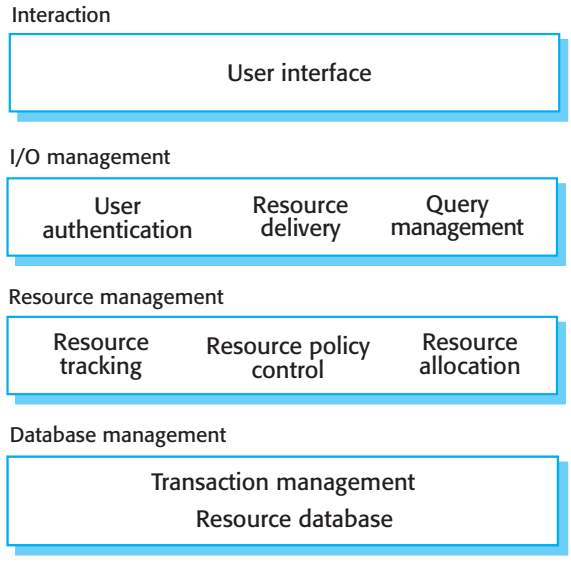


Figure 15.8 The architecture of a resource management system

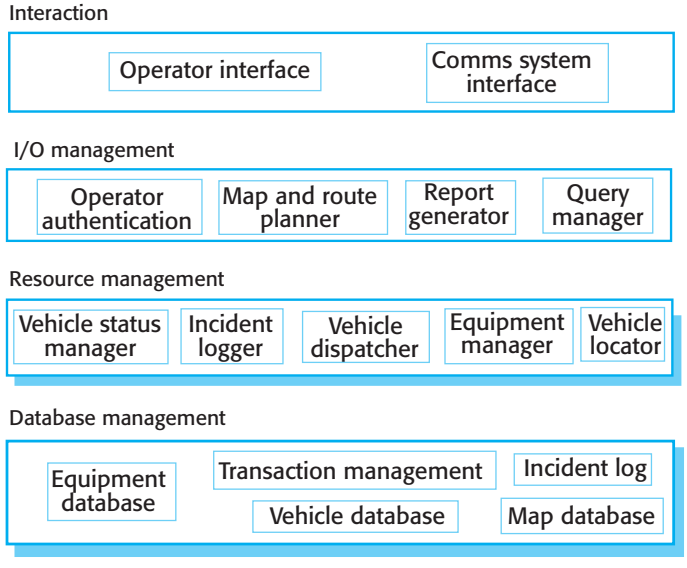


Figure 15.9 A product-line architecture of a vehicle dispatcher system

2. At the I/O management level (level 2), components handle operator authentication, generate reports of incidents and vehicles dispatched, support map output and route planning, and provide a mechanism for operators to query the system databases.
3. At the resource management level (level 3), components allow vehicles to be located and dispatched, update the status of vehicles and equipment, and log details of incidents.
4. At the database level, as well as the usual transaction management support, there are separate databases of vehicles, equipment, and maps.

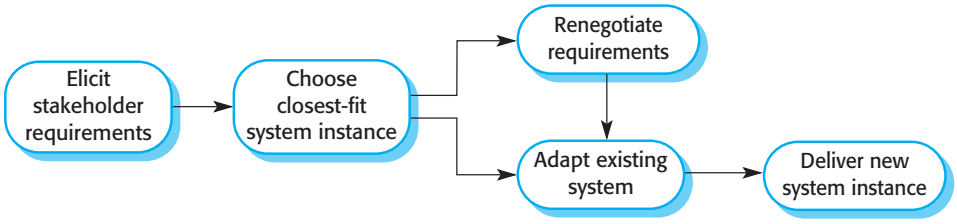


Figure 15.10 Product instance development

To create a new instance of this system, you may have to modify individual components. For example, the police have a large number of vehicles but a relatively small number of vehicle types. By contrast, the fire service has many types of specialized vehicles but relatively few vehicles. Therefore, when you are implementing a system for these different services, you may have to define a different vehicle database structure.

Various types of specialization of a software product line may be developed:

1. *Platform specialization* Versions of the application may be developed for different platforms. For example, versions of the application may exist for Windows, Mac OS, and Linux platforms. In this case, the functionality of the application is normally unchanged; only those components that interface with the hardware and operating system are modified.
2. *Environment specialization* Versions of the application may be created to handle different operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions, depending on the communications hardware used by each service. For example, police radios may have built-in encryption that has to be used. The product-line components are changed to reflect the functionality and characteristics of the equipment used.
3. *Functional specialization* Versions of the application may be created for specific customers who have different requirements. For example, a library automation system may be modified depending on whether it is used in a public library, a reference library, or a university library. In this case, components that implement functionality may be modified and new components added to the system.
4. *Process specialization* The system may be adapted to cope with specific business processes. For example, an ordering system may be adapted to cope with a centralized ordering process in one company and with a distributed process in another.

Figure 15.10 shows the process for extending a software product line to create a new application. The activities in this process are:

1. *Elicit stakeholder requirements* You may start with a normal requirements engineering process. However, because a system already exists, you can demonstrate the system and have stakeholders experiment with it, expressing their requirements as modifications to the functions provided.

2. *Select the existing system that is the closest fit to the requirements* When creating a new member of a product line, you may start with the nearest product instance. The requirements are analyzed, and the family member that is the closest fit is chosen for modification.

3. *Renegotiate requirements* As more details of required changes emerge and the project is planned, some requirements may be renegotiated with the customer to minimize the changes that will have to be made to the base application.

4. *Adapt existing system* New modules are developed for the existing system, and existing system modules are adapted to meet the new requirements.

5. *Deliver new product family member* The new instance of the product line is delivered to the customer. Some deployment-time configuration may be required to reflect the particular environments where the system will be used. At this stage, you should document its key features so that it may be used as a basis for other system developments in the future.

When you create a new member of a product line, you may have to find a compromise between reusing as much of the generic application as possible and satisfying detailed stakeholder requirements. The more detailed the system requirements, the less likely it is that the existing components will meet these requirements. However, if stakeholders are willing to be flexible and to limit the system modifications that are required, you can usually deliver the system more quickly and at a lower cost.

Software product lines are designed to be reconfigurable. This reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business processes. This configuration may occur at different stages in the development process:

1. *Design-time configuration* The organization that is developing the software modifies a common product-line core by developing, selecting, or adapting components to create a new system for a customer.

2. *Deployment-time configuration* A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in the configuration data used by the generic system.

When a system is configured at design time, the supplier starts with either a generic system or an existing product instance. By modifying and extending modules in this system, the supplier creates a specific system that delivers the required customer functionality. This usually involves changing and extending the source code of the system so that greater flexibility is possible than with deployment-time configuration.

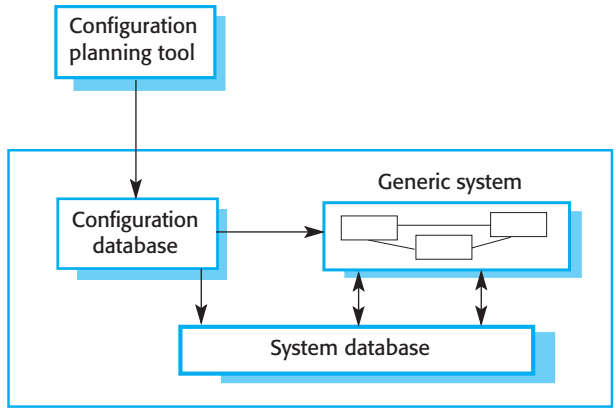


Figure 15.11 Deployment-time configuration

Design-time configuration is used when it is impossible to use the existing deployment-time configuration facilities in a system to develop a new system version. However, over time, when you have created several family members with comparable functionality, you may decide to refactor the core product line to include functionality that has been implemented in several application family members. You then make that new functionality configurable when the system is deployed.

Deployment-time configuration involves using a configuration tool to create a specific system configuration that is recorded in a configuration database or as a set of configuration files (Figure 15.11). The executing system, which may either run on a server or as a stand-alone system on a PC, consults this database when executing so that its functionality may be specialized to its execution context.

Several levels of deployment-time configuration may be provided in a system:

1. *Component selection*, where you select the modules in a system that provide the required functionality. For example, in a patient information system, you may select an image management component that allows you to link medical images (X-rays, CT scans, etc.) to the patient's medical record.
2. *Workflow and rule definition*, where you define workflows (how information is processed, stage by stage), and validation rules that should apply to information entered by users or generated by the system.
3. *Parameter definition*, where you specify the values of specific system parameters that reflect the instance of the application that you are creating. For example, you may specify the maximum length of fields for data input by a user or the characteristics of hardware attached to the system.

Deployment-time configuration can be very complex, and for large systems, it may take several months to configure and test a system for a customer. Large configurable systems may support the configuration process by providing software tools, such as planning tools, to support the configuration process. I discuss deployment-time configuration further in Section 15.4.1. This discussion covers the reuse of application systems that have to be configured to work in different operational environments.

15.4 Application system reuse

An application system product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Application systems are developed by a system vendor for a general market; they are not specially developed for an individual customer. These system products are sometimes known as COTS (Commercial Off-the Shelf System) products. However, the term "COTS" is mostly used in military systems, and I prefer to call these system products *application systems*.

Virtually all desktop software for business and many server-based systems are application systems. This software is designed for general use, so it includes many features and functions. It therefore has the potential to be reused in different environments and as part of different applications. Torchiano and Morisio (Torchiano and Morisio 2004) also discovered that open-source products were often used without change and without looking at the source code.

Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs. For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patients. Other configuration features may allow the system to accept plug-ins that extend functionality or check user inputs to ensure that they are valid.

This approach to software reuse has been very widely adopted by large companies since the late 1990s, as it offers significant benefits over customized software development:

1. As with other types of reuse, more rapid deployment of a reliable system may be possible.
2. It is possible to see what functionality is provided by the applications, and so it is easier to judge whether or not they are likely to be suitable. Other companies may already use the applications, so experience of the systems is available.
3. Some development risks are avoided by using existing software. However, this approach has its own risks, as I discuss below.
4. Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
5. As operating platforms evolve, technology updates may be simplified as these are the responsibility of the application system vendor rather than the customer.

Of course, this approach to software engineering has its own problems:

1. Requirements usually have to be adapted to reflect the functionality and mode of operation of the off-the-shelf application system. This can lead to disruptive changes to existing business processes.

Configurable application systems	Application system integration
Single product that provides the functionality required by a customer	Several different application systems are integrated to provide customized functionality
Based on a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

Figure 15.12 Individual and integrated application systems

2. The application system may be based on assumptions that are practically impossible to change. The customer must therefore adapt its business to reflect these assumptions.
3. Choosing the right application system for an enterprise can be a difficult process, especially as many of these systems are not well documented. Making the wrong choice means that it may be impossible to make the new system work as required.
4. There may be a lack of local expertise to support systems development. Consequently, the customer has to rely on the vendor and external consultants for development advice. This advice may be geared to selling products and services, with insufficient time taken to understand the real needs of the customer.
5. The system vendor controls system support and evolution. It may go out of business, be taken over, or make changes that cause difficulties for customers.

Application systems may be used as individual systems or in combination, where two or more systems are integrated. Individual systems consist of a generic application from a single vendor that is configured to customer requirements. Integrated systems involve integrating the functionality of individual systems, often from different vendors, to create a new application system. Figure 15.12 summarizes the differences between these different approaches. I discuss application system integration in Section 15.4.2.

15.4.1 Configurable application systems

Configurable application systems are generic application systems that may be designed to support a particular business type, business activity, or, sometimes, a complete business enterprise. For example, a system produced for dentists may handle appointments, reminders, dental records, patient recall, and billing. At a larger scale, an Enterprise Resource Planning (ERP) system may support the manufacturing, ordering, and customer relationship management processes in a large company.

Domain-specific application systems, such as systems to support a business function (e.g., document management), provide functionality that is likely to be required by a range of potential users. However, they also incorporate built-in assumptions about how

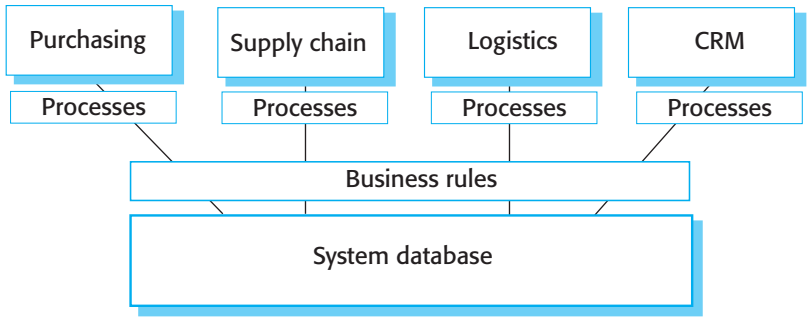


Figure 15.13 The architecture of an ERP system

users work, and these assumptions may cause problems in specific situations. For example, a system to support student registration in a university may assume that students will be registered for one degree at one university. However, if universities collaborate to offer joint degrees, then it may be practically impossible to represent this detail in the system.

Enterprise Resource Planning (ERP) systems, such as those produced by SAP and Oracle, are large-scale, integrated systems designed to support business practices such as ordering and invoicing, inventory management, and manufacturing scheduling (Monk and Wagner 2013). The configuration process for these systems involves gathering detailed information about the customer's business and business processes, and embedding this information in a configuration database. This often requires detailed knowledge of configuration notations and tools and is usually carried out by consultants working alongside system customers.

A generic ERP system includes a number of modules that may be composed in different ways to create a system for a customer. The configuration process involves choosing which modules are to be included, configuring these individual modules, defining business processes and business rules, and defining the structure and organization of the system database. A model of the overall architecture of an ERP system that supports a range of business functions is shown in Figure 15.13.

The key features of this architecture are as follows:

1. A number of modules to support different business functions. These are large grain modules that may support entire departments or divisions of the business. In the example shown in Figure 15.13, the modules that have been selected for inclusion in the system are a module to support purchasing; a module to support supply chain management; a logistics module to support the delivery of goods; and a customer relationship management (CRM) module to maintain customer information.
2. A defined set of business process models, associated with each module, which relate to activities in that module. For example, the ordering process model may define how orders are created and approved. This will specify the roles and activities involved in placing an order.
3. A common database that maintains information about all related business functions. Thus, it should not be necessary to replicate information, such as customer details, in different parts of the business.

4. A set of business rules that apply to all data in the database. Therefore, when data is input from one function, these rules should ensure that it is consistent with the data required by other functions. For example, a business rule may require that all expense claims have to be approved by someone more senior than the person making the claim.

ERP systems are used in almost all large companies to support some or all of their functions. They are, therefore, a very widely used form of software reuse. The obvious limitation of this approach to reuse is that the functionality of the customer's application is restricted to the functionality of the ERP system's built-in modules. If a company needs additional functionality, it may have to develop a separate add-on system to provide this functionality.

Furthermore, the buyer company's processes and operations have to be defined in the ERP system's configuration language. This language embeds the understanding of business processes as seen by the system vendor, and there may be a mismatch between these assumptions and the concepts and processes used in the customer's business. A serious mismatch between the customer's business model and the system model used by the ERP system makes it highly probable that the ERP system will not meet the customer's real needs (Scott 1999).

For example, in an ERP system that was sold to a university, a fundamental system concept was the notion of a customer. In this system, a customer was an external agent that bought goods and services from a supplier. This concept caused great difficulties when configuring the system. Universities do not really have customers. Rather, they have customer-type relationships with a range of people and organizations such as students, research funding agencies, and educational charities. None of these relationships is compatible with a customer relationship where a person or business buys products or services from another. In this particular case, it took several months to resolve this mismatch, and the final solution only partially met the university's requirements.

ERP systems usually require extensive configuration to adapt them to the requirements of each organization where they are installed. This configuration may involve:

1. Selecting the required functionality from the system, for example, by deciding what modules should be included.
2. Establishing a data model that defines how the organization's data will be structured in the system database.
3. Defining business rules that apply to that data.
4. Defining the expected interactions with external systems.
5. Designing the input forms and the output reports generated by the system.
6. Designing new business processes that conform to the underlying process model supported by the system.
7. Setting parameters that define how the system is deployed on its underlying platform.

Once the configuration settings are completed, the new system is then ready for testing. Testing is a major problem when systems are configured rather than programmed using a conventional language. There are two reasons for this:

1. Test automation may be difficult or impossible. There may be no easy access to an API that can be used by testing frameworks such as JUnit, so the system has to be tested manually by testers inputting test data to the system. Furthermore, systems are often specified informally, so defining test cases may be difficult without a lot of help from end-users.
2. Systems errors are often subtle and specific to business processes. The application system or ERP system is a reliable platform, so technical system failures are rare. The problems that occur are often due to misunderstandings between those configuring the system and user stakeholders. System testers without detailed knowledge of the end-user processes cannot detect these errors.

15.4.2 Integrated application systems

Integrated application systems include two or more application systems or, sometimes, legacy systems. You may use this approach when no single application system meets all of your needs or when you wish to integrate a new application system with systems that you are already using. The component systems may interact through their APIs or service interfaces if these are defined. Alternatively, they may be composed by connecting the output of one system to the input of another or by updating the databases used by the applications.

To develop integrated application systems, you have to make a number of design choices:

1. *Which individual application systems offer the most appropriate functionality?* Typically, several system products will be available, which can be combined in different ways. If you don't already have experience with a particular application system, it can be difficult to decide which product is the most suitable.
2. *How will data be exchanged?* Different systems normally use unique data structures and formats. You have to write adaptors that convert from one representation to another. These adaptors are runtime systems that operate alongside the constituent application systems.
3. *What features of a product will actually be used?* Individual application systems may include more functionality than you need, and functionality may be duplicated across different products. You have to decide which features in what product are most appropriate for your requirements. If possible, you should also deny access to unused functionality because this can interfere with normal system operation.

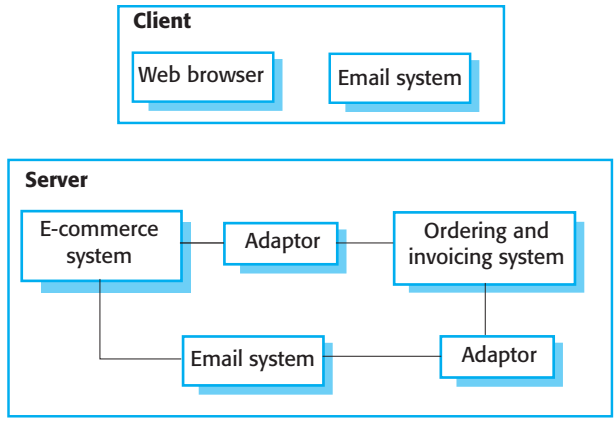


Figure 15.14 An integrated procurement system

Consider the following scenario as an illustration of application system integration. A large organization intends to develop a procurement system that allows staff to place orders from their desk. By introducing this system across the organization, the company estimates that it can save $5 million per year. By centralizing buying, the new procurement system can ensure that orders are always made from suppliers who offer the best prices and should reduce the administration associated with orders. As with manual systems, the system involves choosing the goods available from a supplier, creating an order, having the order approved, sending the order to a supplier, receiving the goods, and confirming that payment should be made.

The company has a legacy ordering system that is used by a central procurement office. This order processing software is integrated with an existing invoicing and delivery system. To create the new ordering system, the legacy system is integrated with a web-based e-commerce platform and an email system that handles communications with users. The structure of the final procurement system is shown in Figure 15.14.

This procurement system should be a client–server system with standard web browsing and email systems used on the client. On the server, the e-commerce platform has to integrate with the existing ordering system through an adaptor. The e-commerce system has its own format for orders, confirmations of delivery, and so forth, and these have to be converted into the format used by the ordering system. The e-commerce system uses the email system to send notifications to users, but the ordering system was never designed for this purpose. Therefore, another adaptor has to be written to convert the notifications from the ordering system into email messages.

Months, sometimes years, of implementation effort can be saved, and the time to develop and deploy a system can be drastically reduced by integrating existing application systems. The procurement system described above was implemented and deployed in a very large company in nine months. It had originally been estimated that it would take three years to develop a procurement system in Java that could be integrated with the legacy ordering system.

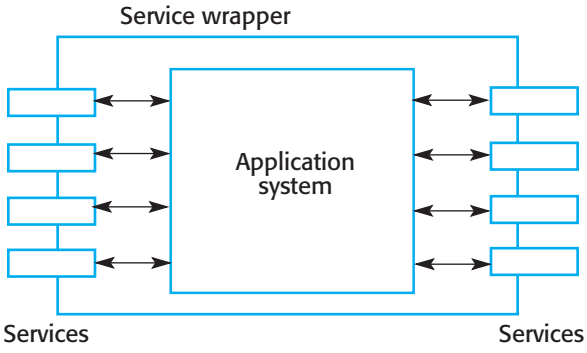


Figure 15.15
Application wrapping

Application system integration can be simplified if a service-oriented approach is used. Essentially, a service-oriented approach means allowing access to the application system's functionality through a standard service interface, with a service for each discrete unit of functionality. Some applications may offer a service interface, but sometimes this service interface has to be implemented by the system integrator. Essentially, you have to program a wrapper that hides the application and provides externally visible services (Figure 15.15). This approach is particularly valuable for legacy systems that have to be integrated with newer application systems.

In principle, integrating application systems is the same as integrating any other component. You have to understand the system interfaces and use them exclusively to communicate with the software; you have to trade off specific requirements against rapid development and reuse; and you have to design a system architecture that allows the application systems to operate together.

However, the fact that these products are usually large systems in their own right, and are often sold as separate standalone systems, introduces additional problems. Boehm and Abts (Boehm and Abts 1999) highlight four important system integration problems:

1. *Lack of control over functionality and performance* Although the published interface of a product may appear to offer the required facilities, the system may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation. Fixing these problems may be a priority for the system integrator but may not be of real concern for the product vendor. Users may simply have to find workarounds to problems if they wish to reuse the application system.

2. *Problems with system interoperability* It is sometimes difficult to get individual application systems to work together because each system embeds its own assumptions about how it will be used. Garlan et al. (Garlan, Allen, and Ockerbloom 1995), reporting on their experience integrating four application systems, found that three of these products were event-based but that each used a different model of events. Each system assumed that it had exclusive access to the event queue. As a consequence, integration was very difficult. The project

required five times as much effort as originally predicted. The schedule was extended to two years rather than the predicted six months.

In a retrospective analysis of their work 10 years later, Garlan et al. (Garlan, Allen, and Ockerbloom 2009) concluded that the integration problems that they discovered had not been solved. Torchiano and Morisio (Torchiano and Morisio 2004) found that lack of compliance with standards in many application systems meant that integration was more difficult than anticipated.

3. *No control over system evolution* Vendors of application systems make their own decisions on system changes, in response to market pressures. For PC products in particular, new versions are often produced frequently and may not be compatible with all previous versions. New versions may have additional unwanted functionality, and previous versions may become unavailable and unsupported.

4. *Support from system vendors* The level of support available from system vendors varies widely. Vendor support is particularly important when problems arise as developers do not have access to the source code and detailed documentation of the system. While vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a system vendor may decide to discontinue a product because of limited demand, or they may be taken over by another company that does not wish to support the products that have been acquired.

Boehm and Abts reckon that, in many cases, the cost of system maintenance and evolution may be greater for integrated application systems. The above difficulties are life-cycle problems; they don't just affect the initial development of the system. The further removed the people involved in the system maintenance become from the original system developers, the more likely it is that difficulties will arise with the integrated system.

KEY POINTS

- There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- The advantages of software reuse are lower costs, faster software development, and lower risks. System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects. They usually incorporate good design practice through design patterns.

- Software product lines are related applications that are developed from one or more base applications. A generic system is adapted and specialized to meet specific requirements for functionality, target platform, or operational configuration.
- Application system reuse is concerned with the reuse of large-scale, off-the-shelf systems. These provide a lot of functionality, and their reuse can radically reduce costs and development time. Systems may be developed by configuring a single, generic application system or by integrating two or more application systems.
- Potential problems with application system reuse include lack of control over functionality, performance, and system evolution; the need for support from external vendors; and difficulties in ensuring that systems can interoperate.

FURTHER READING

"Overlooked Aspects of COTS-Based Development." An interesting article that discusses a survey of developers using a COTS-based approach, and the problems that they encountered. (M. Torchiano and M. Morisio, *IEEE Software*, 21 (2), March–April 2004) http://dx.doi.org/10.1109/MS.2004.1270770

CRUISE—Component Reuse in Software Engineering. This e-book covers a wide range of reuse topics, including case studies, component-based reuse, and reuse processes. However, its coverage of application system reuse is limited. (L. Nascimento et al., 2007) http://www.academia.edu/179616/C.R.U.I.S.E_-_Component_Reuse_in_Software_Engineering

"Construction by Configuration: A New Challenge for Software Engineering." In this invited paper, I discuss the problems and difficulties of constructing a new application by configuring existing systems. (I. Sommerville, *Proc. 19th Australian Software Engineering Conference*, 2008) http://dx.doi.org/10.1109/ASWEC.2008.75

"Architectural Mismatch: Why Reuse Is Still So Hard." This article looks back on an earlier paper that discussed the problems of reusing and integrating a number of application systems. The authors concluded that, although some progress has been made, there were still problems in conflicting assumptions made by the designers of the individual systems. (D. Garlan et al., *IEEE Software*, 26 (4), July–August 2009) http://dx.doi.org//10.1109/MS.2009.86

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

EXERCISES

15.1. What major technical and nontechnical factors hinder software reuse? Do you personally reuse much software and, if not, why not?

15.2. List the benefits of software reuse and explain why the expected lifetime of the software should be considered when planning reuse.

15.3. How does the base application's design in the product line simplify reuse and reconfiguration?

15.4. Explain what is meant by "inversion of control" in application frameworks. Explain why this approach could cause problems if you integrated two separate systems that were originally created using the same application framework.

15.5. Using the example of the weather station system described in Chapters 1 and 7, suggest a product-line architecture for a family of applications that are concerned with remote monitoring and data collection. You should present your architecture as a layered model, showing the components that might be included at each level.

15.6. Most desktop software, such as word processing software, can be configured in a number of different ways. Examine software that you regularly use and list the configuration options for that software. Suggest difficulties that users might have in configuring the software. Microsoft Office (or one of its open-source alternatives) is a good example to use for this exercise.

15.7. Why have many large companies chosen ERP systems as the basis for their organizational information system? What problems may arise when deploying a large-scale ERP system in an organization?

15.8. What are the significant benefits offered by the application system reuse approach when compared with the custom software development approach?

15.9. Explain why adaptors are usually needed when systems are constructed by integrating application systems. Suggest three practical problems that might arise in writing adaptor software to link two application systems.

15.10. The reuse of software raises a number of copyright and intellectual property issues. If a customer pays a software contractor to develop a system, who has the right to reuse the developed code? Does the software contractor have the right to use that code as a basis for a generic component? What payment mechanisms might be used to reimburse providers of reusable components? Discuss these issues and other ethical issues associated with the reuse of software.

REFERENCES

Baumer, D., G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zullighoven. 1997. "Framework Development for Large Systems." *Comm. ACM* 40 (10): 52–59. doi:10.1145/262793.262804.

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Fayad, M.E., and D.C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40 (10): 32–38. doi:10.1145/262793.262798.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., R. Allen, and J. Ockerbloom. 1995. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software* 12 (6): 17–26. doi:10.1109/52.469757.

———. 2009. "Architectural Mismatch: Why Reuse Is Still so Hard." *IEEE Software* 26 (4): 66–69. doi:10.1109/MS.2009.86.

Holdener, A.T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

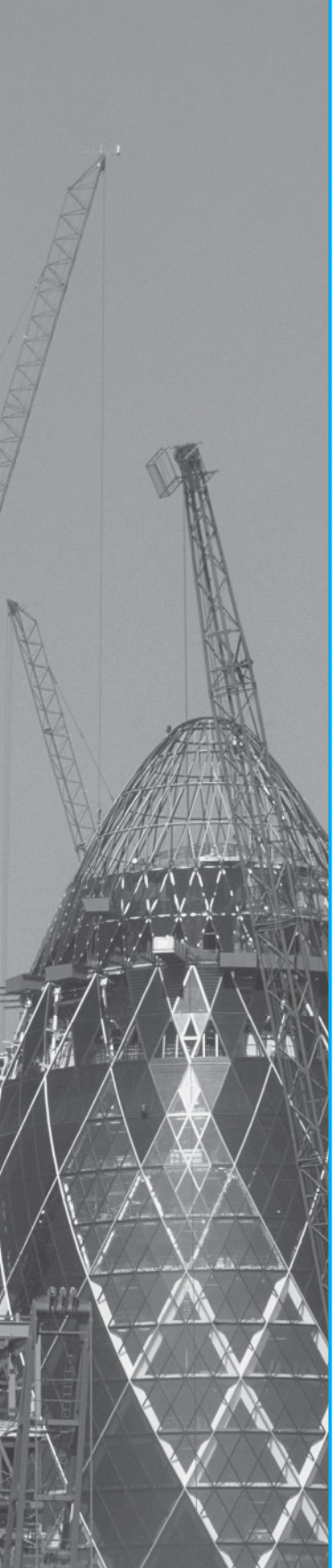
Monk, E., and B. Wagner. 2013. *Concepts in Enterprise Resource Planning, 4th ed.* Independence, KY: CENGAGE Learning.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." *ACM Queue* 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

Scott, J. E. 1999. "The FoxMeyer Drug's Bankruptcy: Was It a Failure of ERP." In *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI. http://www.uta.edu/faculty/weltman/OPMA5364TW/FoxMeyer.pdf

Torchiano, M., and M. Morisio. 2004. "Overlooked Aspects of COTS-Based Development." *IEEE Software* 21 (2): 88–93. doi:10.1109/MS.2004.1270770.



16

Component-based software engineering

Objectives

The objective of this chapter is to describe an approach to software reuse based on the composition of standardized, reusable components. When you have read this chapter, you will:

- understand what is meant by a software component that may be included in a program as an executable element;
- understand the key elements of software component models and the support provided by middleware for these models;
- be aware of the key activities in the component-based software engineering (CBSE) process for reuse and the CBSE process with reuse;
- understand three different types of component composition and some of the problems that have to be resolved when components are composed to create new components or systems.

Contents

16.1 Components and component models
16.2 CBSE processes
16.3 Component composition

Component-based software engineering (CBSE) emerged in the late 1990s as an approach to software systems development based on reusing software components. Its creation was motivated by frustration that object-oriented development had not led to extensive reuse, as had been originally suggested. Single-object classes were too detailed and specific and often had to be bound with an application at compile-time. You had to have detailed knowledge of the classes to use them, which usually meant that you had to have the component source code. Selling or distributing objects as individual reusable components was therefore practically impossible.

Components are higher-level abstractions than objects and are defined by their interfaces. They are usually larger than individual objects, and all implementation details are hidden from other components. Component-based software engineering is the process of defining, implementing, and integrating or composing these loosely coupled, independent components into systems.

CBSE has become as an important software development approach for large-scale enterprise systems, with demanding performance and security requirements. Customers are demanding secure and dependable software that is delivered and deployed more quickly. The only way that these demands can be met is to build software by reusing existing components.

The essentials of component-based software engineering are:

1. Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation. This means that one implementation of a component can be replaced by another, without the need to change other parts of the system.
2. Component standards that define interfaces and so facilitate the integration of components. These standards are embodied in a component model. They define, at the very minimum, how component interfaces should be specified and how components communicate. Some models go much further and define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3. Middleware that provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. In addition, middleware for component support may provide support for resource allocation, transaction management, security, and concurrency.
4. A development process that is geared to component-based software engineering. You need a development process that allows requirements to evolve, depending on the functionality of available components.

Component-based development embodies good software engineering practice. It often makes sense to design a system using components, even if you have to develop



Problems with CBSE

CBSE is now a mainstream approach to software engineering and is widely used when creating new systems. However, when used as an approach to reuse, problems include component trustworthiness, component certification, requirements compromises, and prediction of the properties of components, especially when they are integrated with other components.

http://software-engineering-book.com/web/cbse-problems/

rather than reuse these components. Underlying CBSE are sound design principles that support the construction of understandable and maintainable software:

1. Components are independent, so they do not interfere with each other's operation. Implementation details are hidden. The component's implementation can be changed without affecting the rest of the system.
2. Components communicate through well-defined interfaces. If these interfaces are maintained, one component can be replaced by another component providing additional or enhanced functionality.
3. Component infrastructures offer a range of standard services that can be used in application systems. This reduces the amount of new code that has to be developed.

The initial motivation for CBSE was the need to support both reuse and distributed software engineering. A component was seen as an element of a software system that could be accessed, using a remote procedure call mechanism, by other components running on separate computers. Each system that reused a component had to incorporate its own copy of that component. This idea of a component extended the notion of distributed objects, as defined in distributed systems models such as the CORBA specification (Pope 1997). Several different protocols and technology-specific "standards" were introduced to support this view of a component, including Sun's Enterprise Java Beans (EJB), Microsoft's COM and .NET, and CORBA's CCM (Lau and Wang 2007).

Unfortunately, the companies involved in proposing standards could not agree on a single standard for components, thereby limiting the impact of this approach to software reuse. It is impossible for components developed using different approaches to work together. Components that are developed for different platforms, such as .NET or J2EE, cannot interoperate. Furthermore, the standards and protocols proposed were complex and difficult to understand. This was also a barrier to their adoption.

In response to these problems, the notion of a component as a service was developed, and standards were proposed to support service-oriented software engineering. The most significant difference between a component as a service and the original notion of a component is that services are stand-alone entities that are external to a program using them. When you build a service-oriented system, you reference the external service rather than including a copy of that service in your system.

Service-oriented software engineering is a type of component-based software engineering. It uses a simpler notion of a component than that originally proposed in CBSE,

where components were executable routines that were included in larger systems. Each system that used a component embedded its own version of that component. Service-oriented approaches are gradually replacing CBSE with embedded components as an approach to systems development. In this chapter, I discuss the use of CBSE with embedded components; service-oriented software engineering is covered in Chapter 18.

16.1 Components and component models

The software reuse community generally agrees that a component is an independent software unit that can be composed with other components to create a software system. Beyond that, however, people have proposed varying definitions of a software component. Councill and Heineman (Councill and Heineman 2001) define a component as:

> *A software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard.*[†]

This definition is standards-based so that a software unit that conforms to these standards is a component. Szyperski (Szyperski 2002), however, does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

> *A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*[‡]

Both of these definitions were developed around the idea of a component as an element that is embedded in a system, rather than a service that is referenced by the system. However, they are equally applicable to service components.

Szyperski also states that a component has no externally observable state; that is, copies of components are indistinguishable. However, some component models, such as the Enterprise Java Beans model, allow stateful components, so these do not correspond with Szyperski's definition. While stateless components are certainly simpler to use, in some systems stateful components are more convenient and reduce system complexity.

What the above definitions have in common is that they agree that components are independent and that they are the fundamental unit of composition in a system. I think that, if we combine these proposals, we get a more rounded description of a reusable component. Figure 16.1 shows what I consider to be the essential characteristics of a component as used in CBSE.

[†]Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In Component-Based Software Engineering, edited by G T Heineman and W T Councill, 5–20. Boston: Addison Wesley.

[‡]Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming, 2nd Ed. Harlow, UK: Addison Wesley.

Component characteristic	Description
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of that component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a "requires" interface specification.
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

Figure 16.1 Component characteristics

A useful way of thinking about a component is as a provider of one or more services, even if the component is embedded rather than implemented as a service. When a system needs something to be done, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component. For example, a component in a system used in a public library might provide a search service that allows users to search the library catalog. A component that converts from one graphical format to another (e.g., TIFF to JPEG) provides a data conversion service and so on.

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:

1. The component is an independent executable entity that is defined by its interfaces. You don't need any knowledge of its source code to use it. It can either be referenced as an external service or included directly in a program.
2. The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterized operations, and its internal state is never exposed.

In principle, all components have two related interfaces, as shown in Figure 16.2. These interfaces reflect the services that the component provides and the services that the component requires to operate correctly:

1. The "provides" interface defines the services provided by the component. This interface is the component API. It defines the methods that can be called by a user

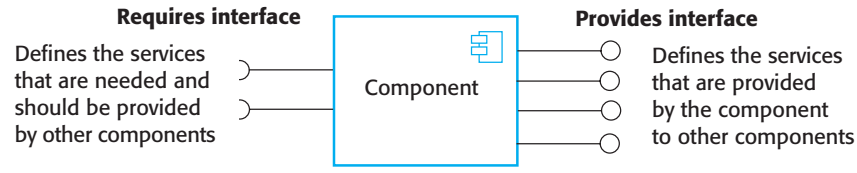


Figure 16.2 Component interfaces

of the component. In a UML component diagram, the "provides" interface for a component is indicated by a circle at the end of a line from the component icon.

2. The "requires" interface specifies the services that other components in the system must provide if a component is to operate correctly. If these services are not available, then the component will not work. This does not compromise the independence or deployability of a component because the "requires" interface does not define how these services should be provided. In the UML, the symbol for a "requires" interface is a semicircle at the end of a line from the component icon. Notice that "provides" and "requires" interface icons can fit together like a ball and socket.

To illustrate these interfaces, Figure 16.3 shows a model of a component that has been designed to collect and collate information from an array of sensors. It runs autonomously to collect data over a period of time and, on request, provides collated data to a calling component. The "provides" interface includes methods to add, remove, start, stop, and test sensors. The report method returns the sensor data that has been collected, and the listAll method provides information about the attached sensors. Although I have not shown them here, these methods have associated parameters specifying the sensor identifiers, locations, and so on.

The "requires" interface is used to connect the component to the sensors. It assumes that sensors have a data interface, accessed through sensorData, and a management interface, accessed through sensorManagement. This interface has been designed to connect to different types of sensors so that it does not include specific sensor operations such as Test and provideReading. Instead, the commands used by a specific type of sensor are embedded in a string, which is a parameter to the operations in the "requires" interface. Adaptor components parse this parameter string and translate the embedded commands into the specific control interface of each type of sensor. I discuss the use of adaptors later in this chapter, where I show how the data collector component may be connected to a sensor (Figure 16.12).

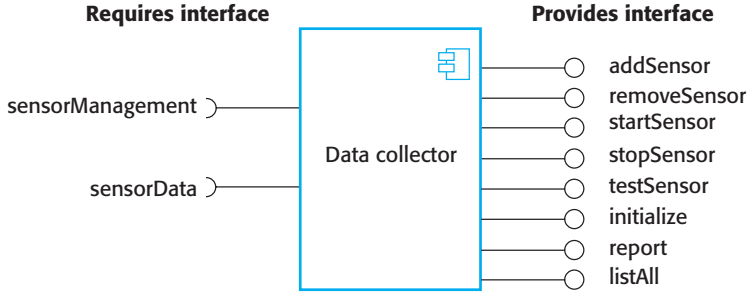


Figure 16.3 A model of a data collector component



Components and objects

Components are often implemented in object-oriented languages, and, in some cases, accessing the "provides" interface of a component is done through method calls. However, components and object classes are not the same thing. Unlike object classes, components are independently deployable, do not define types, are language-independent, and are based on a standard component model.

http://software-engineering-book.com/web/components-and-objects/

Components are accessed using remote procedure calls (RPCs). Each component has a unique identifier and, using this name, may be called from another computer. The called component uses the same mechanism to access the "required" components that are defined in its interface.

An important difference between a component as an external service and a component as a program element accessed using a remote procedure call is that services are completely independent entities. They do not have an explicit "requires" interface. Of course, they do require other components to support their operation, but these are provided internally. Other programs can use services without the need to implement any additional support required by the service.

16.1.1 Component models

A component model is a definition of standards for component implementation, documentation, and deployment. These standards are for component developers to ensure that components can interoperate. They are also for providers of component execution infrastructures who provide middleware to support component operation. For service components, the most important component model is the Web Service models, and for embedded components, widely used models include the Enterprise Java Beans (EJB) model and Microsoft's .NET model (Lau and Wang 2007).

The basic elements of an ideal component model are discussed by Weinreich and Sametinger (Weinreich and Sametinger 2001). I summarize these model elements in Figure 16.4. This diagram shows that the elements of a component model define the component interfaces, the information that you need to use the component in a program, and how a component should be deployed:

1. *Interfaces* Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition. The model should also specify the language used to define the component interfaces.

   For web services, interface specification uses XML-based languages as discussed in Chapter 18; EJB is Java-specific, so Java is used as the interface definition language; in .NET, interfaces are defined using Microsoft's Common

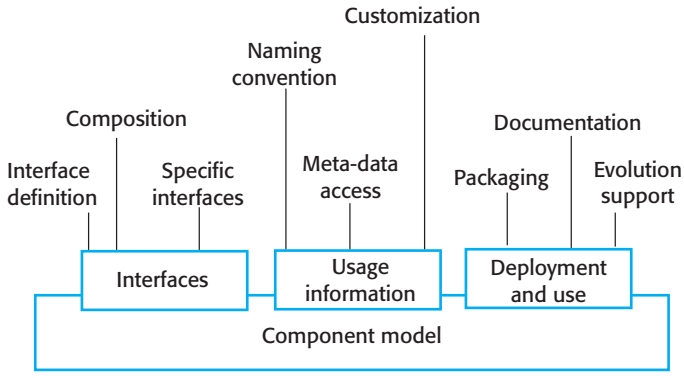


Figure 16.4 Basic elements of a component model

Intermediate Language (CIL). Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure, which provides standardized services such as security and transaction management.

2. *Usage* In order for components to be distributed and accessed remotely via RPCs, they need to have a unique name or handle associated with them. This has to be globally unique. For example, in EJB, a hierarchical name is generated with the root based on an Internet domain name. Services have a unique URI (Uniform Resource Identifier).

   Component meta-data is data about the component itself, such as information about its interfaces and attributes. The meta-data is important because it allows users of the component to find out what services are provided and required. Component model implementations normally include specific ways (such as the use of a reflection interface in Java) to access this component meta-data.

   Components are generic entities, and, when deployed, they have to be configured to fit into an application system. For example, you could configure the Data collector component (Figure 16.3) by defining the maximum number of sensors in a sensor array. The component model may therefore specify how the binary components can be customized for a particular deployment environment.

3. *Deployment* The component model includes a specification of how components should be packaged for deployment as independent, executable routines. Because components are independent entities, they have to be packaged with all supporting software that is not provided by the component infrastructure, or is not defined in a "requires" interface. Deployment information includes information about the contents of a package and its binary organization.

   Inevitably, as new requirements emerge, components will have to be changed or replaced. The component model may therefore include rules governing when and how component replacement is allowed. Finally, the component model may define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

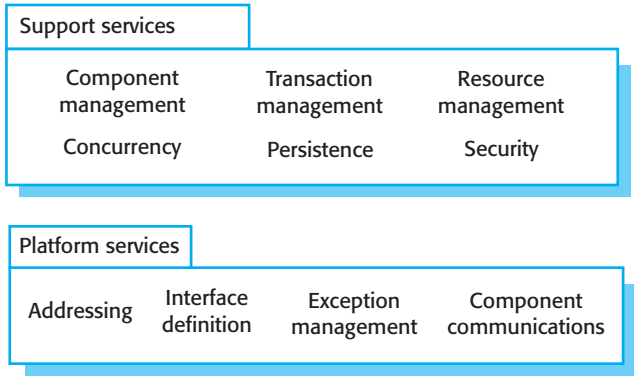


Figure 16.5 Middleware services defined in a component model

For components that are executable routines rather than external services, the component model defines the services to be provided by the middleware that supports the executing components. Weinreich and Sametinger use the analogy of an operating system to explain component models. An operating system provides a set of generic services that can be used by applications. A component model implementation provides comparable shared services for components. Figure 16.5 shows some of the services that may be provided by an implementation of a component model.

The services provided by a component model implementation fall into two categories:

1. *Platform services*, which enable components to communicate and interoperate in a distributed environment. These are the fundamental services that must be available in all component-based systems.
2. *Support services*, which are common services that many different components are likely to require. For example, many components require authentication to ensure that the user of component services is authorized. It makes sense to provide a standard set of middleware services for use by all components. This reduces the costs of component development, and potential component incompatibilities can be avoided.

Middleware implements the common component services and provides interfaces to them. To make use of the services provided by a component model infrastructure, you can think of the components as being deployed in a "container." A container is an implementation of the support services plus a definition of the interfaces that a component must provide to integrate it with the container. Conceptually, when you add a component to the container, the component can access the support services and the container can access the component interfaces. When in use, the component interfaces themselves are not accessed directly by other components. They are accessed through a container interface that invokes code to access the interface of the embedded component.

Containers are large and complex and, when you deploy a component in a container, you get access to all middleware services. However, simple components may

not need all of the facilities offered by the supporting middleware. The approach taken in web services to common service provision is therefore rather different. For web services, standards have been defined for common services such as transaction management and security, and these standards have been implemented as program libraries. If you are implementing a service component, you only use the common services that you need.

The services associated with a component model have much in common with the facilities provided by object-oriented frameworks, which I discussed in Chapter 15. Although the services provided may not be as comprehensive, framework services are often more efficient than container-based services. As a consequence, some people think that it is best to use frameworks such as SPRING (Wheeler and White 2013) for Java development rather than the fully-featured component model in EJB.

16.2 CBSE processes

CBSE processes are software processes that support component-based software engineering. They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components. Figure 16.6 (Kotonya 2003) presents an overview of the processes in CBSE. At the highest level, there are two types of CBSE processes:

1. *Development for reuse* This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.
2. *Development with reuse* This process is the process of developing new applications using existing components and services.

These processes have different objectives and therefore include different activities. In the development for reuse process, the objective is to produce one or more reusable components. You know the components that you will be working with, and you have access to their source code to generalize them. In development with reuse, you don't know what components are available, so you need to discover these components and design your system to make the most effective use of them. You may not have access to the component source code.

You can see from Figure 16.6 that the basic processes of CBSE with and for reuse have supporting processes that are concerned with component acquisition, component management, and component certification:

1. *Component acquisition* is the process of acquiring components for reuse or development into a reusable component. It may involve accessing locally developed components or services or finding these components from an external source.

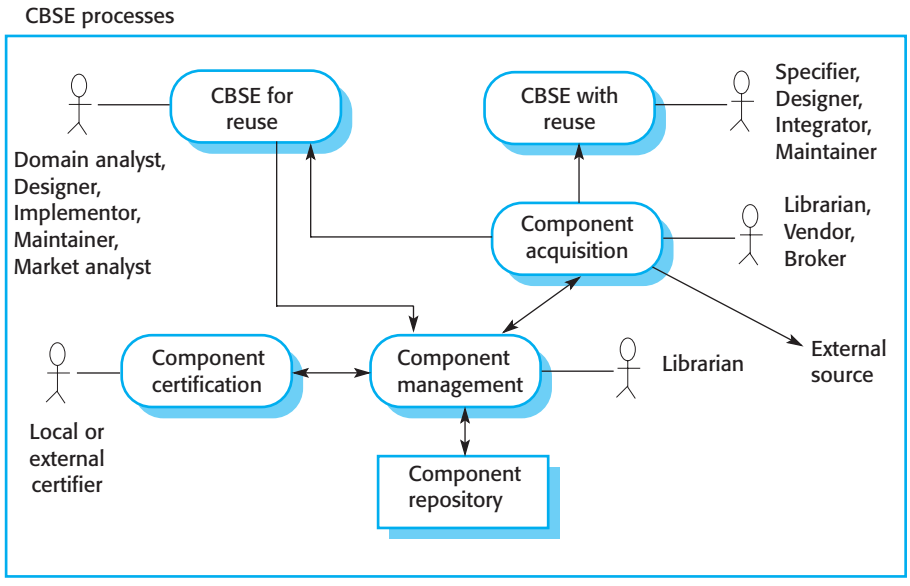


Figure 16.6 CBSE processes

2. *Component management* is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored, and made available for reuse.
3. *Component certification* is the process of checking a component and certifying that it meets its specification.

Components maintained by an organization may be stored in a component repository that includes both the components and information about their use.

16.2.1 CBSE for reuse

CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system. The vision of early supporters of CBSE (Szyperski 2002) was that a thriving component marketplace would develop. There would be specialist component providers and component vendors who would organize the sale of components from different developers. Software developers would buy components to include in a system or pay for services as they were used. However, this vision has not been realized. There are relatively few component suppliers, and buying off-the-shelf components is uncommon.

Consequently, CBSE for reuse is mostly used within organizations that have made a commitment to reuse-driven software engineering. These companies have a base of internally developed components that can be reused. However, these internally developed components may not be reusable without change. They often include application-specific features and interfaces that are unlikely to be required in other programs where the component is reused.

To make components reusable, you have to adapt and extend the application-specific components to create more generic and therefore more reusable versions. Obviously, this adaptation has an associated cost. You have to decide first, whether a component is likely to be reused and second, whether the cost savings from future reuse justify the costs of making the component reusable.

To answer the first of these questions, you have to decide whether or not the component implements one or more stable domain abstractions. Stable domain abstractions are fundamental elements of the application domain that change slowly. For example, in a banking system, domain abstractions might include accounts, account holders, and statements. In a hospital management system, domain abstractions might include patients, treatments, and nurses. These domain abstractions are sometimes called business objects. If the component is an implementation of a commonly used domain abstraction or group of related business objects, it can probably be reused.

To answer the question about cost-effectiveness, you have to assess the costs of changes that are required to make the component reusable. These costs are the costs of component documentation and component validation, and of making the component more generic. Changes that you may make to a component to make it more reusable include:

- removing application-specific methods;
- changing names to make them more general;
- adding methods to provide more complete functional coverage;
- making exception handling consistent for all methods;
- adding a "configuration" interface to allow the component to be adapted to different situations of use;
- integrating required components to increase independence.

The problem of exception handling is a difficult one. In principle, components should not handle exceptions themselves because each application will have its own requirements for exception management. Rather, the component should define what exceptions can arise and should publish these exceptions as part of the interface. For example, a simple component implementing a stack data structure should detect and publish stack overflow and stack underflow exceptions. In practice, however, there are two problems with this process:

1. Publishing all exceptions leads to bloated interfaces that are harder to understand. This may put off potential users of the component.
2. The operation of the component may depend on local exception handling, and changing this may have serious implications for the functionality of the component.

You therefore have to take a pragmatic approach to component exception handling. Common technical exceptions, where recovery is important for the functioning of the component, should be handled locally. These exceptions and how they are handled

should be documented with the component. Other exceptions that are related to the business function of the component should be passed to the calling component for handling.

Mili et al. (Mili et al. 2002) discuss ways of estimating the costs of making a component reusable and the returns from that investment. The benefits of reusing rather than redeveloping a component are not simply productivity gains. There are also quality gains, because a reused component should be more dependable, and time-to-market gains. These are the increased returns that accrue from deploying the software more quickly.

Mili et al. present various formulas for estimating these gains, as does the COCOMO model, discussed in Chapter 23. However, the parameters of these formulas are difficult to estimate accurately, and the formulas must be adapted to local circumstances, making them difficult to use. I suspect that few software project managers use these models to estimate the return on investment from component reusability.

Whether or not a component is reusable depends on its application domain, functionality, and generality. If the domain is a general one and the component implements standard functionality in that domain, then it is more likely to be reusable. As you add generality to a component, you increase its reusability because it can be applied in a wider range of environments. Unfortunately, this normally means that the component has more operations and is more complex, which makes the component harder to understand and use.

There is, therefore, a trade-off between the reusability and understandability of a component. To make a component reusable you have to provide a set of generic interfaces with operations that cater to all of the ways in which the component could be used. Reusability adds complexity and hence reduces component understandability. This makes it more difficult and time consuming to decide whether a component is suitable for reuse. Because of the time involved in understanding a reusable component, it is sometimes more cost-effective to reimplement a simpler component with the specific functionality that is required.

A potential source of components is legacy systems. As I discussed in Chapter 9, legacy systems are systems that fulfill an important business function but are written using obsolete software technologies. As a result, it may be difficult to use them with new systems. However, if you convert these old systems to components, their functionality can be reused in new applications.

Of course, these legacy systems do not normally have clearly defined "requires" and "provides" interfaces. To make these components reusable, you have to create a wrapper that defines the component interfaces. The wrapper hides the complexity of the underlying code and provides an interface for external components to access services that are provided. Although this wrapper is a fairly complex piece of software, the cost of wrapper development may be significantly less than the cost of reimplementing the legacy system.

Once you have developed and tested a reusable component or service, it then has to be managed for future reuse. Management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component, and keeping track of different component versions. If the component is open-source, you may make it available in a public repository such as GitHub or Sourceforge. If it is intended for use in a company, then you may use an internal repository system.

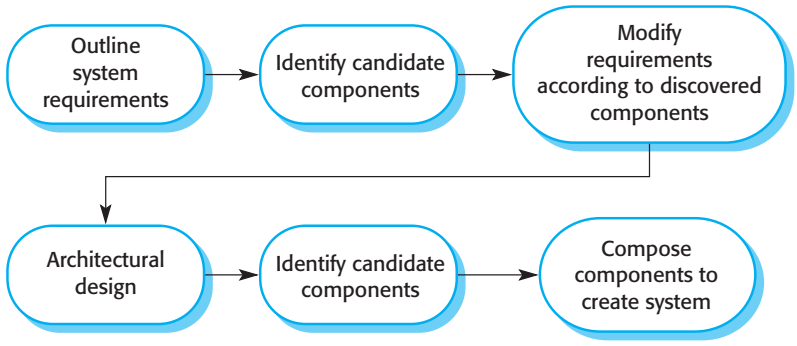


Figure 16.7 CBSE with reuse

A company with a reuse program may carry out some form of component certification before the component is made available for reuse. Certification means that someone apart from the developer checks the quality of the component. They test the component and certify that it has reached an acceptable quality standard, before it is made available for reuse. However, this process can be expensive, and so many companies simply leave testing and quality checking to the component developers.

16.2.2 CBSE with reuse

The successful reuse of components requires a development process tailored to including reusable components in the software being developed. The CBSE with reuse process has to include activities that find and integrate reusable components. The structure of such a process was discussed in Chapter 2, and Figure 16.7 shows the principal activities within that process. Some of these activities, such as the initial discovery of user requirements, are carried out in the same way as in other software processes. However, the essential differences between CBSE with reuse and software processes for original software development are as follows:

1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. Requirements that are too specific limit the number of components that could meet these requirements. However, unlike incremental development, you need a complete description of the requirements so that you can identify as many components as possible for reuse.
2. Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported by the reusable components. Users may be willing to change their minds if this means cheaper or quicker system delivery.
3. There is a further component search and design refinement activity after the system architecture has been designed. Apparently, usable components may turn out



Figure 16.8 The component identification process

to be unsuitable or may not work properly with other chosen components. You may have to find alternatives to these components. Further requirements changes may therefore be necessary, depending on the functionality of these components.

4. Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing adaptors that reconcile the interfaces of incompatible components. Of course, additional functionality may also be required over and above that provided by reused components.

The architectural design stage is particularly important. Jacobsen et al. (Jacobsen, Griss, and Jonsson 1997) found that defining a robust architecture is critical for successful reuse. During the architectural design activity, you may choose a component model and implementation platform. However, many companies have a standard development platform (e.g., .NET), so the component model is predetermined. As I discussed in Chapter 6, you also establish the high-level architecture of the system at this stage and make decisions about system distribution and control.

An activity that is unique to the CBSE process is identifying candidate components or services for reuse. This involves a number of subactivities, as shown in Figure 16.8. Initially, your focus should be on search and selection. You need to convince yourself that components are available to meet your requirements. Obviously, you should do some initial checking that the component is suitable, but detailed testing may not be required. In the later stage, after the system architecture has been designed, you should spend more time on component validation. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes.

The first step in identifying components is to look for components that are available within your company or from trusted suppliers. There are few component vendors, so you are most likely to be looking for components that have been developed in your own organization or in the repositories of open-source software that are available. Software development companies can build their own database of reusable components without the risks inherent in using components from external suppliers. Alternatively, you may decide to search code libraries available on the web, such as Sourceforge, GitHub, or Google Code, to see if source code for the component that you need is available.

Once the component search process has identified possible components, you have to select candidate components for assessment. In some cases, this will be a straightforward task. Components on the list will directly implement the user requirements, and there will not be competing components that match these requirements. In other cases, however, the selection process is more complex. There will not be a clear mapping of requirements onto components. You may find that several components have to be integrated to meet a

The Ariane 5 launcher failure

While developing the Ariane 5 space launcher, the designers decided to reuse the inertial reference software that had performed successfully in the Ariane 4 launcher. The inertial reference software maintains the stability of the rocket. The designers decided to reuse this without change (as you would do with components), although it included additional functionality that was not required in Ariane 5.

In the first launch of Ariane 5, the inertial navigation software failed, and the rocket could not be controlled. The rocket and its payload were destroyed. The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the runtime system to shut down the inertial reference system, and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

This illustrates an important problem with software reuse. Software may be based on assumptions about the context where the system will be used, and these assumptions may not be valid in a different situation.

More information about this failure is available at: http://software-engineering-book.com/case-studies/ariane5/

Figure 16.9 An example of validation failure with reused software

specific requirement or group of requirements. You therefore have to decide which of these component compositions provide the best coverage of the requirements.

Once you have selected components for possible inclusion in a system, you should then validate them to check that they behave as advertised. The extent of the validation required depends on the source of the components. If you are using a component that has been developed by a known and trusted source, you may decide that component testing is unnecessary. You simply test the component when it is integrated with other components. On the other hand, if you are using a component from an unknown source, you should always check and test that component before including it in your system.

Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests. The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. Components are usually specified informally, with the only formal documentation being their interface specification. This may not include enough information for you to develop a complete set of tests that would convince you that the component's advertised interface is what you require.

As well as testing that a component for reuse does what you require, you may also have to check that the component does not include malicious code or functionality that you don't need. Professional developers rarely use components from untrusted sources, especially if these sources do not provide source code. Therefore, the malicious code problem does not usually arise. However, reused components may often contain functionality that you don't need, and you have to check that this functionality will not interfere with your use of the component.

The problem with unnecessary functionality is that it may be activated by the component itself. While this may have no effect on the application reusing the component, it can slow down the component, cause it to produce surprising results or, in exceptional cases, cause serious system failures. Figure 16.9 summarizes a situation where the failure of a reused software system, which had unnecessary functionality, led to catastrophic system failure.

The problem in the Ariane 5 launcher arose because the assumptions made about the software for Ariane 4 were invalid for Ariane 5. This is a general problem with reusable components. They are originally implemented for a specific application environment and, naturally, embed assumptions about that environment. These assumptions are rarely documented, so when the component is reused, it is impossible to develop tests to check if the assumptions are still valid. If you are reusing a component in a new environment, you may not discover the embedded environmental assumptions until you use the component in an operational system.

16.3 Component composition

Component composition is the process of integrating components with each other, and with specially written "glue code" to create a system or another component. You can compose components in several different ways, as shown in Figure 16.10. From left to right these diagrams illustrate sequential composition, hierarchical composition, and additive composition. In the discussion below, I assume that you are composing two components (A and B) to create a new component:

1. *Sequential composition* In a sequential composition, you create a new component from two existing components by calling the existing components in sequence. You can think of the composition as a composition of the "provides interfaces." That is, the services offered by component A are called, and the results returned by A are then used in the call to the services offered by component B. The components do not call each other in sequential composition but are called by the external application. This type of composition may be used with embedded or service components.

   Some extra glue code may be required to call the component services in the right order and to ensure that the results delivered by component A are compatible with the inputs expected by component B. The "glue code" transforms these outputs to be of the form expected by component B.

2. *Hierarchical composition* This type of composition occurs when one component calls directly on the services provided by another component. That is, component A calls component B. The called component provides the services that are required by the calling component. Therefore, the "provides" interface of the called component must be compatible with the "requires" interface of the calling component.

   Component A calls on component B directly, and, if their interfaces match, there may be no need for additional code. However, if there is a mismatch between the "requires" interface of A and the "provides" interface of B, then some conversion code may be required. As services do not have a "requires" interface, this mode of composition is not used when components are implemented as services accessed over the web.

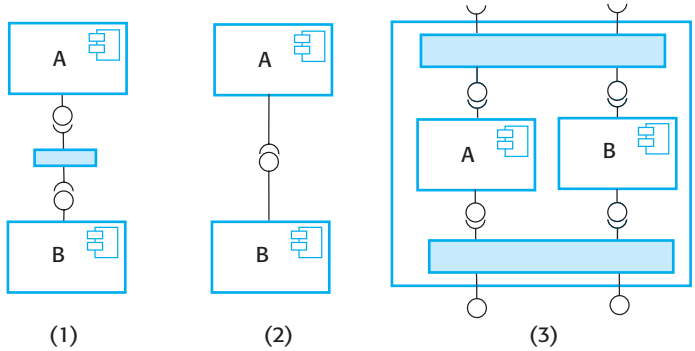


Figure 16.10 Types of component composition

3. *Additive composition* This occurs when two or more components are put together (added) to create a new component, which combines their functionality. The "provides" interface and "requires" interface of the new component are a combination of the corresponding interfaces in components A and B. The components are called separately through the external interface of the composed component and may be called in any order. A and B are not dependent and do not call each other. This type of composition may be used with embedded or service components.

You might use all the forms of component composition when creating a system. In all cases, you may have to write "glue code" that links the components. For example, for sequential composition, the output of component A typically becomes the input to component B. You need intermediate statements that call component A, collect the result, and then call component B, with that result as a parameter. When one component calls another, you may need to introduce an intermediate component that ensures that the "provides" interface and the "requires" interface are compatible.

When you write new components especially for composition, you should design the interfaces of these components so that they are compatible with other components in the system. You can therefore easily compose these components into a single unit. However, when components are developed independently for reuse, you will often be faced with interface incompatibilities. This means that the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur:

1. *Parameter incompatibility* The operations on each side of the interface have the same name, but their parameter types or the number of parameters are different. In Figure 16.11, the location parameter returned by addressFinder is incompatible with the parameters required by the displayMap and printMap methods in mapDB.
2. *Operation incompatibility* The names of the operations in the provides and "requires" interfaces are different. This is a further incompatibility between the components shown in Figure 16.11.
3. *Operation incompleteness* The "provides" interface of a component is a subset of the "requires" interface of another component, or vice versa.

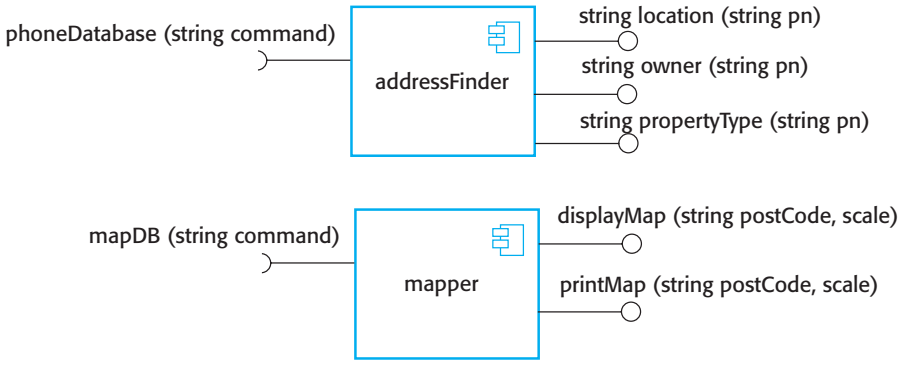


Figure 16.11 Components with incompatible interfaces

In all cases, you tackle the problem of incompatibility by writing an adaptor that reconciles the interfaces of the two components being reused. An adaptor component converts one interface to another.

The precise form of the adaptor depends on the type of composition. Sometimes, as in the next example, the adaptor takes a result from one component and converts it into a form where it can be used as an input to another. In other cases, the adaptor may be called by component A as a proxy for component B. This situation occurs if A wishes to call B, but the details of the "requires" interface of A do not match the details of the "provides" interface of B. The adaptor reconciles these differences by converting its input parameters from A into the required input parameters for B. It then calls B to deliver the services required by A.

To illustrate adaptors, consider the two simple components shown in Figure 16.11, whose interfaces are incompatible. These might be part of a system used by the emergency services. When the emergency operator takes a call, the phone number is input to the addressFinder component to locate the address. Then, using the mapper component, the operator prints a map to be sent to the vehicle dispatched to the emergency.

The first component, addressFinder, finds the address that matches a phone number. It can also return the owner of the property associated with the phone number and the type of property. The mapper component takes a post code (in the United States, a standard ZIP code with the additional four digits identifying property location) and displays or prints a street map of the area around that code at a specified scale.

These components are composable in principle because the property location includes the post or ZIP code. However, you have to write an adaptor component called postCodeStripper that takes the location data from addressFinder and strips out the post code. This post code is then used as an input to mapper, and the street map is displayed at a scale of 1:10,000. The following code, which is an example of sequential composition, illustrates the sequence of calls that is required to implement this process:

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

Another case in which an adaptor component may be used is in hierarchical composition, where one component wishes to make use of another but there is an incompatibility

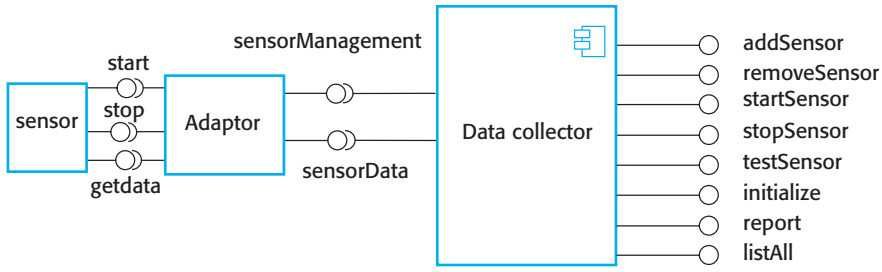


Figure 16.12 An adaptor linking a data collector and a sensor

between the "provides" interface and "requires" interface of the components in the composition. I have illustrated the use of an adaptor in Figure 16.12 where an adaptor is used to link a data collector and a sensor component. These could be used in the implementation of a wilderness weather station system, as discussed in Chapter 7.

The sensor and data collector components are composed using an adaptor that reconciles the "requires" interface of the data collection component with the "provides" interface of the sensor component. The data collector component has been designed with a generic "requires" interface that supports sensor data collection and sensor management. For each of these operations, the parameter is a text string representing the specific sensor commands. For example, to issue a collect command, you would say sensorData("collect"). As I have shown in Figure 16.12, the sensor itself has separate operations such as start, stop, and getdata.

The adaptor parses the input string, identifies the command (e.g., collect), and then calls Sensor.getdata to collect the sensor value. It then returns the result (as a character string) to the data collector component. This interface style means that the data collector can interact with different types of sensor. A separate adaptor, which converts the sensor commands from Data collector to the sensor interface, is implemented for each type of sensor.

The above discussion of component composition assumes you can tell from the component documentation whether or not interfaces are compatible. Of course, the interface definition includes the operation name and parameter types, so you can make some assessment of the compatibility from this. However, you depend on the component documentation to decide whether the interfaces are semantically compatible.

To illustrate this problem, consider the composition shown in Figure 16.13. These components are used to implement a system that downloads images from a camera and stores them in a photograph library. The system user can provide additional information to describe and catalog the photograph. To avoid clutter, I have not shown all interface

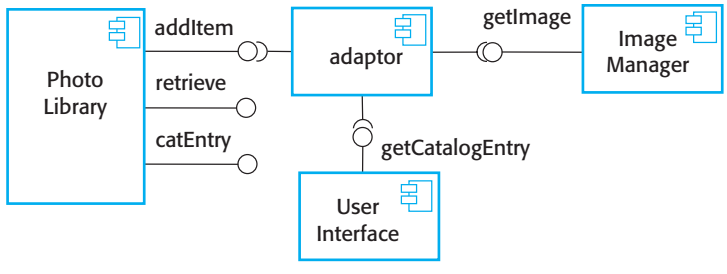


Figure 16.13 Photo library composition

```
— The context keyword names the component to which the conditions apply

context addItem

— The preconditions specify what must be true before execution of addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

— The postconditions specify what is true after execution
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:    PhotoLibrary.retrieve(pid) <> null ;

post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre—1
```

Figure 16.14 The OCL description of the Photo Library interface

methods here. Rather, I simply show the methods that are needed to illustrate the component documentation problem. The methods in the interface of **Photo Library** are:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid) ;
```

Assume that the documentation for the **addItem** method in Photo Library is:

This method adds a photograph to the library and associates the photograph identifier and catalog descriptor with the photograph.

This description appears to explain what the component does, but consider the following questions:

- What happens if the photograph identifier is already associated with a photograph in the library?
- Is the photograph descriptor associated with the catalog entry as well as the photograph? That is, if you delete the photograph, do you also delete the catalog information?

There is not enough information in the informal description of **addItem** to answer these questions. Of course, it is possible to add more information to the natural language description of the method, but in general, the best way to resolve ambiguities is to use a formal language to describe the interface. The specification shown in Figure 16.14 is part of the description of the interface of **Photo Library** that adds information to the informal description.

Figure 16.14 shows pre- and postconditions that are defined in a notation based on the object constraint language (OCL), which is part of the UML (Warmer and Kleppe 2003). OCL is designed to describe constraints in UML object models; it allows you to express predicates that must always be true, that must be true before a method has executed; and that must be true after a method has executed. These are invariants, preconditions, and postconditions. To access the value of a variable before an operation, you add @pre after its name. Therefore, using age as an example:

```
age = age@pre + 1
```

This statement means that the value of age after an operation is one more than it was before that operation.

OCL-based approaches are primarily used in model-based software engineering to add semantic information to UML models. The OCL descriptions may be used to drive code generators in model-driven engineering. The general approach has been derived from Meyer's Design by Contract approach (Meyer 1992), in which the interfaces and obligations of communicating objects are formally specified and enforced by the runtime system. Meyer suggests that using Design by Contract is essential if we are to develop trusted components (Meyer 2003).

Figure 16.14 shows the specification for the addItem and delete methods in **Photo Library**. The method being specified is indicated by the keyword **context** and the pre- and postconditions by the keywords **pre** and **post**. The preconditions for **addItem** state that:

1. There must not be a photograph in the library with the same identifier as the photograph to be entered.
2. The library must exist—assume that creating a library adds a single item to it so that the size of a library is always greater than zero.
3. The postconditions for **addItem** state that:

   The size of the library has increased by 1 (so only a single entry has been made).

   If you retrieve using the same identifier, then you get back the photograph that you added.

   If you look up the catalog using that identifier, you get back the catalog entry that you made.

The specification of **delete** provides further information. The precondition states that to delete an item, it must be in the library, and, after deletion, the photo can no longer be retrieved and the size of the library is reduced by 1. However, **delete** does not delete the catalog entry—you can still retrieve it after the photo has been deleted. The reason for this is that you may wish to maintain information in the catalog about why a photo was deleted, its new location, and so on.

When you create a system by composing components, you may find that there are potential conflicts between functional and non-functional requirements, the need to deliver a system as quickly as possible, and the need to create a system that

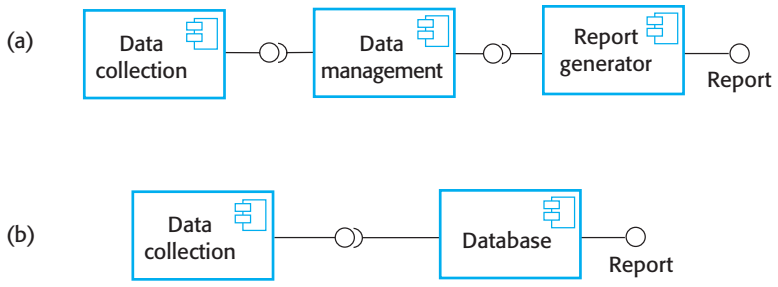


Figure 16.15 Data collection and report generation components

can evolve as requirements change. You may have to take trade-offs into account for component decisions:

1. What composition of components is most effective for delivering the functional requirements for the system?
2. What composition of the components will make it easier to adapt the composite component when its requirements change?
3. What will be the emergent properties of the composed system? These properties include performance and dependability. You can only assess these properties once the complete system is implemented.

Unfortunately, in many situations the solutions to the composition problems may conflict. For example, consider a situation such as that illustrated in Figure 16.15, where a system can be created through two alternative compositions. The system is a data collection and reporting system where data is collected from different sources, stored in a database, and then different reports summarizing that data are produced.

Here, there is a potential conflict between adaptability and performance. Composition (a) is more adaptable, but composition (b) is likely to be faster and more reliable. The advantages of composition (a) are that reporting and data management are separate, so there is more flexibility for future change. The data management system could be replaced, and, if reports are required that the current reporting component cannot produce, that component can also be replaced without having to change the data management component.

In composition (b), a database component with built-in reporting facilities (e.g., Microsoft Access) is used. The key advantage of composition (b) is that there are fewer components, so this will be a faster implementation because there are no component communication overheads. Furthermore, data integrity rules that apply to the database will also apply to reports. These reports will not be able to combine data in incorrect ways. In composition (a), there are no such constraints, so errors in reports could occur.

In general, a good composition principle to follow is the principle of separation of concerns. That is, you should try to design your system so that each component has a clearly defined role. Ideally, component roles should not overlap. However, it may be cheaper to buy one multifunctional component rather than two or three separate components. Furthermore, dependability or performance penalties may be incurred when multiple components are used.

KEY POINTS

- Component-based software engineering is a reuse-based approach to defining, implementing, and composing loosely coupled independent components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by a set of public interfaces. Components can be composed with other components without knowledge of their implementation and can be deployed as an executable unit.
- Components may be implemented as executable routines that are included in a system or as external services that are referenced from within a system.
- A component model defines a set of standards for components, including interface standards, usage standards, and deployment standards. The implementation of the component model provides a set of common services that may be used by all components.
- During the CBSE process, you have to interleave the processes of requirements engineering and system design. You have to trade off desirable requirements against the services that are available from existing reusable components.
- Component composition is the process of "wiring" components together to create a system. Types of composition include sequential composition, hierarchical composition, and additive composition.
- When composing reusable components that have not been written for your application, you may need to write adaptors or "glue code" to reconcile the different component interfaces.
- When choosing compositions, you have to consider the required functionality of the system, the non-functional requirements, and the ease with which one component can be replaced when the system is changed.

FURTHER READING

Component Software: Beyond Object-Oriented Programming, 2nd ed. This updated edition of the first book on CBSE covers technical and nontechnical issues in CBSE. It has more detail on specific technologies than Heineman and Councill's book and includes a thorough discussion of market issues. (C. Szyperski, Addison-Wesley, 2002).

"Specification, Implementation and Deployment of Components." A good introduction to the fundamentals of CBSE. The same issue of the *CACM* includes articles on components and component-based development. (I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, *Comm. ACM,* 45(10), October 2002) http://dx.doi.org/10.1145/570907.570928

"Software Component Models." This comprehensive discussion of commercial and research component models classifies these models and explains the differences between them. (K-K. Lau and Z. Wang, *IEEE Transactions on Software Engineering,* 33 (10), October 2007) http://dx.doi.org/10.1109/TSE.2007.70726

"Software Components Beyond Programming: From Routines to Services." This is the opening article in a special issue of the magazine that includes several articles on software components. This article discusses the evolution of components and how service-oriented components are replacing executable program routines. (I. Crnkovic, J. Stafford, and C. Szyperski, *IEEE Software*, 28 (3), May/June 2011) http://dx.doi.org/10.1109/MS.2011.62

Object Constraint Language (OCL) Tutorial. A good introduction to the use of the object-constraint language. (J. Cabot, 2012) http://modeling-languages.com/ocl-tutorial/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

A more detailed discussion of the Ariane5 accident:

http://software-engineering-book.com/case-studies/ariane5/

EXERCISES

16.1. What are the design principles underlying the CBSE that support the construction of understandable and maintainable software?

16.2. The principle of component independence means that it ought to be possible to replace one component with another that is implemented in a completely different way. Using an example, explain how such component replacement could have undesired consequences and may lead to system failure.

16.3. In a reusable component, what are the critical characteristics that are emphasized when the component is viewed as a service?

16.4. Why is it important that components should be based on a standard component model?

16.5. Using an example of a component that implements an abstract data type such as a stack or a list, show why it is usually necessary to extend and adapt components for reuse.

16.6. What are the essential differences between CBSE with reuse and software processes for original software development?

16.7. Design the "provides" interface and the "requires" interface of a reusable component that may be used to represent a patient in the Mentcare system that I introduced in Chapter 1.

16.8. Using examples, illustrate the different types of adaptor needed to support sequential composition, hierarchical composition, and additive composition.

16.9. Design the interfaces of components that might be used in a system for an emergency control room. You should design interfaces for a call-logging component that records calls made, and a vehicle discovery component that, given a post code (zip code) and an incident type, finds the nearest suitable vehicle to be dispatched to the incident.

16.10. It has been suggested that an independent certification authority should be established. Vendors would submit their components to this authority, which would validate that the component was trustworthy. What would be the advantages and disadvantages of such a certification authority?

REFERENCES

Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 5–20. Boston: Addison-Wesley.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

Kotonya, G. 2003. "The CBSE Process: Issues and Future Visions." In *2nd CBSEnet Workshop*. Budapest, Hungary. http://miro.sztaki.hu/projects/cbsenet/budapest/presentations/Gerald-CBSEProcess.ppt

Lau, K-K., and Z. Wang. 2007. "Software Component Models." *IEEE Trans. on Software Eng.* 33 (10): 709–724. doi:10.1109/TSE.2007.70726.

Meyer, B. 1992. "Applying Design by Contract." *IEEE Computer* 25 (10): 40–51. doi:10.1109/2.161279.

———. 2003. "The Grand Challenge of Trusted Components." In *Proc. 25th Int. Conf. on Software Engineering*. Portland, OR: IEEE Press. doi:10.1109/ICSE.2003.1201252.

Mili, H., A. Mili, S. Yacoub, and E. Addy. 2002. *Reuse-Based Software Engineering*. New York: John Wiley & Sons.

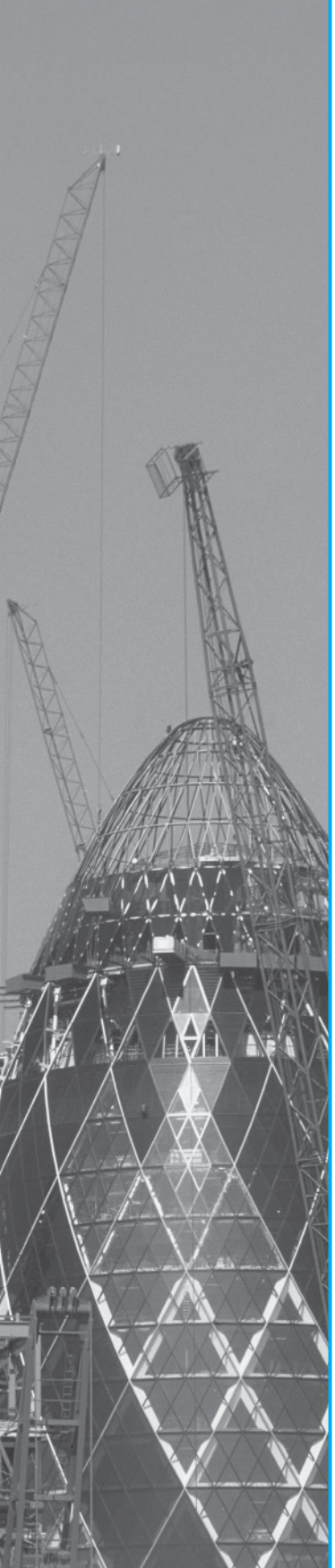
Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Harlow, UK: Addison-Wesley.

Warmer, J., and A. Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston: Addison-Wesley.

Weinreich, R., and J. Sametinger. 2001. "Component Models and Component Services: Concepts and Principles." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 33–48. Boston: Addison-Wesley.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.



17

Distributed software engineering

Objectives

The objective of this chapter is to introduce distributed systems engineering and distributed systems architectures. When you have read this chapter, you will:

- know the key issues that have to be considered when designing and implementing distributed software systems;
- understand the client–server computing model and the layered architecture of client–server systems;
- have been introduced to commonly used patterns for distributed systems architectures and know the types of system for which each architectural pattern is applicable;
- understand the notion of software as a service, providing web-based access to remotely deployed application systems.

Contents

17.1 Distributed systems
17.2 Client–server computing
17.3 Architectural patterns for distributed systems
17.4 Software as a service

Most computer-based systems are now distributed systems. A distributed system is one involving several computers rather than a single application running on a single machine. Even apparently self-contained applications on a PC or laptop, such as image editors, are distributed systems. They execute on a single computer system but often rely on remote cloud systems for update, storage, and other services. Tanenbaum and Van Steen (Tanenbaum and Van Steen 2007) define a distributed system to be "a collection of independent computers that appears to the user as a single coherent system."†

When you are designing a distributed system, there are specific issues that have to be taken into account simply because the system is distributed. These issues arise because different parts of the system are running on independently managed computers and because the characteristics of the network, such as latency and reliability, may have to be considered in your design.

Coulouris et al. (Coulouris et al. 2011) identify the five benefits of developing systems as distributed systems:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
2. *Openness* Distributed systems are normally open systems—systems designed around standard Internet protocols so that equipment and software from different vendors can be combined.
3. *Concurrency* In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
4. *Scalability* In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
5. *Fault tolerance* The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 11). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.‡

Distributed systems are inherently more complex than centralized systems. This makes them more difficult to design, implement, and test. It is harder to understand the emergent properties of distributed systems because of the complexity of the interactions between system components and system infrastructure. For example, rather than being dependent on the execution speed of one processor, system performance

†Tanenbaum, A. S., and M. Van Steen. 2007. Distributed Systems: Principles and Paradigms, 2nd Ed. Upper Saddle River, NJ: Prentice-Hall.

‡Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. Distributed Systems: Concepts and Design, 5th Edition. Harlow, UK.: Addison Wesley.

depends on network bandwidth, network load, and the speed of other computers that are part of the system. Moving resources from one part of the system to another can significantly affect the system's performance.

Furthermore, as all users of the WWW know, distributed systems are unpredictable in their response. Response time depends on the overall load on the system, its architecture, and the network load. As all of these factors may change over a short time, the time taken to respond to a user request may change significantly from one request to another.

The most important developments that have affected distributed software systems in the past few years are service-oriented systems and the advent of cloud computing, delivering infrastructure, platforms, and software as a service. In this chapter, I focus on general issues of distributed systems, and in Section 17.4 I cover the idea of software as a service. In Chapter 18, I discuss other aspects of service-oriented software engineering.

17.1 Distributed systems

As I discussed in the introduction to this chapter, distributed systems are more complex than systems that run on a single processor. This complexity arises because it is practically impossible to have a top-down model of control of these systems. The nodes in the system that deliver functionality are often independent systems that are managed and controlled by their owners. There is no single authority in charge of the entire distributed system. The network connecting these nodes is also a separately managed system. It is a complex system in its own right and cannot be controlled by the owners of systems using the network. There is, therefore, an inherent unpredictability in the operation of distributed systems that has to be taken into account when you are designing a system.

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. *Transparency* To what extent should the distributed system appear to the user as a single system? When is it useful for users to understand that the system is distributed?
2. *Openness* Should a system be designed using standard protocols that support interoperability, or should more specialized protocols be used? Although standard network protocols are now universally used, this is not the case for higher levels of interaction, such as service communication.
3. *Scalability* How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. *Security* How can usable security policies be defined and implemented that apply across a set of independently managed systems?
5. *Quality of service* How should the quality of service that is delivered to system users be specified, and how should the system be implemented to deliver an acceptable quality of service to all users.
6. *Failure management* How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?



CORBA—Common Object Request Broker Architecture

CORBA was proposed as a specification for a middleware system in the 1990s by the Object Management Group. It was intended as an open standard that would allow the development of middleware to support distributed component communications and execution, as well as provide a set of standard services that could be used by these components.

Several implementations of CORBA were produced, but the system was not widely adopted. Users preferred proprietary systems such as those from Microsoft or Oracle, or they moved to service-oriented architectures.

http://software-engineering-book.com/web/corba/

In an ideal world, the fact that a system is distributed would be transparent to users. Users would see the system as a single system whose behavior is not affected by the way that the system is distributed. In practice, this is impossible to achieve because there is no central control over the system as a whole. As a result, individual computers in a system may behave differently at different times. Furthermore, because it always takes a finite length of time for signals to travel across a network, network delays are unavoidable. The length of these delays depends on the location of resources in the system, the quality of the user's network connection, and the network load.

To make a distributed system transparent (i.e., conceal its distributed nature), you have to hide the underlying distribution. You create abstractions that hide the system resources so that the location and implementation of these resources can be changed without having to change the distributed application. Middleware (discussed in Section 17.1.2) is used to map the logical resources referenced by a program onto the actual physical resources and to manage resource interactions.

In practice, it is impossible to make a system completely transparent, and users, generally, are aware that they are dealing with a distributed system. You may therefore decide that it is best to expose the distribution to users. They can then be prepared for some of the consequences of distribution such as network delays and remote node failures.

Open distributed systems are built according to generally accepted standards. Components from any supplier can therefore be integrated into the system and can interoperate with the other system components. At the networking level, openness is now taken for granted, with systems conforming to Internet protocols, but at the component level, openness is still not universal. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.

The CORBA standard (Pope 1997), developed in the 1990s, was intended to be the universal standard for open distributed systems, However, the CORBA standard never achieved a critical mass of adopters. Rather, many companies preferred to develop systems using proprietary standards for components from companies such as Sun (now Oracle) and Microsoft. These provided better implementations and support software and better long-term support for industrial protocols.

Web service standards (discussed in Chapter 18) for service-oriented architectures were developed to be open standards. However, these standards have met with significant resistance because of their perceived inefficiency. Many developers of service-based systems have opted instead for so-called RESTful protocols because

these have an inherently lower overhead than web service protocols. The use of RESTful protocols is not standardized.

The scalability of a system reflects its ability to deliver high-quality service as demands on the system increase. The three dimensions of scalability are size, distribution, and manageability.

1. *Size* It should be possible to add more resources to a system to cope with increasing numbers of users. Ideally, then, as the number of users increases, the system should increase in size automatically to handle the increased number of users.
2. *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance. As new components are added, it should not matter where these are located. Large companies can often make use of computing resources in their different facilities around the world.
3. *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations. This is one of the most difficult challenges of scale as it involves managers communicating and agreeing on management policies. In practice, the manageability of a system is often the factor that limits the extent to which it can be scaled.

Changing the size of a system may involve either scaling up or scaling out. Scaling up means replacing resources in the system with more powerful resources. For example, you may increase the memory in a server from 16 Gb to 64 Gb. Scaling out means adding more resources to the system (e.g., an extra web server to work alongside an existing server). Scaling out is often more cost-effective than scaling up, especially now that cloud computing makes it easy to add or remove servers from a system. However, this only provides performance improvements when concurrent processing is possible.

I have discussed general security issues and issues of security engineering in Part 2 of this book. When a system is distributed, attackers may target any of the individual system components or the network itself. If a part of the system is successfully attacked, then the attacker may be able to use this as a "back door" into other parts of the system.

A distributed system must defend itself against the following types of attack:

1. *Interception*, where an attacker intercepts communications between parts of the system so that there is a loss of confidentiality.
2. *Interruption*, where system services are attacked and cannot be delivered as expected. Denial-of-service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
3. *Modification*, where an attacker gains access to the system and changes data or system services.
4. *Fabrication*, where an attacker generates information that should not exist and then uses this information to gain some privileges. For example, an attacker may generate a false password entry and use this to gain access to a system.

The major difficulty in distributed systems is establishing a security policy that can be reliably applied to all of the components in a system. As I discussed in Chapter 13, a security policy sets out the level of security to be achieved by a system. Security mechanisms, such as encryption and authentication, are used to enforce the security policy. The difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. Security compromises may have to be made in order to allow the systems to work together.

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that are acceptable to its users. Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable for two reasons:

1. It may not be cost-effective to design and configure the system to deliver a high quality of service under peak load. The peak demands may mean that you need many extra servers than normal to ensure that response times are maintained. This problem has been lessened by the advent of cloud computing where cloud servers may be rented from a cloud provider for as long as they are required. As demand increases, extra servers can be automatically added.
2. The quality-of-service parameters may be mutually contradictory. For example, increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid.

Quality of service is particularly important when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand. Systems dealing with sound and video should include quality of service negotiation and management components. These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. Failure is so ubiquitous that one flippant definition of a distributed system suggested by Leslie Lamport, a prominent distributed systems researcher, is:

> *You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done.*[†]

This is even truer now that more and more systems are executing in the cloud. Failure management involves applying the fault-tolerance techniques discussed in Chapter 11. Distributed systems should therefore include mechanisms for discovering whether a component of the system has failed, should continue to deliver as many services as possible in spite of that failure, and, as far as possible, should automatically

[†]Leslie Lamport, in Ross J. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems (2nd ed.), Wiley (April 14, 2008).

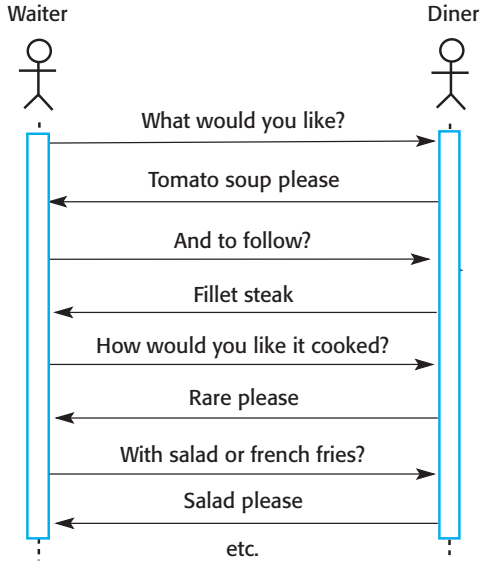


Figure 17.1 Procedural interaction between a diner and a waiter

recover from the failure. One important benefit of cloud computing is that it has dramatically reduced the cost of providing redundant system components.

17.1.1 Models of interaction

Two fundamental types of interaction may take place between the computers in a distributed computing system: procedural interaction and message-based interaction. Procedural interaction involves one computer calling on a known service offered by some other computer and waiting for that service to be delivered. Message-based interaction involves the "sending" computer defining information about what is required in a message, which is then sent to another computer. Messages usually transmit more information in a single interaction than a procedure call to another machine.

To illustrate the difference between procedural and message-based interaction, consider a situation where you are ordering a meal in a restaurant. When you have a conversation with the waiter, you are involved in a series of synchronous, procedural interactions that define your order. You make a request, the waiter acknowledges that request, you make another request, which is acknowledged, and so on. This is comparable to components interacting in a software system where one component calls methods from other components. The waiter writes down your order along with the order of other people with you. He or she then passes this order, which includes details of everything that has been ordered, to the kitchen to prepare the food. Essentially, the waiter is passing a message to the kitchen staff, defining the food to be prepared. This is message-based interaction.

I have illustrated this kind of interaction in Figure 17.1, which shows the synchronous ordering process as a series of calls, and in Figure 17.2, which shows a hypothetical XML message that defines an order made by the table of three people. The difference between these forms of information exchange is clear. The waiter takes the order as a series of

```
<starter>
   <dish name = "soup" type = "tomato" />
   <dish name = "soup" type = "fish" />
   <dish name = "pigeon salad" />
</starter>
<main course>
   <dish name = "steak" type = "sirloin" cooking = "medium" />
   <dish name = "steak" type = "fillet" cooking = "rare" />
   <dish name = "sea bass">
</main>
<accompaniment>
   <dish name = "french fries" portions = "2" />
   <dish name = "salad" portions = "1" />
</accompaniment>
```

Figure 17.2 Message-based interaction between a waiter and the kitchen staff

interactions, with each interaction defining part of the order. However, the waiter has a single interaction with the kitchen where the message defines the complete order.

Procedural communication in a distributed system is usually implemented using remote procedure calls (RPCs). In an RPC, components have globally unique names (such as a URL). Using that name, a component can call on the services offered by another component as if it was a local procedure or method. System middleware intercepts this call and passes it on to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. In Java, remote method invocations (RMIs) are remote procedure calls.

Remote procedure calls require a "stub" for the called procedure to be accessible on the computer that is initiating the call. This stub defines the interface of the remote procedure. The stub is called, and it translates the procedure parameters into a standard representation for transmission to the remote procedure. Through the middleware, it then sends the request for execution to the remote procedure. The remote procedure uses library functions to convert the parameters into the required format, carries out the computation, and then returns the results via the "stub" that is representing the caller.

Message-based interaction normally involves one component creating a message that details the services required from another component. This message is sent to the receiving component via the system middleware. The receiver parses the message, carries out the computations, and creates a message for the sending component with the required results. This is then passed to the middleware for transmission to the sending component.

A problem with the RPC approach to interaction is that both the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other. In essence, an RPC has the same requirements as a local procedure or method call. By contrast, in a message-based approach, unavailability can be tolerated. If the system component that is processing the message is unavailable, the message simply stays in a queue until the receiver comes back online. Furthermore, it is not necessary for the sender to know the name of the message receiver and vice versa. They simply communicate with the middleware, which is responsible for ensuring that messages are passed to the appropriate system.

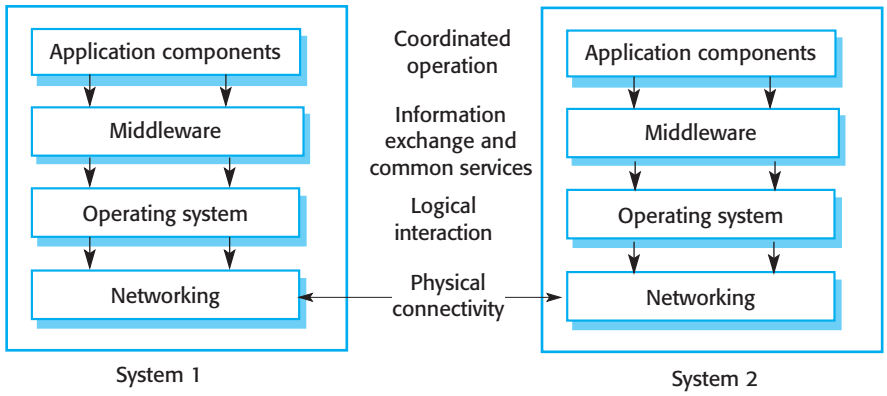


Figure 17.3 Middleware in a distributed system

17.1.2 Middleware

The components in a distributed system may be implemented in different programming languages and may execute on different types of processors. Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts and ensure that they can communicate and exchange data.

The term *middleware* is used to refer to this software—it sits in the middle between the distributed components of the system. This concept is illustrated in Figure 17.3, which shows that middleware is a layer between the operating system and application programs. Middleware is normally implemented as a set of libraries, which are installed on each distributed computer, plus a runtime system to manage communications.

Bernstein (Bernstein 1996) describes types of middleware that are available to support distributed computing. Middleware is general-purpose software that is usually bought off-the-shelf rather than written specially by application developers. Examples of middleware include software for managing communications with databases, transaction managers, data converters, and communication controllers.

In a distributed system, middleware provides two distinct types of support:

1. *Interaction support*, where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components. It may also support parameter conversion if different programming languages are used to implement components, event detection, communication, and so on.

2. *The provision of common services*, where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

I have already given examples of the interaction support that middleware can provide in Section 17.1.1. You use middleware to support remote procedure and remote method calls, message exchange, and so forth.

Common services are those services that may be required by different components irrespective of the functionality of these components. As I discussed in Chapter 16, these may include security services (authentication and authorization), notification and naming services, and transaction management services. For distributed components, you can think of these common services as being provided by a middleware container; for services, they are provided through shared libraries. You then deploy your component, and it can access and use these common services.

17.2 Client–server computing

Distributed systems that are accessed over the Internet are organized as client–server systems. In a client–server system, the user interacts with a program running on their local computer, such as a web browser or app on a mobile device. This interacts with another program running on a remote computer, such as a web server. The remote computer provides services, such as access to web pages, which are available to external clients. This client–server model, as I discussed in Chapter 6, is a general architectural model of an application. It is not restricted to applications distributed across several machines. You can also use it as a logical interaction model where the client and the server run on the same computer.

In a client–server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end-users. Clients need to be aware of the servers that are available but don't have to know anything about other clients. Clients and servers are separate processes, as shown in Figure 17.4. This figure illustrates a situation in which there are four servers (s1–s4) that deliver different services. Each service has a set of associated clients that access these services.

Figure 17.4 shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer. Figure 17.5 shows a possible arrangement where the 12 logical clients shown in Figure 17.4 are running on six computers. The four server processes are mapped onto two physical server computers.

Several different server processes may run on the same processor, but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine. Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, you should design the architecture of distributed client–server systems so that they are structured into several logical layers, with clear interfaces

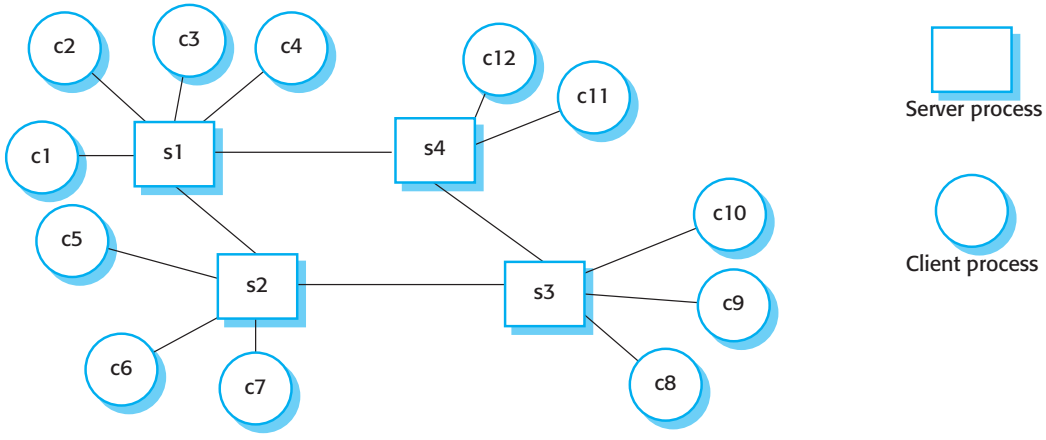


Figure 17.4 Client–server interaction

between these layers. This allows each layer to be distributed to a different computer. Figure 17.6 illustrates this model, showing an application structured into four layers:

1. *A presentation layer* that is concerned with presenting information to the user and managing all user interaction.
2. *A data-handling layer* that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, and so on.
3. *An application processing layer* that is concerned with implementing the logic of the application and so providing the required functionality to end-users.
4. *A database layer* that stores the data and provides transaction management and query services.

The following section explains how different client–server architectures distribute these logical layers in different ways. The client–server model also underlies the notion of software as a service (SaaS), an important way of deploying software and accessing it over the Internet. I cover this topic in Section 17.4.

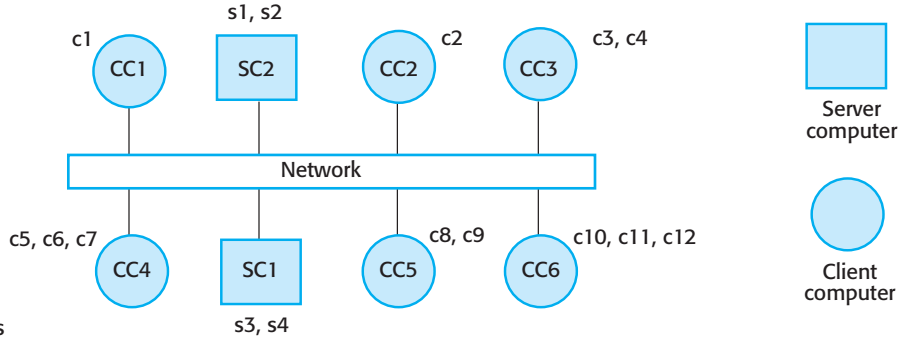


Figure 17.5 Mapping of clients and servers to networked computers

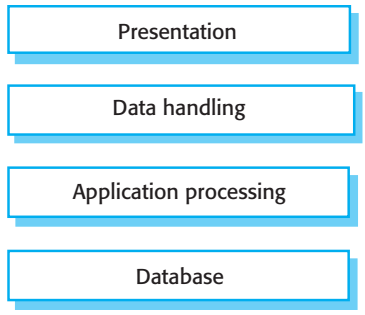


Figure 17.6 Layered architectural model for client–server application

17.3 Architectural patterns for distributed systems

As I explained in the introduction to this chapter, designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. Because no universal model of system organization is appropriate for all circumstances, various distributed architectural styles have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

In this section, I discuss five architectural styles:

1. *Master-slave architecture*, which is used in real-time systems in which guaranteed interaction response times are required.
2. *Two-tier client–server architecture*, which is used for simple client–server systems and in situations where it is important to centralize the system for security reasons.
3. *Multi-tier client–server architecture*, which is used when the server has to process a high volume of transactions.
4. *Distributed component architecture*, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.
5. *Peer-to-peer architecture*, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

17.3.1 Master–slave architectures

Master–slave architectures for distributed systems are commonly used in real-time systems. In those systems, there may be separate processors associated with data acquisition from the system's environment, data processing and computation,

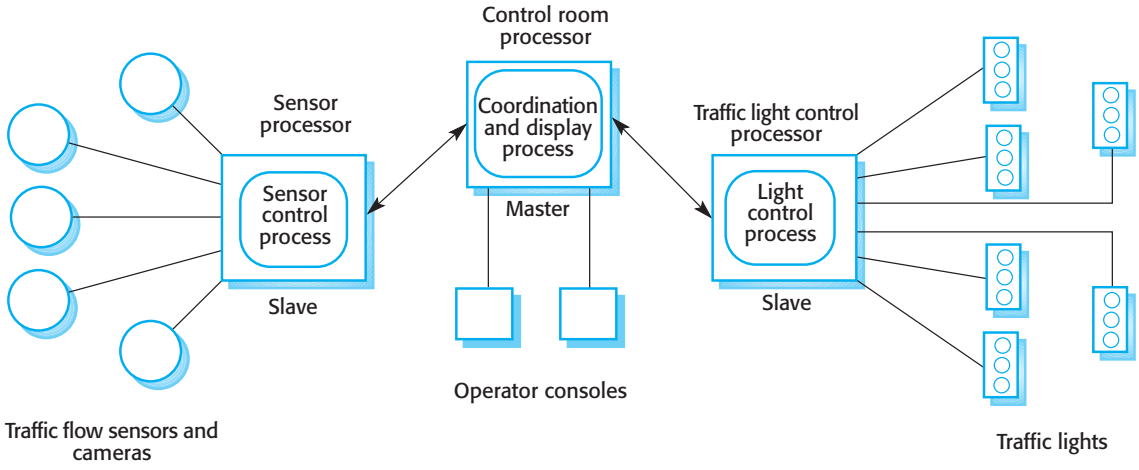


Figure 17.7 A traffic management system with a master–slave architecture

and actuator management. Actuators, as I discuss in Chapter 21, are devices controlled by the software system that act to change the system's environment. For example, an actuator may control a valve and change its state from "open" to "closed." The "master" process is usually responsible for computation, coordination, and communications, and it controls the "slave" processes. "Slave" processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

Figure 17.7 shows an example of this architectural model. A traffic control system in a city has three logical processes that run on separate processors. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing. The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences, and accepting operator commands to modify these sequences. The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware. The master control room system is itself organized as a client–server system, with the client processes running on the operator's consoles.

You use this master–slave model of a distributed system in situations where you can predict the distributed processing that is required and where processing can be easily localized to slave processors. This situation is common in real-time systems, where it is important to meet processing deadlines. Slave processors can be used for computationally intensive operations, such as signal processing and the management of equipment controlled by the system.

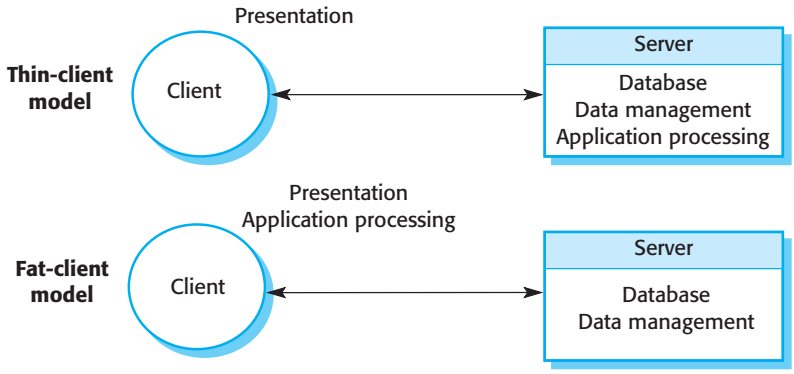


Figure 17.8 Thin- and fat-client architectural models

17.3.2 Two-tier client–server architectures

In Section 17.2, I explained the general organization of client–server systems in which part of the application system runs on the user's computer (the client), and part runs on a remote computer (the server). I also presented a layered application model (Figure 17.6) where the different layers in the system may execute on different computers.

A two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server. This is illustrated in Figure 17.8, which shows two forms of this architectural model:

1. *A thin-client model*, where the presentation layer is implemented on the client and all other layers (data handling, application processing, and database) are implemented on a server. The client presentation software is usually a web browser, but apps for mobile devices may also be available.
2. *A fat-client model*, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. In this case, the client software may be a specially written program that is tightly integrated with the server application.

The advantage of the thin-client model is that it is simple to manage the clients. This becomes a major issue when there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

The disadvantage of the thin-client approach, however, is that it places a heavy processing load on both the server and the network. The server is responsible for all computation, which may lead to the generation of significant network traffic between the client and the server. Implementing a system using this model may therefore require additional investment in network and server capacity.

The fat-client model makes use of available processing power on the computer running the client software, and distributes some or all of the application processing

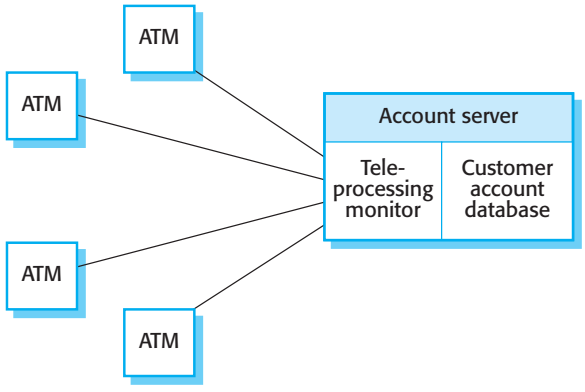


Figure 17.9 A fat-client architecture for an ATM system

and the presentation to the client. The server is essentially a transaction server that manages all database transactions. Data handling is straightforward as there is no need to manage the interaction between the client and the application processing system. The fat-client model requires system management to deploy and maintain the software on the client computer.

An example of a situation in which a fat-client architecture is used is in a bank ATM system, which delivers cash and other banking services to users. The ATM is the client computer, and the server is, typically, a mainframe running the customer account database. A mainframe computer is a powerful machine that is designed for transaction processing. It can therefore handle the large volume of transactions generated by ATMs, other teller systems, and online banking. The software in the teller machine carries out a lot of the customer-related processing associated with a transaction.

Figure 17.9 shows a simplified version of the ATM system organization. The ATMs do not connect directly to the customer database, but rather to a teleprocessing (TP) monitor. A TP monitor is a middleware system that organizes communications with remote clients and serializes client transactions for processing by the database. This ensures that transactions are independent and do not interfere with one other. Using serial transactions means that the system can recover from faults without corrupting the system data.

While a fat-client model distributes processing more effectively than a thin-client model, system management is more complex if a special-purpose client, rather than a browser, is used. Application functionality is spread across many computers. When the application software has to be changed, this involves software reinstallation on every client computer. This can be a major cost if there are hundreds of clients in the system. Auto-update of the client software can reduce these costs but introduces its own problems if the client functionality is changed. The new functionality may mean that businesses have to change the ways they use the system.

The extensive use of mobile devices means that it is important to mimimize network traffic wherever possible. These devices now include powerful computers that can carry out local processing. As a consequence, the distinction between thin-client and fat-client architectures has become blurred. Apps can have inbuilt functionality that carries out local processing, and web pages may include Javascript components

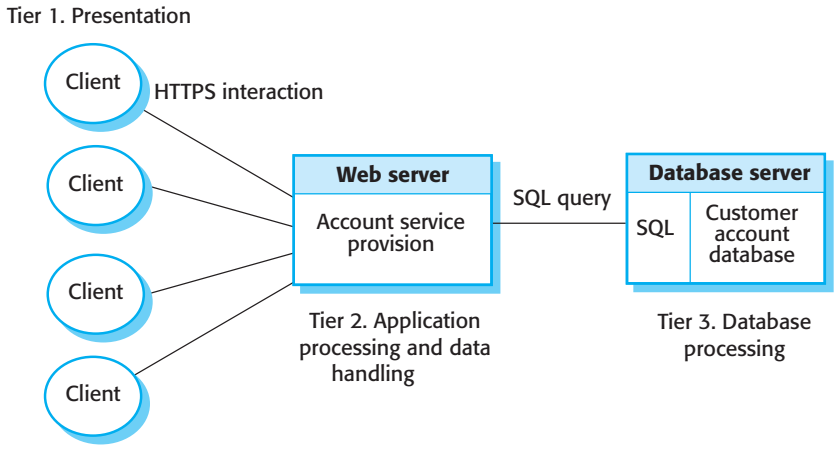


Figure 17.10 Three-tier architecture for an Internet banking system

that execute on the user's local computer. The update problem for apps remains an issue, but it has been addressed, to some extent, by automatically updating apps without explicit user intervention. Consequently, while it is sometimes helpful to use these models as a general basis for the architecture of a distributed system, in practice few web-based applications implement all processing on the remote server.

17.3.3 Multi-tier client–server architectures

The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server. This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. To avoid some of these problems, a "multi-tier client–server" architecture can be used. In this architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.

An Internet banking system (Figure 17.10) is an example of a multi-tier client–server architecture, where there are three tiers in the system. The bank's customer database (usually hosted on a mainframe computer as discussed above) provides database services. A web server provides data management services such as web page generation and some application services. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client. The user's own computer with an Internet browser is the client. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

Architecture	Applications
Two-tier client–server architecture with thin clients	Legacy system applications that are used when separating application processing and data handling is impractical. Clients may access these as services, as discussed in Section 17.4. Computationally intensive applications such as compilers with little or no requirements for data handling. Data-intensive applications (browsing and querying) with non-intensive application processing. Simple web browsing is the most common example of a situation where this architecture is used.
Two-tier client–server architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client. Applications where computationally intensive processing of data (e.g., data visualization) is required. Mobile applications where internet connectivity cannot be guaranteed. Local processing using cached information from the database is therefore possible.
Multi-tier client–server architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

Figure 17.11 Use of client–server architectural patterns

The three-tier client–server model can be extended to a multi-tier variant, where additional servers are added to the system. This may involve using a web server for data management and separate servers for application processing and database services. Multi-tier systems may also be used when applications need to access and use data from different databases. In this case, you may need to add an integration server to the system. The integration server collects the distributed data and presents it to the application server as if it were from a single database. As I discuss in the following section, distributed component architectures may be used to implement multi-tier client–server systems.

Multi-tier client–server systems that distribute application processing across several servers are more scalable than two-tier architectures. The tiers in the system can be independently managed, with additional servers added as the load increases. Processing may be distributed between the application logic and the data-handling servers, thus leading to more rapid response to client requests.

Designers of client–server architectures must take a number of factors into account when choosing the most appropriate distribution architecture. Situations in which the client–server architectures discussed here may be used are described in Figure 17.11.

17.3.4 Distributed component architectures

By organizing processing into layers, as shown in Figure 17.6, each layer of a system can be implemented as a separate logical server. This model works well for many types of application. However, it limits the flexibility of system designers in that they

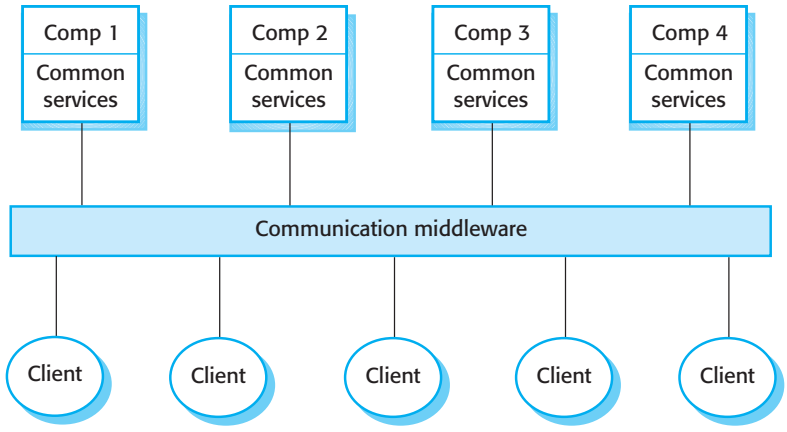


Figure 17.12 A distributed component architecture

have to decide what services should be included in each layer. In practice, however, it is not always clear whether a service is a data management service, an application service, or a database service. Designers must also plan for scalability and so provide some means for servers to be replicated as more clients are added to the system.

A more general approach to distributed system design is to design the system as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, can be implemented using a separate object or component. In a distributed component architecture (Figure 17.12), the system is organized as a set of interacting components as I discussed in Chapter 16. These components provide an interface to a set of services that they provide. Other components call on these services through middleware, using remote procedure or method calls.

Distributed component systems are reliant on middleware. This manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use. The CORBA standard (Orfali, Harkey, and Edwards 1997) defined middleware for distributed component systems, but CORBA implementations have never been widely adopted. Enterprises preferred to use proprietary software such as Enterprise Java Beans (EJB) or .NET.

Using a distributed component model for implementing distributed systems has a number of benefits:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, or a user interface layer.
2. It is a very open-system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.
3. The system is flexible and scalable. New objects or replicated objects can be added as the load on the system increases, without disrupting other parts of the system.

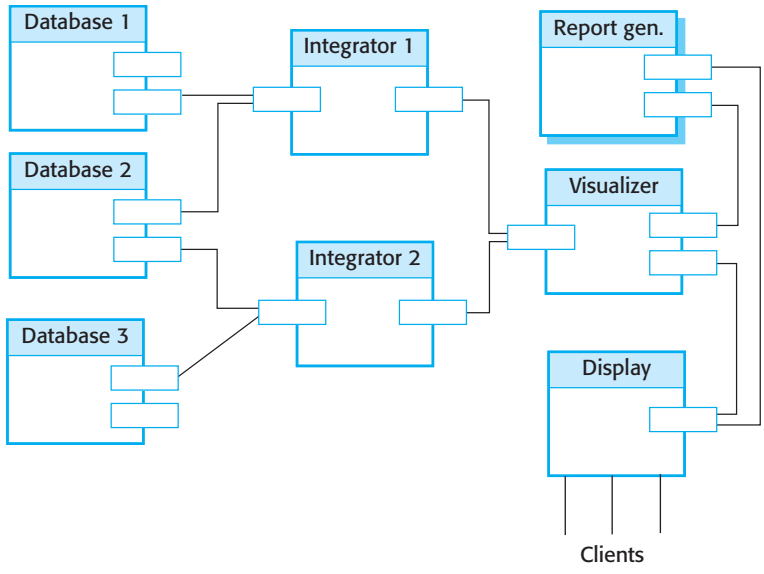


Figure 17.13 A distributed component architecture for a data-mining system

4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed component architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services. You then work out how to implement these services. For example, a retail application may have application components concerned with stock control, customer communications, goods ordering, and so on.

Data-mining systems are a good example of a type of system that can be implemented using a distributed component architecture. Data-mining systems look for relationships between the data that may be distributed across databases (Figure 17.13). These systems pull in information from several separate databases, carry out computationally intensive processing, and present easy-to-understand visualizations of the relationships that have been discovered.

An example of such a data-mining application might be a system for a retail business that sells food and books. Retail businesses maintain separate databases with detailed information about food products and books. They use a loyalty card system to keep track of customers' purchases, so there is a large database linking bar codes of products with customer information. The marketing department wants to find relationships between a customer's food and book purchases. For instance, a relatively high proportion of people who buy pizzas might also buy crime novels. With this knowledge, the business can specifically target customers who make specific food purchases with information about new novels when they are published.

In this example, each sales database can be encapsulated as a distributed component with an interface that provides read-only access to its data. Integrator components are each concerned with specific types of relationships, and they collect information from all of the databases to try to deduce the relationships. There might be an integrator component that is concerned with seasonal variations in goods sold, and another integrator that is concerned with relationships between different types of goods.

Visualizer components interact with integrator components to create a visualization or a report on the relationships that have been discovered. Because of the large volumes of data that are handled, visualizer components normally present their results graphically. Finally, a display component may be responsible for delivering the graphical models to clients for final presentation in their web browser.

A distributed component architecture rather than a layered architecture is appropriate for this type of application because you can add new databases to the system without major disruption. Each new database is simply accessed by adding another distributed component. The database access components provide a simplified interface that controls access to the data. The databases that are accessed may reside on different machines. The architecture also makes it easy to mine new types of relationships by adding new integrator objects.

Distributed component architectures suffer from two major disadvantages:

1. They are more complex to design than client–server systems. Multilayer client–server systems appear to be a fairly intuitive way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. The complexity of distributed component architectures increases the costs of implementation.
2. There are no universal standards for distributed component models or middleware. Rather, different vendors, such as Microsoft and Sun, developed different, incompatible middleware. This middleware is complex, and reliance on it significantly increases the complexity of distributed component systems.

As a result of these problems, distributed component architectures are being replaced by service-oriented systems (discussed in Chapter 18). However, distributed component systems have performance benefits over service-oriented systems. RPC communications are usually faster than the message-based interaction used in service-oriented systems. Distributed component architectures are therefore still used for high-throughput systems in which large numbers of transactions have to be processed quickly.

17.3.5 Peer-to-peer architectures

The client–server model of computing that I have discussed in previous sections of the chapter makes a clear distinction between servers, which are providers of services, and clients, which are receivers of services. This model usually leads to an uneven distribution of load on the system, where servers do more work than clients. This may lead to organizations spending a lot on server capacity while there is unused processing capacity on the hundreds or thousands of PCs and mobile devices used to access the system servers.

Peer-to-peer (p2p) systems (Oram 2001) are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself, and each node must run a copy of that application.

Peer-to-peer technologies have mostly been used for personal rather than business systems. The fact that there are no central servers means that these systems are harder to monitor; therefore, a higher level of communication privacy is possible.

For example, file-sharing systems based on the BitTorrent protocol are widely used to exchange files on users' PCs. Private instant messaging systems, such as ICQ and Jabber, provide direct communications between users without an intermediate server. Bitcoin is a peer-to-peer payments system using the Bitcoin electronic currency. Freenet is a decentralized database that has been designed to make it easier to publish information anonymously and to make it difficult for authorities to suppress this information.

Other p2p systems have been developed where privacy is not the principal requirement. Voice over IP (VoIP) phone services, such as Viber, rely on peer-to-peer communication between the parties involved in the phone call or conference. SETI@home is a long-running project that processes data from radio telescopes on home PCs in order to search for indications of extraterrestrial life. In these systems, the advantage of the p2p model is that a central server is not a processing bottleneck.

Peer-to-peer systems have also been used by businesses to harness the power in their PC networks (McDougall 2000). Intel and Boeing have both implemented p2p systems for computationally intensive applications. Such systems take advantage of unused processing capacity on local computers. Instead of buying expensive high-performance hardware, engineering computations can be run overnight when desktop computers are unused. Businesses also make extensive use of commercial p2p systems, such as messaging and VoIP systems.

In principle, every node in a p2p network could be aware of every other node. Nodes could connect to and exchange data directly with any other node in the network. In practice, this is impossible unless the network has only a few members. Consequently, nodes are usually organized into "localities," with some nodes acting as bridges to other node localities. Figure 17.14 shows this decentralized p2p architecture.

In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another. For example, assume that Figure 17.14 represents a decentralized, document-management system. A consortium of researchers uses this system to share documents. Each member of the consortium maintains his or her own document store. However, when a document is retrieved, the node retrieving that document also makes it available to other nodes.

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their "locality." These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore if n1 issues a search for a

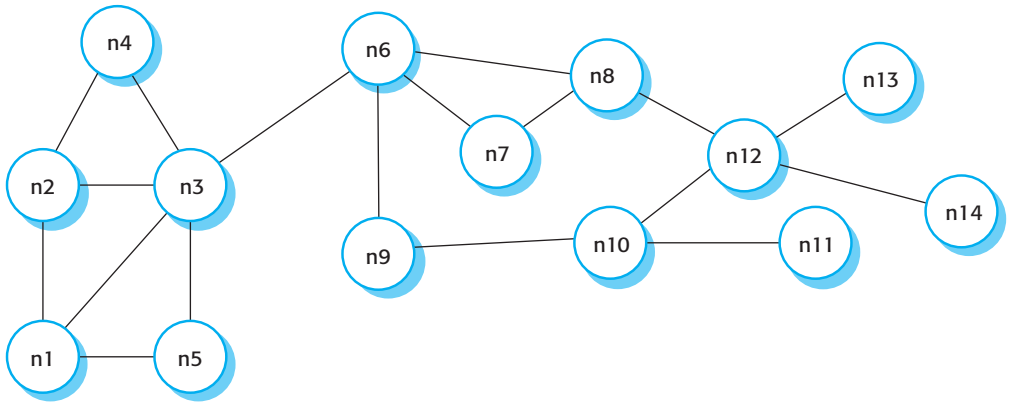


Figure 17.14
A decentralized p2p architecture

document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

This decentralized architecture has the advantage of being highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network. However, the disadvantages are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

An alternative p2p architectural model, which departs from a pure p2p architecture, is a semicentralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes. Figure 17.15 illustrates how this semicentralized architectural model differs from the completely decentralized model shown in Figure 17.14.

In a semicentralized architecture, the role of the server (sometimes called a superpeer) is to help establish contact between peers in the network or to coordinate the results of a computation. For example, if Figure 17.15 represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server becomes unnecessary. Therefore, nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be superpeers. Their role is to distribute work to other nodes and to collate and check the results of the computation.

The peer-to-peer architectural model may be the best model for a distributed system in two circumstances:

1. Where the system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations. For example, a peer-to-peer system that supports computational drug discovery distributes computations that look for potential cancer treatments by analyzing a huge number of molecules to see if they have the characteristics required to suppress the growth of cancers. Each molecule can be considered separately, so there is no need for the peers in the system to communicate.

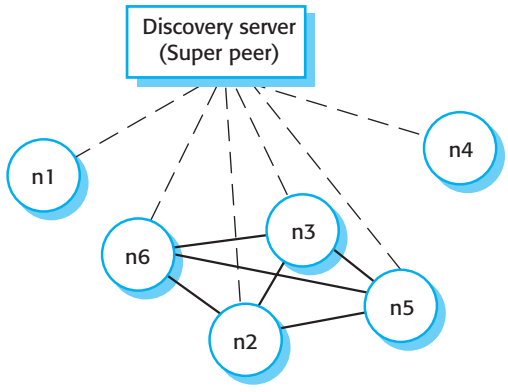


Figure 17.15 A semicentralized p2p architecture

2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed. Examples of such applications include file-sharing systems that allow peers to exchange local files such as music and video files, and phone systems that support voice and video communications between computers.

Peer-to-peer architectures allow for the efficient use of capacity across a network. However, security concerns are the principal reason why these systems have not become more widely used, especially in business (Wallach 2003). The lack of centralized management means that attackers can set up malicious nodes that deliver spam and malware to legitimate p2p system users. Peer-to-peer communications involve opening your computer to direct interactions with other peers and this means that these systems could potentially access any of your resources. To counter this possibility, you need to organize your system so that these resources are protected. If this is done incorrectly, then your system is insecure and vulnerable to external corruption.

17.4 Software as a service

In the previous sections, I discussed client–server models and how functionality may be distributed between the client and the server. To implement a client–server system, you may have to install a program or an app on the client computer, which communicates with the server, implements client-side functionality, and manages the user interface. For example, a mail client, such as Outlook or Mac Mail, provides mail management features on your own computer. This avoids the problem of server overload in thin-client systems, where all of the processing is carried out at the server.

The problems of server overload can be significantly reduced by using web technologies such as AJAX (Holdener, 2008) and HTML5 (Sarris 2013). These technologies support efficient management of web page presentation and local computation by executing scripts that are part of the web page. This means that a browser can be configured and used as client, with significant local processing. The application software can be

thought of as a remote service, which can be accessed from any device that can run a standard browser. Widely used examples of SaaS include web-based mail systems, such as Yahoo and Gmail, and office applications, such as Google Docs and Office 365.

This idea of software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. The key elements of SaaS are as follows:

1. Software is deployed on a server (or more commonly in the cloud) and is accessed through a web browser. It is not deployed on a local PC.
2. The software is owned and managed by a software provider rather than the organizations using the software.
3. Users may pay for the software according to how much use they make of it or through an annual or monthly subscription. Sometimes the software is free for anyone to use, but users must then agree to accept advertisements, which fund the software service.

The development of SaaS has accelerated over the past few years as cloud computing has become widely used. When a service is deployed in the cloud, the number of servers can quickly change to match the user demands for that service. There is no need for service providers to provision for peak loads; as a result, the costs for these providers have been dramatically reduced.

For software purchasers, the benefit of SaaS is that the costs of management of software are transferred to the provider. The provider is responsible for fixing bugs and installing software upgrades, dealing with changes to the operating system platform, and ensuring that hardware capacity can meet demand. Software license management costs are zero. If someone has several computers, there is no need to license software for all of these. If a software application is only used occasionally, the pay-per-use model may be cheaper than buying an application. The software may be accessed from mobile devices, such as smartphones, from anywhere in the world.

The main problem that inhibits the use of SaaS is data transfer with the remote service. Data transfer takes place at network speeds, and so transferring a large amount of data, such as video or high-quality images takes a lot of time. You may also have to pay the service provider according to the amount transferred. Other problems are lack of control over software evolution (the provider may change the software when it wishes) and problems with laws and regulations. Many countries have laws governing the storage, management, preservation, and accessibility of data, and moving data to a remote service may breach these laws.

The notion of software as a service and service-oriented architectures (SOA), discussed in Chapter 18, are related, but they are not the same:

1. Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions, for example, editing a document.

2. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These services may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result.

SaaS is a way of delivering application functionality to users, whereas SOA is an implementation technology for application systems. Systems that are implemented using SOA do not have to be accessed by users as web services. SaaS applications for business may be implemented using components rather than services. However, if SaaS is implemented using SOA, it becomes possible for applications to use service APIs to access the functionality of other applications. They can then be integrated into more complex systems. These systems are called mashups and are another approach to software reuse and rapid software development.

From a software development perspective, the process of service development has much in common with other types of software development. However, service construction is not usually driven by user requirements, but by the service provider's assumptions about what users need. Accordingly, the software needs to be able to evolve quickly after the provider gets feedback from users on their requirements. Agile development with incremental delivery is therefore an effective approach for software that is to be deployed as a service.

Some software that is implemented as a service, such as Google Docs for web users, offers a generic experience to all users. However, businesses may wish to have specific services that are tailored to their own requirements. If you are implementing SaaS for business, you may base your software service on a generic service that is tailored to the needs of each business customer. Three important factors have to be considered:

1. *Configurability* How do you configure the software for the specific requirements of each organization?
2. *Multi-tenancy* How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
3. *Scalability* How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

The notion of product-line architectures, discussed in Chapter 16, is one way of configuring software for users who have overlapping but not identical requirements. You start with a generic system and adapt it according to the specific requirements of each user.

This does not work for SaaS, however, for it would mean deploying a different copy of the service for each organization that uses the software. Rather, you need to design configurability into the system and provide a configuration interface that allows users to specify their preferences. You then use these preferences to adjust the behavior of the software dynamically as it is used. Configuration facilities may allow for:

1. *Branding*, where users from each organization are presented with an interface that reflects their own organization.

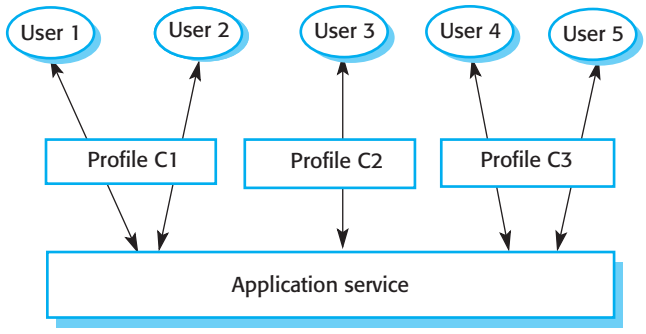


Figure 17.16 Configuration of a software system offered as a service

2. *Business rules and workflows*, where each organization defines its own rules that govern the use of the service and its data.
3. *Database extensions*, where each organization defines how the generic service data model is extended to meet its specific needs.
4. *Access control*, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Figure 17.16 illustrates this situation. This diagram shows five users of the application service, who work for three different customers of the service provider. Users interact with the service through a customer profile that defines the service configuration for their employer.

Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources. However, it must appear to users that they each have sole use of the system. Multi-tenancy involves designing the system so that there is an absolute separation between system functionality and system data. All operations must therefore be stateless so that they can be shared. Data must either be provided by the client or should be available in a storage system or database that can be accessed from any system instance.

A particular problem in multi-tenant systems is data management. The simplest way to provide data management is for all customers to have their own database, which they may use and configure as they wish. However, this requires the service provider to maintain many different database instances (one per customer) and to make these databases available on demand.

As an alternative, the service provider can use a single database, with different users being virtually isolated within that database. This is illustrated in Figure 17.17, where you can see that database entries also have a "tenant identifier" that links these entries to specific users. By using database views, you can extract the entries for each service customer and so present users from that customer with a virtual, personal database. This process can be extended to meet specific customer needs using the configuration features discussed above.

Scalability is the ability of the system to cope with increasing numbers of users without reducing the overall quality of service that is delivered to any user. Generally,

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figure 17.17 A multi-tenant database

when considering scalability in the context of SaaS, you are considering "scaling out" rather than "scaling up." Recall that scaling out means adding additional servers and so also increasing the number of transactions that can be processed in parallel. Scalability is a complex topic that I cannot cover in detail here, but following are some general guidelines for implementing scalable software:

1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction, a user may therefore interact with instances of the same service that are running on several different servers.
2. Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish.
3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
4. Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.
5. Use a cloud PaaS platform, such as Google App Engine (Sanderson 2012) or other PaaS platform for system implementation. These include mechanisms that will automatically scale out your system as the load increases.

The notion of software as a service is a major paradigm shift for distributed computing. We have already seen consumer software and professional applications, such as Photoshop, move to this model of delivery. Increasingly, businesses are replacing their own systems, such as CRM and inventory systems, with cloud-based SaaS systems from external providers such as Salesforce. Specialized software companies that implement business applications prefer to provide SaaS because it simplifies software update and management.

SaaS represents a new way to think about the engineering of enterprise systems. It has always been helpful to think of systems delivering services to users, but, until SaaS, this function has involved using different abstractions, such as objects, when implementing the system. Where there is a closer match between user and system abstractions, the resultant systems are easier to understand, maintain, and evolve.

KEY POINTS

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services (even if some parts of the system fail), and they enable resources to be shared.
- Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service, and failure management.
- Client–server systems are distributed systems in which the system is structured into layers, with the presentation layer implemented on a client computer. Servers provide data management, application, and database services.
- Client–server systems may have several tiers, with different layers of the system distributed to different computers.
- Architectural patterns for distributed systems include master–slave architectures, two-tier and multi-tier client–server architectures, distributed component architectures, and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow objects to be added to and removed from the system.
- Peer-to-peer architectures are decentralized architectures in which there are no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client–server systems, where the client is a web browser.

FURTHER READING

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Although this book does not have a lot of information on p2p architectures, it is an excellent introduction to p2p computing and discusses the organization and approach used in a number of p2p systems. (A. Oram (ed.), O'Reilly and Associates Inc., 2001).

"Turning Software into a Service." A good overview paper that discusses the principles of service-oriented computing. Unlike many papers on this topic, it does not conceal these principles behind a discussion of the standards involved. (M. Turner, D. Budgen, and P. Brereton, *IEEE Computer*, 36 (10), October 2003) http://dx.doi.org/10.1109/MC.2003.1236470

Distributed Systems, 5th ed. A comprehensive textbook that discusses all aspects of distributed systems design and implementation. It includes coverage of peer-to-peer systems and mobile systems. (G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Addison-Wesley, 2011).

Engineering Software as a Service: An Agile Approach Using Cloud Computing. This book accompanies the authors' online course on the topic. A good practical book that is aimed at people new to this type of development. (A. Fox and D. Patterson, Strawberry Canyon LLC, 2014) http://www.saasbook.info

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

17.1. What do you understand by "scalability"? Discuss the differences between scaling up and scaling out and explain when these different approaches to scalability may be used.

17.2. Explain why distributed software systems are more complex than centralized software systems, where all of the system functionality is implemented on a single computer.

17.3. Using an example of a remote procedure call, explain how middleware coordinates the interaction of computers in a distributed system.

17.4. What are the different logical layers in an application with a distributed client–server architecture?

17.5. You have been asked to design a secure system that requires strong authentication and authorization. The system must be designed so that communications between parts of the system cannot be intercepted and read by an attacker. Suggest the most appropriate client–server architecture for this system and, giving the reasons for your answer, propose how functionality should be distributed between the client and the server systems.

17.6. Your customer wants to develop a system for stock information where dealers can access information about companies and evaluate various investment scenarios using a simulation system. Each dealer uses this simulation in a different way, according to his or her experience and the type of stocks in question. Suggest a client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.

17.7. Using a distributed component approach, propose an architecture for a national theater booking system. Users can check seat availability and book seats at a group of theaters. The system should support ticket returns so that people may return their tickets for last-minute resale to other customers.

17.8. What is the fundamental problem with a two-tier client–server approach? Define how a multi-tier client–server approach overcomes this.

17.9. List the benefits that a distributed component model has when used for implementing distributed systems.

17.10. Your company wishes to move from using desktop applications to accessing the same functionality remotely as services. Identify three risks that might arise and suggest how these risks may be reduced.

REFERENCES

Bernstein, P. A. 1996. "Middleware: A Model for Distributed System Services." *Comm. ACM* 39 (2): 86–97. doi:10.1145/230798.230809.

Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. *Distributed Systems: Concepts and Design, 5th ed.* Harlow, UK: Addison-Wesley.

Holdener, A. T. (2008). *Ajax: The Definitive Guide. Sebastopol, CA.: O' Reilly & Associates*.

McDougall, P. 2000. "The Power of Peer-to-Peer." *Information Week* (August 28, 2000). http://www.informationweek.com/801/peer.htm

Oram, A. 2001. "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology." Sebastopol, CA: O'Reilly & Associates.

Orfali, R., D. Harkey, and J. Edwards. 1997. *Instant CORBA*. Chichester, UK: John Wiley & Sons.

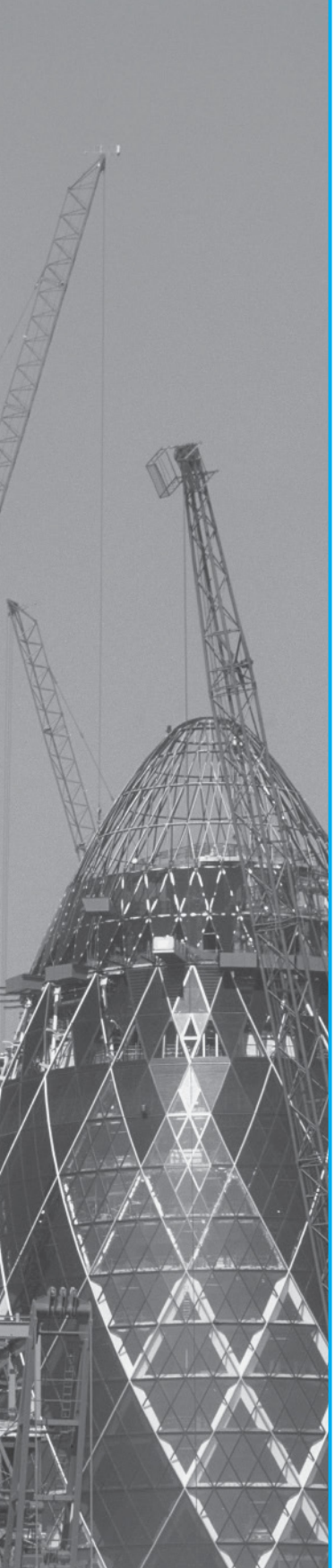
Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Sanderson, D. 2012. *Programming with Google App Engine*. Sebastopol, CA: O'Reilly Media Inc.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Tanenbaum, A. S., and M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms, 2nd ed.* Upper Saddle River, NJ: Prentice-Hall.

Wallach, D. S. 2003. "A Survey of Peer-to-Peer Security Issues." In *Software Security: Theories and Systems*, edited by M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, 42–57. Heidelberg: Springer-Verlag. doi:10.1007/3-540-36532-X_4.



18

Service-oriented software engineering

Objectives

The objective of this chapter is to introduce service-oriented software engineering as a way of building distributed applications using web services. When you have read this chapter, you will:

- understand the basic notions of a web service, web service standards, and service-oriented architecture;
- understand the idea of RESTful services and the important differences between RESTful and SOAP-based services;
- understand the service engineering process that is intended to produce reusable web services;
- understand how workflow-based service composition can be used to create service-oriented software that supports business processes.

Contents

18.1 Service-oriented architecture
18.2 RESTful services
18.3 Service engineering
18.4 Service composition

The development of the Web in the 1990s revolutionized organizational information exchange. Client computers could gain access to information on remote servers outside their own organizations. However, access was solely through a web browser, and direct access to the information by other programs was not practical. This meant that opportunistic connections between servers, where, for example, a program could query a number of catalogs from different suppliers, were not possible.

To get around this problem, web services were developed that allowed programs to access and update resources available on the web. Using a web service, organizations that wish to make their information accessible to other programs can do so by defining and publishing a programmatic web service interface. This interface defines the data available and how it can be accessed and used.

More generally, a web service is a standard representation for some computational or information resource that can be used by other programs. These may be information resources, such as a parts catalog, computer resources, such as a specialized processor, or storage resources. For example, an archive service could be implemented that permanently and reliably stores organizational data that, by law, has to be maintained for many years.

A web service is an instance of a more general notion of a service, which Lovelock et al. (Lovelock et al., 1996) defined as:

> *an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.*[†]

Services are a natural development of software components where the component model is, in essence, a set of standards associated with web services. A web service can therefore be defined as:

> *A loosely coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.*

A critical distinction between a service and a software component, as defined in component-based software engineering, is that services should be independent and loosely coupled. That is, they should always operate in the same way, irrespective of their execution environment. They should not rely on external components that may have different functional and non-functional behavior. Therefore, web services do not have a "requires" interface that, in CBSE, defines the other system components that must be present. A web service interface is simply a "provides" interface that defines the service functionality and parameters.

Service-oriented systems are a way of developing distributed systems where the system components are stand-alone services, executing on geographically distributed computers. Services are platform and implementation-language independent. Software systems can be constructed by composing local services and external services from different providers, with seamless interaction between the services in the system.

[†]Lovelock, C., Vandermerwe, S. and Lewis, B. (1996). Services Marketing. Englewood Cliffs, NJ: Prentice Hall.

As I discussed in Chapter 17, the ideas of "software as a service" and "service-oriented systems" are not the same thing. Software as a service means offering software functionality to users remotely over the web, rather than through applications installed on a user's computer. Service-oriented systems are systems that are implemented using reusable service components and that are accessed by other programs, rather than directly by users. Software that is offered as a service may be implemented using a service-oriented system. However, you don't have to implement software in this way to offer it as a user service.

Adopting a service-oriented approach to software engineering has a number of important benefits:

1. Services can be offered by any service provider inside or outside of an organization. Assuming these services conform to certain standards, organizations can create applications by integrating services from a range of providers. For example, a manufacturing company can link directly to services provided by its suppliers.
2. The service provider makes information about the service public so that any authorized user can use the service. The service provider and the service user do not need to negotiate about what the service does before it can be incorporated in an application program.
3. Applications can delay the binding of services until they are deployed or until execution. Therefore, an application using a stock price service (say) could, in principle, dynamically change service providers while the system was executing. This means that applications can be reactive and adapt their operation to cope with changes to their execution environment.
4. Opportunistic construction of new services is possible. A service provider may recognize new services that can be created by linking existing services in innovative ways.
5. Service users can pay for services according to their use rather than their provision. Therefore, instead of buying an expensive component that is rarely used, the application writer can use an external service that will be paid for only when required.
6. Applications can be made smaller, which is particularly important for mobile devices with limited processing and memory capabilities. Some computationally intensive processing and exception handling can be offloaded to external services.

Service-oriented systems have loosely coupled architectures where service bindings may change during system execution. A different, but equivalent, version of the service may therefore be executed at different times. Some systems will be solely built using web services, and others will mix web services with locally developed components. To illustrate how applications that use a mixture of services and components may be organized, consider the following scenario:

An in-car information system provides drivers with information on weather, road traffic conditions, local information and so forth. This is linked to the car

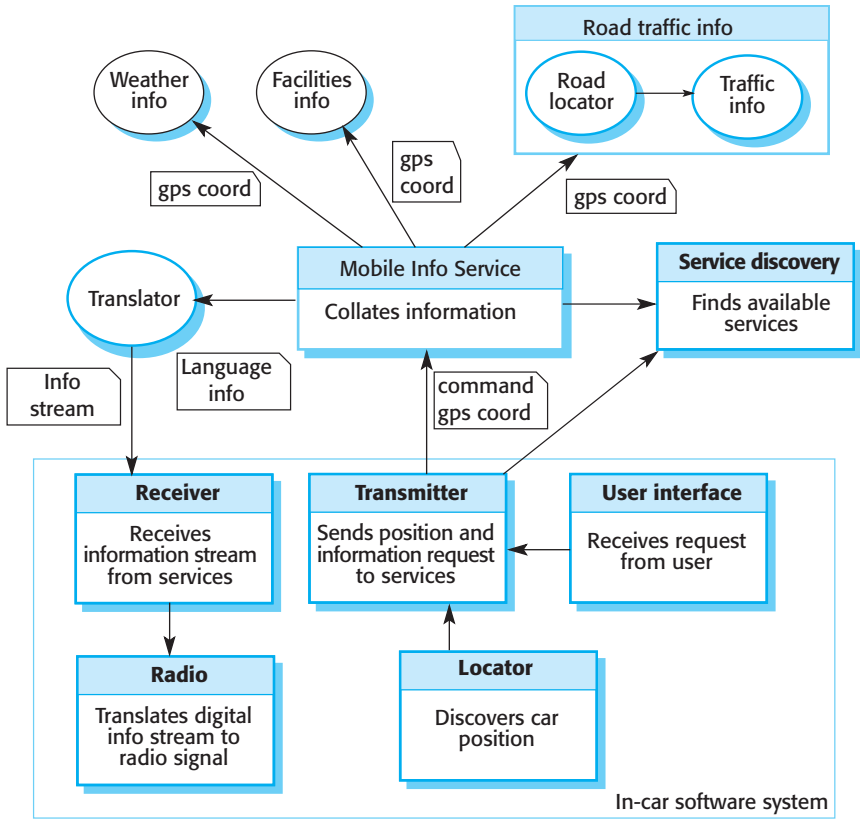


Figure 18.1 A service-based, in-car information system

radio so that information is delivered as a signal on a specific radio channel. The car is equipped with GPS receiver to discover its position, and, based on that position, the system accesses a range of information services. Information may then be delivered in the driver's specified language.

Figure 18.1 illustrates a possible organization for such a system. The in-car software includes five modules. These handle communications with the driver, with a GPS receiver that reports the car's position, and with the car radio. The Transmitter and Receiver modules handle all communications with external services.

The car communicates with an external mobile information service that aggregates information from a range of other services, providing information on weather, traffic, and local facilities. Different providers in different places offer these services, and the in-car system accesses an external discovery service to find the services available in the local area. The mobile information service also uses the discovery service to bind to the appropriate weather, traffic, and facilities services. The aggregated information is then sent to the car through a service that translates that information into the driver's preferred language.

This example illustrates one of the key advantages of the service-oriented approach. When the system is programmed or deployed, you don't have to decide what service

provider should be used or what specific services should be accessed. As the car moves around, the in-car software uses the service discovery service to find the most useful local information service. Because of the use of a translation service, it can move across borders and make local information available to people who don't speak the local language.

I think that the service-oriented approach to software engineering is as important a development as object-oriented software engineering. Service-oriented systems are essential to the cloud and mobile systems. Newcomer and Lomow (Newcomer and Lomow 2005), in their book on SOA, summarize the potential of service-oriented approaches, which is now being realized:

> *Driven by the convergence of key technologies and the universal adoption of Web services, the service-oriented enterprise promises to significantly improve corporate agility, speed time-to-market for new products and services, reduce IT costs and improve operational efficiency.*[†]

Building applications based on services allows companies and other organizations to cooperate and make use of each other's business functions. Thus, systems that involve extensive information exchange across company boundaries, such as supply chain systems where one company orders goods from another, can easily be automated. Service-based applications may be constructed by linking services from various providers using either a standard programming language or a specialized workflow language, as discussed in Section 18.4.

Initial work on service provision and implementation was heavily influenced by the failure of the software industry to agree on component standards. It was therefore standards-driven, with all of the main industrial companies involved in standards development. This led to a whole range of standards (WS* standards) and the notion of service-oriented architectures. These were proposed as architectures for service-based systems, with all service communication being standards-based. However, the standards proposed were complex and had a significant execution overhead. This problem has led many companies to adopt an alternative architectural approach, based on so-called RESTful services. A RESTful approach is a simpler approach than a service-oriented architecture, but it is less suited to services that offer complex functionality. I discuss both of these architectural approaches in this chapter.

18.1 Service-oriented architecture

Service-oriented architecture (SOA) is an architectural style based on the idea that executable services can be included in applications. Services have well-defined, published interfaces, and applications can choose whether or not these are appropriate. An important idea underlying SOA is that the same service may be available from different providers and that applications could make a runtime decision of which service provider to use.

[†]Newcomer, E. and Lomow, G. (2005). Understanding SOA with Web Services. Boston: Addison-Wesley.

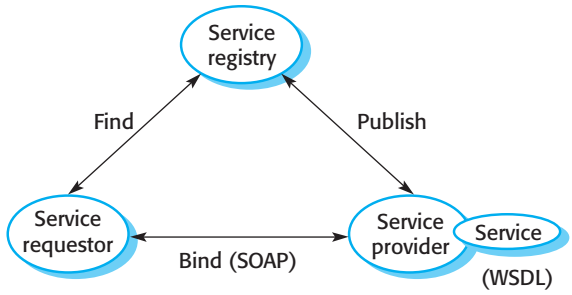


Figure 18.2 Service-oriented architecture

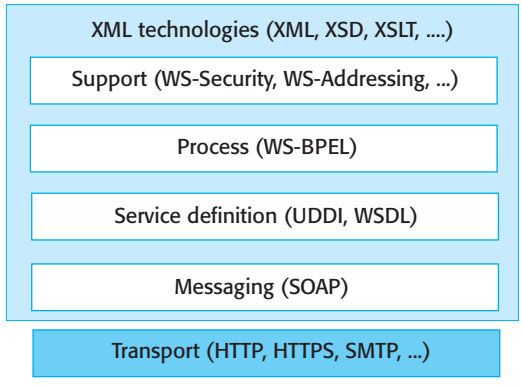


Figure 18.3 Web service standards

Figure 18.2 illustrates the structure of a service-oriented architecture. Service providers design and implement services and specify the interface to these services. They also publish information about these services in an accessible registry. Service requestors (sometimes called service clients) who wish to make use of a service discover the specification of that service and locate the service provider. They can then bind their application to that specific service and communicate with it, using standard service protocols.

The development and use of internationally agreed standards is fundamental to SOA. As a result, service-oriented architectures have not suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology. Figure 18.3 shows the stack of key standards that have been established to support web services.

Web service protocols cover all aspects of service-oriented architectures, from the basic mechanisms for service information exchange (SOAP) to programming language standards (WS-BPEL). These standards are all based on XML, a human and machine-readable notation that allows the definition of structured data where text is tagged with a meaningful identifier. XML has a range of supporting technologies, such as XSD for schema definition, which are used to extend and manipulate XML descriptions. Erl (Erl 2004) provides a good summary of XML technologies and their role in web services.

Briefly, the fundamental standards for service-oriented architectures are:

1. *SOAP* This is a message interchange standard that supports communication between services. It defines the essential and optional components of messages

passed between services. Services in a service-oriented architecture are sometimes called SOAP-based services.

2. *WSDL* The Web Service Description Language (WSDL) is a standard for service interface definition. It sets out how the service operations (operation names, parameters, and their types) and service bindings should be defined.

3. *WS-BPEL* This is a standard for a workflow language that is used to define process programs involving several different services. I explain what process programs are in Section 18.3.

The UDDI (Universal Description, Discovery, and Integration) discovery standard defines the components of a service specification intended to help potential users discover the existence of a service. This standard was meant to allow companies to set up registries, with UDDI descriptions defining the services they offered. Some companies set up UDDI registries in the early years of the 21st century, but users preferred standard search engines to find services. All public UDDI registries have now closed.

The principal SOA standards are supported by a range of supporting standards that focus on more specialized aspects of SOA. There are many supporting standards because they are intended to support SOA in different types of enterprise application. Some examples of these standards include:

1. *WS-Reliable Messaging,* a standard for message exchange that ensures messages will be delivered once and once only.

2. *WS-Security,* a set of standards supporting web service security, including standards that specify the definition of security policies and standards that cover the use of digital signatures.

3. *WS-Addressing,* which defines how address information should be represented in a SOAP message.

4. *WS-Transactions,* which defines how transactions across distributed services should be coordinated.

Web service standards are a huge topic, and I don't have space to discuss them in detail here. I recommend Erl's books (Erl 2004, 2005) for an overview of these standards. Their detailed descriptions are also available as public documents on the Web (W3C 2013).

18.1.1 Service components in an SOA

Message exchange, as I explained in Section 17.1, is an important mechanism for coordinating actions in a distributed computing system. Services in a SOA communicate by exchanging messages, expressed in XML, and these messages are distributed using standard Internet transport protocols such as HTTP and TCP/IP.

A service defines what it needs from another service by setting out its requirements in a message, which is sent to that service. The receiving service parses the

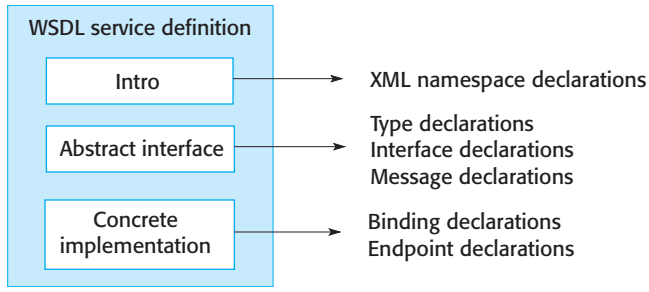


Figure 18.4
Organization of a WSDL specification

message, carries out the computation, and, upon completion, sends a reply, as a message, to the requesting service. This service then parses the reply to extract the required information. Unlike software components, services do not use remote procedure or method calls to access functionality associated with other services.

When you intend to use a web service, you need to know where the service is located (its Uniform Resource Identifier—URI) and the details of its interface. These details are provided in a service description that is written in an XML-based language called WSDL (Web Service Description Language). The WSDL specification defines three aspects of a Web service: what the service does, how it communicates, and where to find it:

1. The "what" part of a WSDL document, called an interface, specifies what operations the service supports and defines the format of the messages sent and received by the service.
2. The "how" part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a Web service.
3. The "where" part of a WSDL document describes the location of a specific Web service implementation (its endpoint).

The WSDL conceptual model (Figure 18.4) shows the elements of a service description. Each element is expressed in XML and may be provided in separate files. These elements are:

1. An introductory part that usually defines the XML namespaces used and that may include a documentation section providing additional information about the service.
2. An optional description of the types used in the messages exchanged by the service.
3. A description of the service interface, that is, the operations that the service provides for other services or users.
4. A description of the input and output messages processed by the service.
5. A description of the binding used by the service, that is, the messaging protocol that will be used to send and receive messages. The default is SOAP, but other

```
Define some of the types used. Assume that the namespace prefixes 'ws' refers to
the namespace URI for XML schemas and the namespace prefix associated with this
definition is weathns.
<types>
   <xs: schema targetNameSpace = "http://.../weathns"
        xmlns: weathns = "http://.../weathns" >
        <xs:element name = "PlaceAndDate" type = "pdrec" />
        <xs:element name = "MaxMinTemp" type = "mmtrec" />
        <xs:element name = "InDataFault" type = "errmess" />

        <xs:complexType name = "pdrec"
        <xs:sequence>
             <xs:element name = "town" type = "xs:string"/>
             <xs:element name = "country" type = "xs:string"/>
             <xs:element name = "day" type = "xs:date" />
        </xs:complexType>

        Definitions of MaxMinType and InDataFault here

   </schema>

</types>

Now define the interface and its operations. In this case, there is only a single
operation to return maximum and minimum temperatures

<interface name = "weatherInfo" >
   <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
   <input messageLabel = "In" element = "weathns: PlaceAndDate" />
   <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
   <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

Figure 18.5 Part of a WSDL description for a web service

bindings may also be specified. The binding sets out how the input and output messages associated with the service should be packaged into a message, and specifies the communication protocols used. The binding may also specify how supporting information, such as security credentials or transaction identifiers, is included in messages to the service.

6. An endpoint specification that is the physical location of the service, expressed as a URI—the address of a resource that can be accessed over the Internet.

Figure 18.5 shows part of the interface for a simple service that, given a date and a place, specified as a town within a country, returns the maximum and minimum temperature recorded in that place on that date. The input message also specifies whether these temperatures are to be returned in degrees Celsius or degrees Fahrenheit.

XML-based service descriptions include definitions of XML namespaces. A namespace identifier may precede any identifier used in the XML description, making it possible to distinguish between identifiers with the same name that have been defined in different parts of an XML description. You don't have to understand the details of namespaces to

understand the examples here. You only need to know that names may be prefixed with a namespace identifier and that the namespace:name pair should be unique.

In Figure 18.5, the first part of the description shows part of the element and type definition that is used in the service specification. This defines the elements PlaceAndDate, MaxMinTemp, and InDataFault. I have only included the specification of PlaceAndDate, which you can think of as a record with three fields—town, country and date. A similar approach would be used to define MaxMinTemp and InDataFault.

The second part of the description shows how the service interface is defined. In this example, the service weatherInfo has a single operation, although there are no restrictions on the number of operations that may be defined. The weatherInfo operation has an associated in-out pattern meaning that it takes one input message and generates one output message. The WSDL 2.0 specification allows for a number of message exchange patterns such as in-only, in-out, out-only, in-optional-out, and out-in. The input and output messages, which refer to the definitions made earlier in the types section, are then defined.

A service interface that is defined in WSDL is simply a description of the service signature, that is, the operations and their parameters. It does not include any information about the semantics of the service or its non-functional characteristics, such as performance and dependability. If you plan to use the service, you have to work out what the service actually does and the meaning of the input and output messages. You have to experiment to discover the service's performance and dependability. While meaningful names and documentation help with understanding the service functionality, it is still possible to misunderstand what the service actually does.

XML-based service descriptions are long, detailed, and tedious to read. WSDL specifications are not normally written by hand, and most of the information in a specification is automatically generated.

18.2 RESTful services

The initial developments of web services and service-oriented software engineering were standards-based, with XML-based messages exchanged between services. This is a general approach that allows for the development of complex services, dynamic service binding, and control over quality of service and service dependability. However, as services were developed, it emerged that most of these were single-function services with relatively simple input and output interfaces. Service users were not really interested in dynamic binding and the use of multiple service providers. They rarely use web service standards for quality of service, reliability, and so forth.

The problem is that web services standards are "heavyweight" standards that are sometimes overly general and inefficient. Implementing these standards requires a considerable amount of processing to create, transmit, and interpret the associated XML messages. This slows down communications between services, and, for high-throughput systems, additional hardware may be required to deliver the quality of service required.

In response to this situation, an alternative "lightweight" approach to web service architecture has been developed. This approach is based on the REST architectural

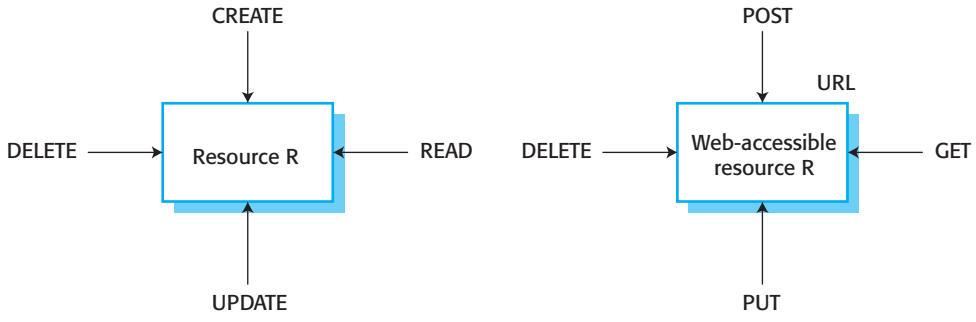


Figure 18.6 Resources and actions

(a) General resource actions

(b) Web resources

style, where REST stands for Representational State Transfer (Fielding 2000). REST is an architectural style based on transferring representations of resources from a server to a client. It is the style that underlies the web as a whole and has been used as a much simpler method than SOAP/WSDL for implementing web service interfaces.

The fundamental element in a RESTful architecture is a resource. Essentially, a resource is simply a data element such as a catalog and a medical record, or a document, such as this book chapter. In general, resources may have multiple representations; that is, they can exist in different formats. For example, this book chapter has three representations. These are a MS Word representation, which is used for editing, a PDF representation, which is used for web display, and a InDesign representation, which is used for publishing. The underlying logical resource made up of text and images is the same in all of these representations.

In a RESTful architecture, everything is represented as a resource. Resources have a unique identifier, which is their URL. Resources are a bit like objects, with four fundamental polymorphic operations associated with them, as shown in Figure 18.6(a):

1. Create—bring the resource into existence.
2. Read—return a representation of the resource.
3. Update—change the value of the resource.
4. Delete—make the resource inaccessible.

The Web is an example of a system that has a RESTful architecture. Web pages are resources, and the unique identifier of a web page is its URL.

The web protocols http and https are based on four actions, namely, POST, GET, PUT, and DELETE. These map onto the basic resource operations, as I have shown in Figure 18.6(b):

1. POST is used to create a resource. It has associated data that defines the resource.
2. GET is used to read the value of a resource and return that to the requestor in the specified representation, such as XHTML, that can be rendered in a web browser.

3. PUT is used to update the value of a resource.
4. DELETE is used to delete the resource.

All services, in some way, operate on data. For example, the service described in Section 18.2 that returns the maximum and minimum temperatures for a location on a given data uses a weather information database. SOAP-based services execute actions on this database to return particular values from it. RESTful services (Richardson and Ruby 2007) access the data directly.

When a RESTful approach is used, the data is exposed and is accessed using its URL. RESTful services use http or https protocols, with the only allowed actions being POST, GET, PUT, and DELETE. Therefore, the weather data for each place in the database might be accessed using URLs such as:

http://weather-info-example.net/temperatures/boston
http://weather-info-example.net/temperatures/edinburgh

This would invoke the GET operation and return a list of maximum and minimum temperatures. To request the temperatures for a specific date, a URL query can be used:

http://weather-info-example.net/temperatures/edinburgh?date=20140226

URL queries can also be used to disambiguate the request, given that there may be several places in the world with the same name:

http://weather-info-example.net/temperatures/boston?date=20140226&country=USA&state="Mass"

An important difference between RESTful services and SOAP-based services is that RESTful services are not exclusively XML-based. So, when a resource is requested, created, or changed, the representation may be specified. This is important for RESTful services because representations such as JSON (Javascript Object Notation), as well as XML, may be used. These can be processed more efficiently than XML-based notations, thus reducing the overhead involved in a service call. Therefore, the above request for maximum and minimum temperatures for Boston may return the following information:

```
{
"place": "Boston",
"country  "USA",
"state": "Mass",
"date": "26 Feb 2014",
"units": "Fahrenheit",
"max temp": 41,
"min temp": 29
}
```

The response to a GET request in a RESTful service may include URLs. Therefore, if the response to a request is a set of resources, then the URL of each of

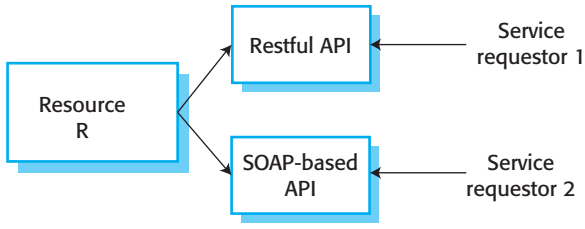


Figure 18.7 RESTful and SOAP-based APIs

these services may be included. The requesting service may then process the requests in its own way. Therefore, a request for weather information given a place name that is not unique may return the URLs of all of the places that match the request. For example:

http://weather-info-example.net/temperatures/edinburgh-scotland
http://weather-info-example.net/temperatures/edinburgh-australia
http://weather-info-example.net/temperatures/edinburgh-maryland

A fundamental design principle for RESTful services is that they should be stateless. That is, in an interaction session, the resource itself should not include any state information, such as the time of the last request. Instead, all necessary state information should be returned to the requestor. If state information is required in later requests, it should be returned to the server by the requestor.

RESTful services have become more widely used over the past few years because of the widespread use of mobile devices. These devices have limited processing capabilities, so the lower overhead of RESTful services allows better system performance. They are also easy to use with existing websites—implementing a RESTful API for a website is usually fairly straightforward.

However, there are problems with the RESTful approach:

1. When a service has a complex interface and is not a simple resource, it can be difficult to design a set of RESTful services to represent this interface.
2. There are no standards for RESTful interface description, so service users must rely on informal documentation to understand the interface.
3. When you use RESTful services, you have to implement your own infrastructure for monitoring and managing the quality of service and the service reliability. SOAP-based services have additional infrastructure support standards such as WS-Reliability and WS-Transactions.

Pautasso et al. (Pautasso, Zimmermann, and Leymann 2008) discuss when RESTful and SOAP-based should be used. However, it is often possible to provide both SOAP-based and RESTful interfaces to the same service or resource (Figure 18.7). This dual approach is now common for cloud services from providers such as Microsoft, Google, and Amazon. Service clients can then choose the service access method that is best suited to their applications.

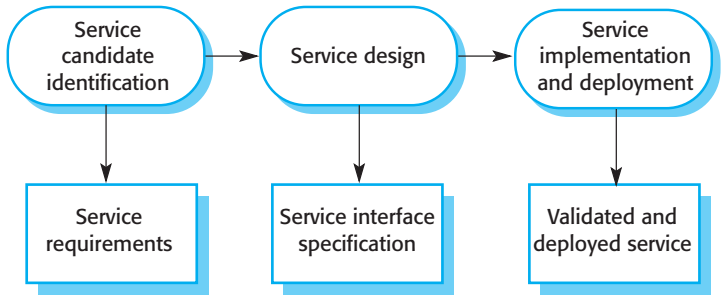


Figure 18.8 The service engineering process

18.3 Service engineering

Service engineering is the process of developing services for reuse in service-oriented applications. It has much in common with component engineering. Service engineers have to ensure that the service represents a reusable abstraction that could be useful in different systems. They must design and develop generally useful functionality associated with that abstraction and ensure that the service is robust and reliable. They have to document the service so that it can be discovered and understood by potential users.

As shown in Figure 18.8, there are three logical stages in the service engineering process:

1. *Service candidate identification,* where you identify possible services that might be implemented and define the service requirements.
2. *Service design,* where you design the logical service interface and its implementation interfaces (SOAP-based and/or RESTful).
3. *Service implementation and deployment,* where you implement and test the service and make it available for use.

As I discussed in Chapter 16, the development of a reusable component may start with an existing component that has already been implemented and used in an application. The same is true for services—the starting point for this process will often be an existing service or a component that is to be converted to a service. In this situation, the design process involves generalizing the existing component so that application-specific features are removed. Implementation means adapting the component by adding service interfaces and implementing the required generalizations.

18.3.1 Service candidate identification

The basic idea of service-oriented computing is that services should support business processes. As every organization has a wide range of processes, many possible services may be implemented. Service candidate identification therefore involves

	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document translator Web form to XML converter	Expenses form Student application form	

Figure 18.9 Service classification

understanding and analyzing the organization's business processes to decide which reusable services could be implemented to support these processes.

Erl (Erl 2005) suggests that there are three fundamental types of service:

1. *Utility services.* These services implement some general functionality that may be used by different business processes. An example of a utility service is a currency conversion service that can be accessed to compute the conversion of one currency (e.g., dollars) to another (e.g., euros).
2. *Business services.* These services are associated with a specific business function. An example of a business function in a university would be the registration of students for a course.
3. *Coordination or process services.* These services support a more general business process that usually involves different actors and activities. An example of a coordination service in a company is an ordering service that allows orders to be placed with suppliers, goods accepted, and payments made.

Erl also suggests that services can be thought of as task-oriented or entity-oriented. Task-oriented services are associated with some activity, whereas entity-oriented services are associated with a system resource. The resource is a business entity such as a job application form. Figure 18.9 shows examples of services that are task-oriented or entity-oriented. Utility or business services may be entity-oriented or task-oriented. Coordination services are always task-oriented.

Your goal in service candidate identification should be to identify services that are logically coherent, independent, and reusable. Erl's classification is helpful in this respect, as it suggests how to discover reusable services by looking at business entities as resources and business activities. However, identifying service candidates is sometimes difficult because you have to envisage how the services could be used. You have to think of possible candidates and then ask a series of questions about them to see if they are likely to be useful services. Possible questions that you might ask to identify potentially reusable services are:

1. For an entity-oriented service, is the service associated with a single logical resource that is used in different business processes? What operations are normally performed on that entity that must be supported? Do these fit with the RESTful service operations PUT, CREATE, POST, and DELETE?
2. For a task-oriented service, is the task one that is carried out by different people in the organization? Will they be willing to accept the inevitable standardization

that occurs when a single support service is provided? Can this fit into the RESTful model, or should it be redesigned as an entity-oriented service.

3. Is the service independent? That is, to what extent does it rely on the availability of other services?

4. Does the service have to maintain state? If state information is required, this must either be maintained in a database or passed as a parameter to the service. Using a database affects service reusability as there is a dependency between the service and the required database. In general, services where the state is passed to the service are easier to reuse, as no database binding is required.

5. Might this service be used by external clients? For example, an entity-oriented service associated with a catalog could be made available to both internal and external users.

6. Are different users of the service likely to have different non-functional requirements? If they do, then more than one version of a service should perhaps be implemented.

The answers to these questions help you select and refine abstractions that can be implemented as services. However, there is no formulaic way of deciding which are the best services. You need to use your experience and business knowledge to decide on what are the most appropriate services.

The output of the service selection process is a set of identified services and associated requirements for these services. The functional service requirements should define what the service should do. The non-functional requirements should define the security, performance, and availability requirements of the service.

To help you understand the process of service candidate identification and implementation, consider the following example:

A company, which sells computer equipment, has arranged special prices for approved configurations for some large customers. To facilitate automated ordering, the company wishes to produce a catalog service that will allow customers to select the equipment that they need. Unlike a consumer catalog, orders are not placed directly through a catalog interface. Instead, goods are ordered through the web-based procurement system of each company that accesses the catalog as a web service. The reason for this is that large companies usually have their own budgeting and approval procedures for orders that must be followed when an order is placed.

The catalog service is an example of an entity-oriented service, where the underlying resource is the catalog. The functional catalog service requirements are as follows:

1. A specific version of the catalog shall be provided for each user company. This shall include the approved configurations and equipment that may be ordered by

employees of the customer company and the equipment prices that have been agreed to with that company.

2. The catalog shall allow a customer employee to download a version of the catalog for offline browsing.
3. The catalog shall allow users to compare the specifications and prices of up to six catalog items.
4. The catalog shall provide browsing and search facilities for users.
5. Users of the catalog shall be able to discover the predicted delivery date for a given number of specific catalog items.
6. Users of the catalog shall be able to place "virtual orders" where the items required will be reserved for them for 48 hours. Virtual orders must be confirmed by a real order placed by a procurement system. The real order must be received within 48 hours of the virtual order.

In addition to these functional requirements, the catalog has a number of non-functional requirements:

1. Access to the catalog service shall be restricted to employees of accredited organizations.
2. The prices and configurations offered to each customer shall be confidential, and access to these shall only be provided to employees of that customer.
3. The catalog shall be available without disruption of service from 0700 GMT to 1100 GMT.
4. The catalog service shall be able to process up to 100 requests per second peak load.

There is no non-functional requirement related to the response time of the catalog service. This depends on the size of the catalog and the expected number of simultaneous users. As this is not a time-critical service, there is no need to specify the required performance at this stage.

18.3.2 Service interface design

Once you have identified candidate services, the next stage in the service engineering process is to design the service interfaces. This involves defining the operations associated with the service and their parameters. If SOAP-based services are used, you have to design the input and output messages. If RESTful services are used, you have to think about the resources required and how the standard operations should be used to implement the service operations.

The starting point for service interface design is abstract interface design. where you identify the entities and the operations associated with the service, their inputs and

Operation	Description
MakeCatalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Lookup	Displays all of the data associated with a specified catalog item.
Search	Takes a logical expression and searches the catalog according to that expression. It displays a list of all items that match the search expression.
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
CheckDelivery	Returns the predicted delivery date for an item if ordered that day.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

Figure 18.10 Catalog operations

outputs, and the exceptions associated with these operations. You then need to think about how this abstract interface is realized as SOAP-based or RESTful services.

If you choose a SOAP-based approach, you have to design the structure of the XML messages that are sent and received by the service. The operations and messages are the basis of an interface description written in WSDL. If you choose a RESTful approach, you have to design how the service operations map onto the RESTful operations.

Abstract interface design starts with the service requirements and defines the operation names and parameters. At this stage, you should also define the exceptions that may arise when a service operation is invoked. Figure 18.10 shows the catalog operations that implement the requirements. There is no need for these to be specified in detail; you add detail at the next stage of the design process.

Once you have established an informal description of what the service should do, the next stage is to add more detail of the service inputs and outputs. I have shown this for the catalog service in Figure 18.11, which extends the functional description in Figure 18.10.

Defining exceptions and how these exceptions can be communicated to service users is particularly important. Service engineers do not know how their services will be used. It is usually unwise to make assumptions that service users will have completely understood the service specification. Input messages may be incorrect, so you should define exceptions that report incorrect inputs to the service client. It is generally good practice in reusable component development to leave all exception handling to the user of the component. Service developers should not impose their views on how exceptions should be handled.

In some cases, a textual description of the operations and their inputs and outputs is all that is required. The detailed realization of the service is left as an implementation decision. Sometimes, however, you need to have a more detailed design, and a detailed interface description can be specified in a graphical notation such as the UML or in a readable description format such as JSON. Figure 18.12, which describes the inputs and outputs for the getDelivery operation, shows how you can use the UML to describe the interface in detail.

Operation	Inputs	Outputs	Exceptions
MakeCatalog	*mcIn* Company id PDF-flag	*mcOut* URL of the catalog for that company	*mcFault* Invalid company id
Lookup	*lookIn* Catalog URL Catalog number	*lookOut* URL of page with the item information	*lookFault* Invalid catalog number
Search	*searchIn* Catalog URL Search string	*searchOut* URL of web page with search results	*searchFault* Badly formed search string
Compare	*compIn* Catalog URL Entry attribute (up to 6) Catalog number (up to 4)	*compOut* URL of page showing comparison table	*compFault* Invalid company id Invalid catalog number Unknown attribute
CheckDelivery	*cdIn* Company id Catalog number Number of items required	*cdOut* Expected delivery date	*cdFault* Invalid company id No availability Zero items requested
MakeVirtualOrder	*voIn* Company id Catalog number Number of items required	*voOut* Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	*voFault* Invalid company id Invalid catalog number Zero items requested

Figure 18.11 Catalog interface design

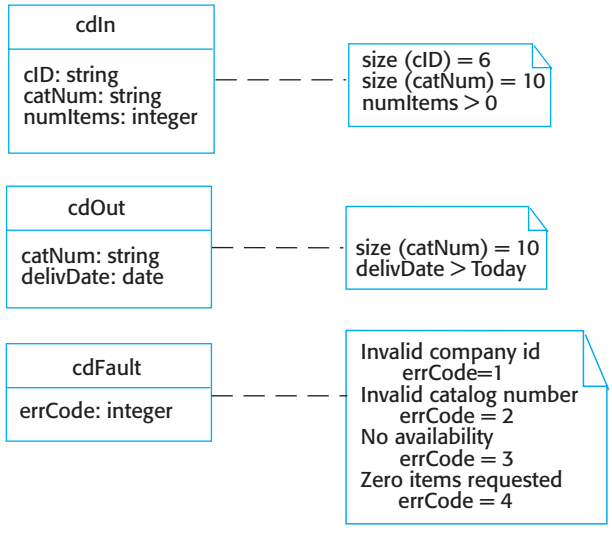


Figure 18.12 UML definition of input and output messages

Notice how I have added detail to the description by annotating the UML diagram with constraints. These details define the length of the strings representing the company and the catalog item, and specify that the number of items must be greater than zero and that delivery must be after the current date. The annotations also show which error codes are associated with each possible fault.

The catalog service is an example of a practical service, which illustrates that it is not always straightforward whether to choose a RESTful or a SOAP-based approach to service implementation. As an entity-based service, the catalog can be represented as a resource, which suggests that a RESTful model is the right one to use. However, operations on the catalog are not simple GET operations, and you need to maintain some state in an interaction session with the catalog. This suggests the use of a SOAP-based approach. Such dilemmas are common in service engineering, and usually local circumstances (e.g., availability of expertise) are a major factor in the decision of which approach to use.

To implement a set of RESTful services, you have to decide on the set of resources that will be used to represent the catalog and how the fundamental GET, POST, and PUT operations will operate on these resources. Some of these design decisions are straightforward:

1. There should be a resource representing a company-specific catalog. This should have a URL of the form <base catalog>/<company name> and should be created using a POST operation.
2. Each catalog item should have its own URL of the form <base catalog>/<company name>/<item identifier>.
3. You use the GET operation to retrieve items. Lookup is implemented by using the URL of an item in a catalog as the GET parameter. Search is implemented by using GET with the company catalog as the URL and the search string as a query parameter. This GET operation returns a list of URLs of the items matching the search.

However, the Compare, CheckDelivery, and MakeVirtualOrder operations are more complex:

1. The Compare operation can be implemented as a sequence of GET operations to retrieve the individual items, followed by a POST operation to create the comparison table and a final GET operation to return this to the user.
2. The CheckDelivery and MakeVirtualOrder operations require an additional resource, representing a virtual order. A POST operation is used to create this resource with the number of items required. The company id is used to automatically fill in the order form, and the delivery date is calculated. The resource can then be retrieved using a GET operation.

You need to think carefully about how exceptions are mapped onto the standard http response codes such as a 404 code, meaning that a URL cannot be retrieved. I don't have space to go into this issue here, but it adds a further level of complexity to the service interface design.



Legacy system services

Legacy systems are old software systems that are used by an organization. It may not be cost-effective to rewrite or replace these systems, and many organizations would like to use them in conjunction with more modern systems. One of the most important uses of services is to implement "wrappers" for legacy systems that provide access to a system's functions and data. These systems can then be accessed over the web and integrated with other applications.

http://software-engineering-book.com/web/legacy-services

For SOAP-based services, the realization process, in this case, is simpler as the logical interface design can be translated automatically into WSDL. Most programming environments that support service-oriented development (e.g., the ECLIPSE environment) include tools that can translate a logical interface description into its corresponding WSDL representation.

18.3.3 Service implementation and deployment

Once you have identified candidate services and designed their interfaces, the final stage of the service engineering process is service implementation. This implementation may involve programming the service using a language such as Java or C#. Both of these languages include libraries with extensive support for developing SOAP-based and RESTful services.

Alternatively, you can implement services by creating service interfaces to existing components or legacy systems. Software assets that have already proved to be useful can therefore be made available for reuse. In the case of legacy systems, it may mean that the system functionality can be accessed by new applications. You can also develop new services by defining compositions of existing services, as I explain in Section 18.4.

Once a service has been implemented, it then has to be tested before it is deployed. This involves examining and partitioning the service inputs (as explained in Chapter 8), creating input messages that reflect these input combinations, and then checking that the outputs are expected. You should always try to generate exceptions during the test to check that the service can cope with invalid inputs. For SOAP-based services, testing tools are available that allow services to be examined and tested, and that generate tests from a WSDL specification. However, these tools can only test the conformity of the service interface to the WSDL. They cannot test the service's functional behavior.

Service deployment, the final stage of the process, involves making the service available for use on a web server. Most server software makes this operation straightforward. You install the file containing the executable service in a specific directory. It then automatically becomes available for use.

If the service is intended to be available within a large organization or as a publicly available service, you then have to provide documentation for external service users. Potential users can then decide if the service is likely to meet their needs and

if they can trust you, as a service provider, to deliver the service reliably and securely. Information that you may include in a service description might be:

1. Information about your business, contact details, and so on. This is important for trust reasons. External users of a service have to be confident that it will not behave maliciously. Information about the service provider allows users to check their credentials with business information agencies.
2. An informal description of the functionality provided by the service. This helps potential users to decide if the service is what they want.
3. A description of how to use the service. For simple services, this can be an informal textual description that explains the input and output parameters. For more complex SOAP-based services, the WSDL description may be published.
4. Subscription information that allows users to register for information about updates to the service.

A general difficulty with service specifications is that the functional behavior of the service is usually specified informally, as a natural language description. Natural language descriptions are easy to read, but they are subject to misinterpretation. To address this problem, there has been extensive research on using ontologies and ontology languages for specifying service semantics by marking up the service with ontology information (W3C 2012). However, ontology-based specification is complex and not widely understood. Consequently, it has not been widely used.

18.4 Service composition

The underlying principle of service-oriented software engineering is that you compose and configure services to create new, composite services. These may be integrated with a user interface implemented in a browser to create a web application, or they may be used as components in some other service composition. The services involved in the composition may be specially developed for the application, business services developed within a company, or services from an external provider. Both RESTful and SOAP-based services can be composed to create services with extended functionality.

Many companies have converted their enterprise applications into service-oriented systems, where the basic application building block is a service rather than a component. This allows for widespread reuse within the company. We are now seeing the emergence of interorganizational applications between trusted suppliers, who use each other's services. The final realization of the long-term vision of service-oriented systems will rely on the development of a "services market," where services are bought from trusted external suppliers.

Service composition may be used to integrate separate business processes to provide an integrated process offering more extensive functionality. Say an airline wishes to develop a travel aggregation service that provides a complete vacation package for

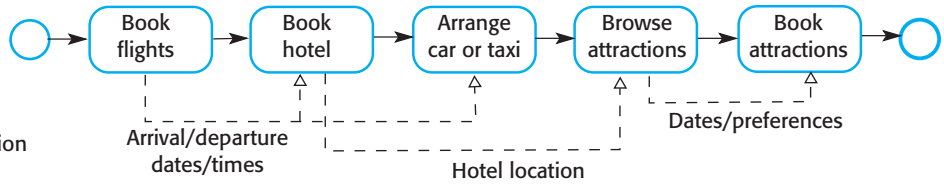


Figure 18.13 Vacation package workflow

travelers. In addition to booking their flights, travelers can also book hotels in their preferred location, arrange car rental or book a taxi from the airport, browse a travel guide, and make reservations to visit local attractions. To create this application, the airline composes its own booking service with services offered by a hotel booking agency, rental car and taxi companies, and reservation services offered by owners of local attractions. The end result is a single service that integrates the services from different providers.

You can think of this process as a sequence of separate steps, as shown in Figure 18.13. Information is passed from one step to the next. For example, the rental car company is informed of the time that the flight is scheduled to arrive. The sequence of steps is called a workflow—a set of activities ordered in time, with each activity carrying out some part of the work. A workflow is a model of a business process; that is, it sets out the steps involved in reaching a particular goal that is important for a business. In this case, the business process is the vacation booking service, offered by the airline.

Workflow is a simple idea, and the above scenario of booking a vacation seems to be straightforward. In practice, service composition is usually more complex than this simple model implies. You have to consider the possibility of service failure and include exception management to handle these failures. You also have to take into account nonstandard demands made by users of the application. For example, say a traveler was disabled and required a wheelchair to be rented and delivered to the airport. This would require extra services to be implemented and composed, with additional steps added to the workflow.

When designing a travel aggregation service, you must be able to cope with situations where the normal execution of one of the services results in an incompatibility with some other service execution. For example, say a flight is booked to leave on June 1 and to return on June 7. The workflow then proceeds to the hotel booking stage. However, the resort is hosting a major convention until June 2, so no hotel rooms are available. The hotel booking service reports this lack of availability. This is not a failure; lack of availability is a common situation.

You therefore have to "undo" the flight booking and pass the information about lack of availability back to the user. He or she then has to decide whether to change the dates or the resort. In workflow terminology, this is called a compensation action. Compensation actions are used to undo actions that have already been completed but that must be changed as a result of later workflow activities.

The process of designing new services by reusing existing services is a process of software design with reuse (Figure 18.13). Design with reuse inevitably involves requirements compromises. The "ideal" requirements for the system have to be modified to reflect the services that are actually available, whose costs fall within budget and whose quality of service is acceptable.

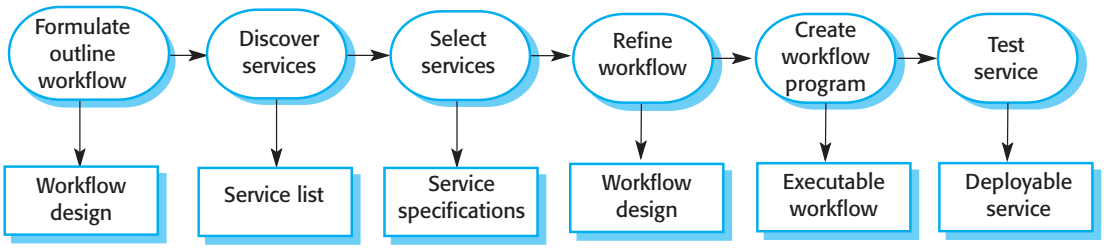


Figure 18.14 Service construction by composition

I have shown the six key stages in the process of system construction by composition in Figure 18.14:

1. *Formulate outline workflow* In this initial stage of service design, you use the requirements for the composite service as a basis for creating an "ideal" service design. You should create a fairly abstract design at this stage, with the intention of adding details once you know more about available services.

2. *Discover services* During this stage of the process, you look for existing services to include in the composition. Most service reuse is within enterprises, so this may involve searching local service catalogs. Alternatively, you may search the services offered by trusted service providers, such as Oracle and Microsoft.

3. *Select possible services* From the set of possible service candidates that you have discovered, you then select possible services that can implement workflow activities. Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

4. *Refine workflow* On the basis of information about the services that you have selected, you then refine the workflow. This involves adding detail to the abstract description and perhaps adding or removing workflow activities. You may then repeat the service discovery and selection stages. Once a stable set of services has been chosen and the final workflow design established, you move on to the next stage in the process.

5. *Create workflow program* During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can implement workflow programs using a programming language, such as Java or C#, or by using a workflow language, such as BPMN (explained below). This stage may also involve the creation of web-based user interfaces to allow the new service to be accessed from a web browser.

6. *Test completed service or application* The process of testing the completed, composite service is more complex than component testing in situations where external services are used. I discuss testing issues in Section 18.4.2.

This process assumes that existing services are available for composition. If you rely on external information that is not available through a service interface, you may have to implement these services yourself. This usually involves a "screen

scraping" process where your program extracts information from the HTML text of web pages that are sent to a browser for rendering.

18.4.1 Workflow design and implementation

Workflow design involves analyzing existing or planned business processes to understand the tasks involved and how these tasks exchange information. You then define the new business process in a workflow design notation. This sets out the stages involved in enacting the process and the information that is passed between the different process stages. However, existing processes may be informal and dependent on the skills and ability of the people involved. There may be no "normal" way of working or process definition. In such cases, you have to use your knowledge of the current process to design a workflow that achieves the same goals.

Workflows represent business process models. They are graphical models that are written using UML activity diagrams or BPMN, the Business Process Modeling Notation (White and Miers 2008; OMG 2011). I use BPMN for the examples in this chapter. If you use SOAP-based services, it is possible to convert BPMN workflows automatically into WS-BPEL, an XML-based workflow language. This is conformant with other web service standards such as SOAP and WSDL. RESTful services may be composed within a program in a standard programming language such as Java. Alternatively, a composition language used for service mashups may be used (Rosenberg et al. 2008).

Figure 18.15 is an example of a simple BPMN model of part of the vacation package scenario, shown in Figure 18.14. The model shows a simplified workflow for hotel booking and assumes the existence of a Hotels service with associated operations called GetRequirements, CheckAvailability, ReserveRooms, NoAvailability, ConfirmReservation, and CancelReservation. The process involves getting requirements from the customer, checking room availability, and then, if rooms are available, making a booking for the required dates.

This model introduces some of the core concepts of BPMN that are used to create workflow models:

1. Rectangles with rounded corners represent activities. An activity can be executed by a human or by an automated service.
2. Circles represent discrete events. An event is something that happens during a business process. A simple circle is used to represent a starting event and a darker circle to represent an end event. A double circle (not shown) is used to represent an intermediate event. Events can be clock events, thus allowing workflows to be executed periodically or timed out.
3. A diamond is used to represent a gateway. A gateway is a stage in the process where some choice is made. For example, in Figure 18.15, a choice is made on the basis of whether or not rooms are available.
4. A solid arrow shows the sequence of activities; a dashed arrow represents message flow between activities. In Figure 18.15, these messages are passed between the hotel booking service and the customer.

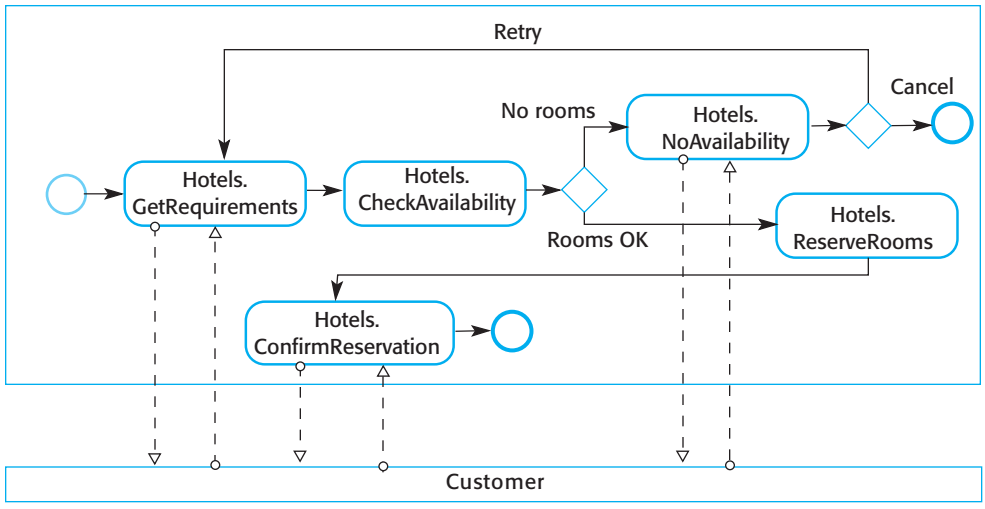


Figure 18.15 A fragment of a hotel booking workflow

These key features are enough to describe most workflows. However, BPMN includes many additional features that I don't have space to describe here. These add information to a business process description that allows it to be automatically translated into an executable service.

Figure 18.15 shows a process that is enacted in a single organization, the company that provides a booking service. However, the key benefit of a service-oriented approach is that it supports interorganizational computing. This means that a computation involves processes and services in different companies. This process is represented in BPMN by developing separate workflows for each of the organizations involved with interactions between them.

To illustrate multiple workflow processes, I use a different example, drawn from high-performance computing, where hardware is offered as a service. Services are created to provide access to high-performance computers to a geographically distributed user community. In this example, a vector-processing computer (a machine that can carry out parallel computations on arrays of values) is offered as a service (VectorProcService) by a research laboratory. This is accessed through another service called SetupComputation. These services and their interactions are shown in Figure 18.16.

In this example, the workflow for the SetupComputation service asks for access to a vector processor and, if a processor is available, establishes the computation required and downloads data to the processing service. Once the computation is complete, the results are stored on the local computer. The workflow for VectorProcService includes the following steps:

Check if a processor is available

Allocate resources for the computation

Initialize the system

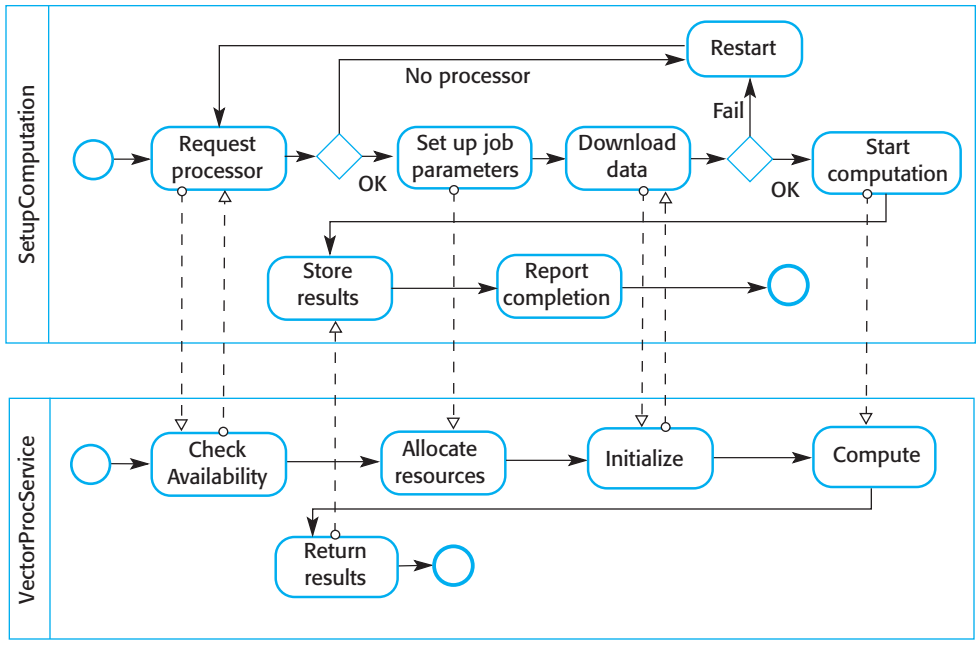


Figure 18.16 Interacting workflows

Carry out the computation

Return the results to the client service

In BPMN terms, the workflow for each organization is represented in a separate pool. It is shown graphically by enclosing the workflow for each participant in the process in a rectangle, with the name written vertically on the left edge. The workflows in each pool are coordinated by exchanging messages. In situations where different parts of an organization are involved in a workflow, pools are divided into named "lanes." Each lane shows the activities in that part of the organization.

Once a business process model has been designed, it has to be refined depending on the services that have been discovered. As I suggested in the discussion of Figure 18.14, the model may go through a number of iterations until a design that allows the maximum possible reuse of available services has been created.

Once the final design is available, you can then develop the final service-oriented system. This involves implementing services that are not available for reuse and converting the workflow model into an executable program. As services are implementation-language independent, new services can be written in any language. The workflow model may be automatically processed to create an executable WS-BPEL model if SOAP-based services are used. Alternatively, if RESTful services are used, the workflow may be manually programmed, with the model acting as a program specification.

18.4.2 Testing service compositions

Testing is important in all system development processes as it demonstrates that a system meets its functional and non-functional requirements and detects defects that

have been introduced during the development process. Many testing techniques, such as program inspections and coverage testing, rely on analysis of the software source code. However, if you use services from an external provider, you will not have access to the source code of the service implementations. You cannot therefore use "white box" testing techniques that rely on the source code of the system.

As well as problems of understanding the implementation of the service, testers may also face further difficulties when testing service compositions:

1. External services are under the control of the service provider rather than the user of the service. The service provider may withdraw these services at any time or may make changes to them, which invalidates any previous application testing. These problems are handled in software components by maintaining different versions of the component, but service versions are not normally supported.
2. If services are dynamically bound, an application may not always use the same service each time that it is executed. Therefore, tests may be successful when an application is bound to a particular service, but it cannot be guaranteed that that service will be used during an actual execution of the system. This problem has been one reason why dynamic binding has not been widely used.
3. The non-functional behavior of a service is not simply dependent on how it is used by the application that is being tested. A service may perform well during testing because it is not operating under a heavy load. In practice, the observed service behavior may be different because of the demands made by other service users.
4. The payment model for services could make service testing very expensive. There are different possible payment models: Some services may be freely available, some may be paid for by subscription, and others may be paid for on a per-use basis. If services are free, then the service provider will not wish them to be loaded by applications being tested; if a subscription is required, then a service user may be reluctant to enter into a subscription agreement before testing the service. Similarly, if the usage is based on payment for each use, service users may find the cost of testing to be prohibitive.
5. I have discussed the notion of compensation actions that are invoked when an exception occurs and previous commitments that have been made (such as a flight reservation) have to be revoked. There is a problem in testing such actions as they may depend on the failure of other services. Simulating the failure of these services during the testing process is usually difficult.

These problems are particularly acute when external services are used. They are less serious when services are used within the same company or where cooperating companies trust services offered by their partners. In such cases, source code may be available to guide the testing process, and payment for services is unlikely to be a problem. Resolving these testing problems and producing guidelines, tools, and techniques for testing service-oriented applications remains an important research issue.

KEY POINTS

- Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems.
- Services may be implemented within a service-oriented architecture using a set of XML-based web service standards. These include standards for service communication, interface definition, and service enactment in workflows.
- Alternatively, a RESTful architecture may be used, which is based on resources and standard operations on these resources. A RESTful approach uses the http and https protocols for service communication and maps operations on the standard http verbs POST, GET, PUT, and DELETE.
- Services may be classified as utility services that provide a general-purpose functionality, business services that implement part of a business process, or coordination services that coordinate the execution of other services.
- The service engineering process involves identifying candidate services for implementation, defining the service interface, and implementing, testing, and deploying the service.
- The development of software using services is based on the idea that programs are created by composing and configuring services to create new composite services and systems.
- Graphical workflow languages, such as BPMN, may be used to describe a business process and the services used in that process. These languages can describe interactions between the organizations that are involved.

FURTHER READING

There is an immense amount of tutorial material on the web covering all aspects of web services. However, I found the book by Thomas Erl to be the best overview and description of services and service standards. Erl includes some discussion of software engineering issues in service-oriented computing. He has also written more recent books on RESTful services.

Service-Oriented Architecture: Concepts, Technology and Design. Erl has written a number of books on service-oriented systems covering both SOA and RESTful architectures. In this book, Erl discusses SOA and web service standards but mostly concentrates on discussing how a service-oriented approach may be used at all stages of the software process. (T. Erl, Prentice-Hall, 2005).

"Service-oriented architecture." This is a good, readable introduction to SOA. (Various authors, 2008) http://msdn.microsoft.com/en-us/library/bb833022.aspx

"RESTful Web Services: The Basics." A good introductory tutorial on the RESTful approach and RESTful services. (A. Rodriguez, 2008). https://www.ibm.com/developerworks/webservices/library/ws-restful/

Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL, and RESTful Web Services. This is a more advanced text for developers who wish to use web services in enterprise applications. It describes a number of common problems and abstract web service solutions to these problems. (R. Daigneau, Addison-Wesley, 2012).

"Web Services Tutorial." This is an extensive tutorial on all aspects of service-oriented architecture, web services, and web service standards, written by people involved in the development of these standards. Very useful if you need a detailed understanding of the standards. (W3C schools, 1999–2014) http://www.w3schools.com/webservices/default.asp

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

EXERCISES

18.1. Why is it important to define exceptions in service engineering?

18.2. Standards are fundamental to service-oriented architectures, and it was believed that standards conformance was essential for successful adoption of a service-based approach. However, RESTful services, which are increasingly widely used, are not standards-based. Discuss why you think this change has occurred and whether or not you think that the lack of standards will inhibit the development and takeup of RESTful services.

18.3. Extend Figure 18.5 to include WSDL definitions for `MaxMinType` and `InDataFault`. The temperatures should be represented as integers, with an additional field indicating whether the temperature is in degrees Fahrenheit or degrees Celsius. `InDataFault` should be a simple type consisting of an error code.

18.4. Suggest how the SimpleInterestCalculator service could be implemented as a RESTful service.

18.5. What is a workflow? List out the key stages in the process of system construction by composition.

18.6. Design possible input and output messages for the services shown in Figure 18.13. You may specify these in the UML or in XML.

18.7. Giving reasons for your answer, suggest two important types of application where you would *not* recommend the use of service-oriented architecture.

18.8. Explain what is meant by a "compensation action" and, using an example, show why these actions may have to be included in workflows.

18.9. For the example of the vacation package reservation service, design a workflow that will book ground transportation for a group of passengers arriving at an airport. They should be given the option of booking either a taxi or a hire car. You may assume that the taxi and rental car companies offer web services to make a reservation.

18.10. Using an example, explain in detail why the thorough testing of services that include compensation actions is difficult.

REFERENCES

Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice-Hall.

———. 2005. *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice-Hall.

Fielding, R. 2000. "Representational State Transfer." *Architectural Styles and the Design of Network-Based Software Architecture*. https://www.ics.uci.edu/~fielding/pubs/. . ./fielding_dissertation.pdf

Lovelock, C, S Vandermerwe, and B Lewis. 1996. *Services Marketing*. Englewood Cliffs, NJ.: Prentice-Hall.

Newcomer, E., and G. Lomow. 2005. *Understanding SOA with Web Services*. Boston: Addison-Wesley.

OMG. 2011. "Documents Associated with Business Process Model and Notation (BPMN) Version 2.0." http://www.omg.org/spec/BPMN/2.0/

Pautasso, C., O. Zimmermann, and F. Leymann. 2008. "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision." In *Proc. WWW 2008*, 805–14. Beijing, China. doi:10.1145/1367497.1367606.

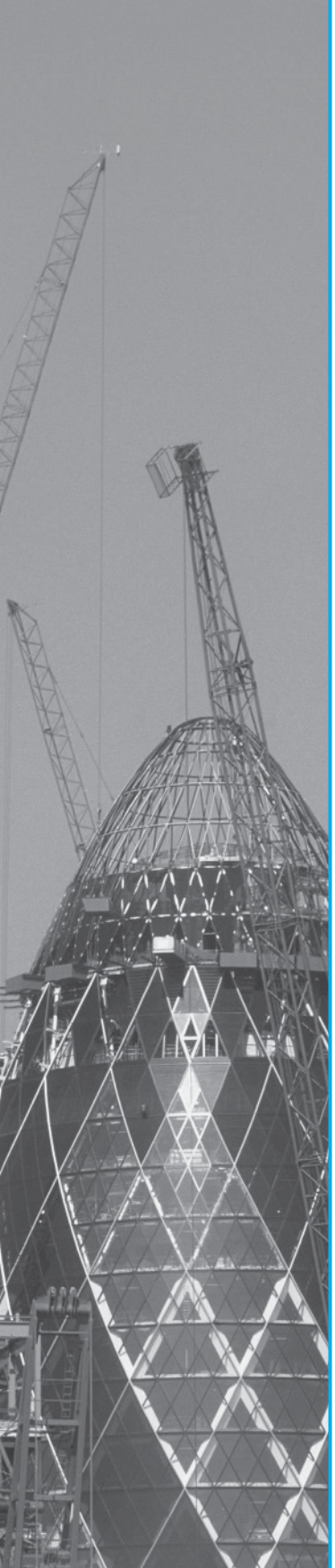
Richardson, L., and S. Ruby. 2007. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media Inc.

Rosenberg, F., F. Curbera, M. Duftler, and R. Khalaf. 2008. "Composing RESTful Services and Collaborative Workflows: A Lightweight Approach." *IEEE Internet Computing* 12 (5): 24–31. doi:10.1109/MIC.2008.98.

W3C. 2012. "OWL 2 Web Ontology Language." http://www.w3.org/TR/owl2-overview/

———. 2013. "Web of Services." http://www.w3.org/standards/webofservices/

White, S. A., and D. Miers. 2008. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL. USA: Future Strategies Inc.



19 Systems engineering

Objectives

The objectives of this chapter are to explain why software engineers should understand systems engineering and to introduce the most important systems engineering processes. When you have read this chapter, you will:

- know what is meant by a sociotechnical system and understand why human, social, and organizational issues affect the requirements and design of software systems;
- understand the idea of conceptual design and why it is an essential first stage in the systems engineering process;
- know what is meant by system procurement and understand why different system procurement processes are used for different types of system;
- know about the key systems engineering development processes and their relationships.

Contents

19.1 Sociotechnical systems
19.2 Conceptual design
19.3 System procurement
19.4 System development
19.5 System operation and evolution

A computer only becomes useful when it includes both software and hardware. Without hardware, a software system is an abstraction—simply a representation of some human knowledge and ideas. Without software, a hardware system is a set of inert electronic devices. However, if you put them together to form a computer system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system: It is more than the sum of its parts. Systems have properties that only become apparent when their components are integrated and operate together. Furthermore, systems are developed to support human activities—work, entertainment, communication, protection of people and the environment, and so on. They interact with people, and their design is influenced by human and organizational concerns. Hardware, human, social, and organizational factors have to be taken into account when developing all professional software systems.

Systems that include software fall into two categories:

1. *Technical computer-based systems* are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones, and other equipment with embedded software. Applications for PCs, computer games, and mobile devices are also technical systems. Individuals and organizations use technical systems for a particular purpose, but knowledge of this purpose is not part of the technical system. For example, the word processor I am using (Microsoft Word) is not aware that it is being used to write a book.

2 *Sociotechnical systems:* include one or more technical systems but, crucially, also people, who understand the purpose of the system, within the system itself. Sociotechnical systems have defined operational processes, and people (the operators) are inherent parts of the system. They are governed by organizational policies and rules and may be affected by external constraints such as national laws and regulatory policies. For example, this book was created through a sociotechnical publishing system that includes various processes (creation, editing, layout, etc.) and technical systems (Microsoft Word and Excel, Adobe Illustrator, InDesign, etc.).

Systems engineering (White et al. 1993; Stevens et al. 1998; Thayer 2002) is the activity of designing entire systems, taking into account the characteristics of hardware, software, and human elements of these systems. Systems engineering includes everything to do with procuring, specifying, developing, deploying, operating, and maintaining both technical and sociotechnical systems. Systems engineers have to consider the capabilities of hardware and software as well as the system's interactions with users and its environment. They must think about the system's services, the constraints under which the system must be built and operated, and the ways in which the system is used.

In this chapter, my focus is on the engineering of large and complex software-intensive systems. These are "enterprise systems," that is, systems that are used to support the goals of a large organization. Enterprise systems are used by government and the military services as well as large companies and other public bodies.

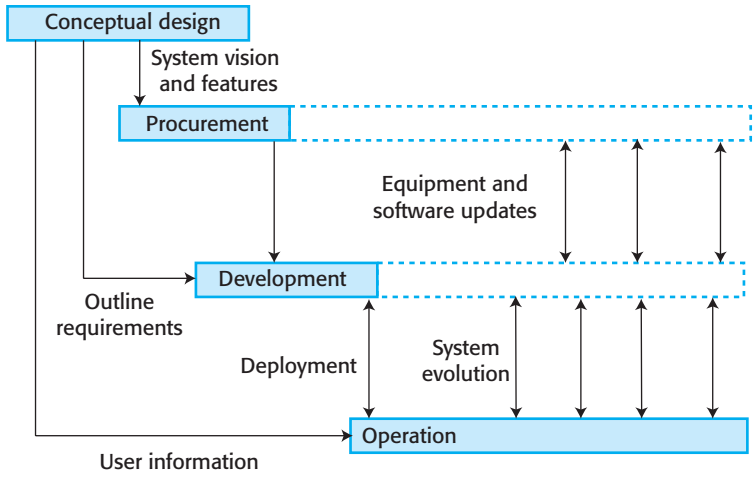


Figure 19.1 Stages of systems engineering

They are sociotechnical systems that are influenced by the ways that the organization works and by national and international rules and regulations. They may be made up of a number of separate systems and are distributed systems with large-scale databases. They have a long lifetime and are critical for the operation of the enterprise.

I believe that it is important for software engineers to know about systems engineering and to be active participants in systems engineering processes for two reasons:

1. Software is now the dominant element in all enterprise systems, yet many senior decision makers in organizations have a limited understanding of software. Software engineers have to play a more active part in high-level systems decision making if the system software is to be dependable and developed on time and to budget.
2. As a software engineer, it helps if you have a broader awareness of how software interacts with other hardware and software systems, and the human, social, and organizational factors that affect the ways in which software is used. This knowledge helps you understand the limits of software and to design better software systems.

There are four overlapping stages (Figure 19.1) in the lifetime of large, complex systems:

1. *Conceptual design* This initial systems engineering activity develops the concept of the type of system that is required. It sets out, in nontechnical language, the purpose of the system, why it is needed, and the high-level features that users might expect to see in the system. It may also describe broad constraints, such as the need for interoperability with other systems. These limit the freedom of systems engineers in designing and developing the system.
2. *Procurement or acquisition* During this stage, the conceptual design is further developed so that information is available to make decisions about the contract for the system development. This may involve making decisions about the distribution of

functionality across hardware, software, and operational processes. You also make decisions about which hardware and software has to be acquired, which suppliers should develop the system, and the terms and conditions of the supply contract.

3. *Development* During this stage, the system is developed. Development processes include requirements definition, system design, hardware and software engineering, system integration, and testing. Operational processes are defined, and the training courses for system users are designed.

4. *Operation* At this stage, the system is deployed, users are trained, and the system is brought into use. The planned operational processes usually then have to change to reflect the real working environment where the system is used. Over time, the system evolves as new requirements are identified. Eventually, the system declines in value, and it is decommissioned and replaced.

Figure 19.1 shows the interactions between these stages. The conceptual design activity is a basis for the system procurement and development but is also used to provide information to users about the system. Development and procurement overlap and further procurement during development, and operation may be needed as new equipment and software become available. Once the system is operational, requirements changes are inevitable; implementing these changes requires further development and, perhaps, software and hardware procurement.

Decisions made at any one of these stages may have a profound influence on the other stages. Design options may be restricted by procurement decisions on the scope of the system and on its hardware and software. Human errors made during the specification, design, and development stages may mean that faults are introduced into the system. A decision to limit testing for budget reasons may mean that faults are not discovered before a system is put into use. During operation, errors in configuring the system for deployment may lead to problems in using the system. Decisions made during the original procurement may be forgotten when system changes are proposed. This may lead to unforeseen consequences arising from the implementation of the changes.

An important difference between systems and software engineering is the involvement of a range of professionals throughout the lifetime of the system. These include engineers who may be involved in hardware and software design, system end-users, managers who are concerned with organizational issues, and experts in the system's application domain. For example, engineering the insulin pump system introduced in Chapter 1 requires experts in electronics, mechanical engineering, software, and medicine.

For very large systems, an even wider range of expertise may be required. Figure 19.2 illustrates the technical disciplines that may be involved in the procurement and development of a new system for air traffic management. Architects and civil engineers are involved because new air traffic management systems usually have to be installed in a new building. Electrical and mechanical engineers are involved to specify and maintain the power and air conditioning. Electronic engineers are concerned with computers, radars, and other equipment. Ergonomists

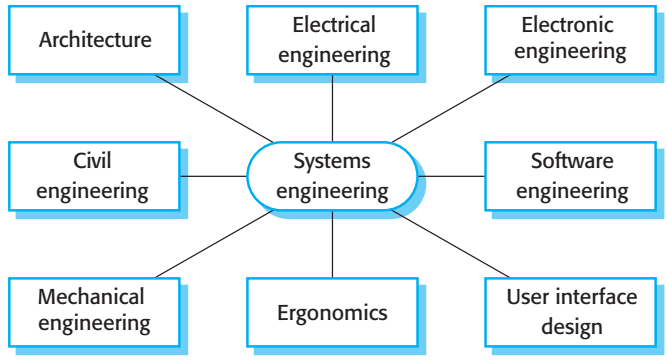


Figure 19.2 Professional disciplines involved in ATC systems engineering

design the controller workstations and software engineers, and user interface designers are responsible for the software in the system.

The involvement of a range of professional disciplines is essential because of the different types of components in complex systems. However, differences and misunderstandings between disciplines can lead to inappropriate design decisions. These poor decisions can delay the system's development or make it less suitable for its intended purpose. There are three reasons why there may be misunderstandings or other differences between engineers with different backgrounds:

1. Different professional disciplines often use the same words, but these words do not always mean the same thing. Consequently, misunderstandings are common in discussions between engineers from different backgrounds. If these are not discovered and resolved during system development, they can lead to errors in delivered systems. For example, an electronic engineer may know a bit about C programming but may not understand that a method in Java is like a function in C.
2. Each discipline makes assumptions about what other disciplines can or cannot do. These assumptions are often based on an inadequate understanding of what is possible. For example, an electronic engineer may decide that all signal processing (a computationally intensive task) should be done by software to simplify the hardware design. However, this may mean significantly greater software effort to ensure that the system processor can cope with the amount of computation that is resolved.
3. Disciplines try to protect their professional boundaries and may argue for certain design decisions because these decisions will call for their professional expertise. Therefore, a software engineer may argue for a software-based door locking system in a building, although a mechanical, key-based system may be more reliable.

My experience is that interdisciplinary working can be successful only if enough time is available for these issues to be discussed and resolved. This requires regular face-to-face discussions and a flexible approach from everyone involved in the systems engineering process.

19.1 Sociotechnical systems

The term *system* is universally used. We talk about computer systems, operating systems, payment systems, the education system, the system of government, and so on. These are all obviously quite different uses of the word "system," although they share the essential characteristic that, somehow, the system is more than simply the sum of its parts.

Abstract systems, such as the system of government, are outside the scope of this book. I focus here on systems that include computers and software and that have some specific purpose such as to enable communication, support navigation, or maintain medical records. A useful working definition of these types of system is as follows:

A system is a purposeful collection of interrelated components of different kinds that work together to deliver a set of services to the system owner and its users.

This general definition can cover a very wide range of systems. For example, a simple system, such as a laser pointer, delivers an indication service. It may include a few hardware components with a tiny control program in read-only memory (ROM). By contrast, an air traffic control system includes thousands of hardware and software components as well as human users who make decisions based on information from that computer system. It delivers a range of services, including providing information to pilots, maintaining safe separation of planes, utilizing airspace, and so on.

In all complex systems, the properties and behavior of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of other components. Software can only operate if the processor is operational. The processor can only carry out computations if the software system defining these computations has been successfully installed.

Large-scale systems are often "systems of systems." That is, they are made up of several separate systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. The same geographical information system may be used in systems for transport logistics and emergency command and control. Engineering systems of systems is an increasingly important topic in software engineering that I cover in Chapter 20.

Large-scale systems are, with a few exceptions, sociotechnical systems, which I explained in Chapter 10. That is, they do not just include software and hardware but also people, processes, and organizational policies. Sociotechnical systems are enterprise systems that are intended to help deliver a business purpose. This purpose might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, and so on. Because they are embedded in an organizational environment, the procurement, development, and use of these systems are influenced by the organization's policies and procedures, as well as by its working culture. The users of the system are people who are influenced by the way the organization is managed and by their interactions with other people inside and outside of the organization.

The close relationships between sociotechnical systems and the organizations that use these systems means that it is often difficult to establish system boundaries.

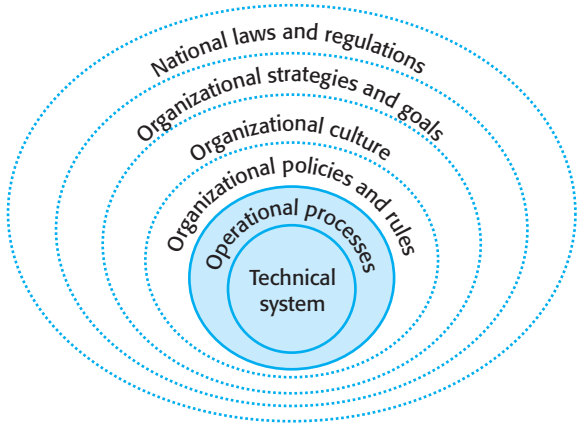


Figure 19.3 Layered structure of sociotechnical systems

Different people within the organization will see the boundaries of the system in different ways. This is significant because establishing what is and what is not in the scope of the system is important when defining the system requirements.

Figure 19.3 illustrates this problem. The diagram shows a sociotechnical system as a set of layers, where each layer contributes, in some way, to the functioning of the system. At the core is a software-intensive technical system and its operational processes (shaded in Figure 19.3). Most people would agree that these are both parts of the system. However, the system's behavior is influenced by a range of sociotechnical factors outside of the core. Should the system boundary simply be drawn around the core, or should it include other organizational levels?

Whether or not these broader sociotechnical considerations should be considered to be part of a system depends on the organization and its policies and rules. If organizational rules and policies can be changed, then some people might argue they should be part of the system. However, it is more difficult to change organizational culture and even more challenging to change strategy and goals. Only governments can change laws to accommodate a system. Moreover, different stakeholders may have different opinions on where the system boundaries should be drawn. There are no simple answers to these questions, but they have to be discussed and negotiated during the system design process.

Generally, large sociotechnical systems are used in organizations. When you are designing and developing sociotechnical systems, you need to understand, as far as possible, the organizational environment in which they will be used. If you don't, the systems may not meet business needs. Users and their managers may reject the system or fail to use it to its full potential.

Figure 19.4 shows the key elements in an organization that may affect the requirements, design, and operation of a sociotechnical system. A new system may lead to changes in some or all of these elements:

1. *Process changes* A new system may mean that people have to change the way that they work. If so, training will certainly be required. If changes are significant, or if they involve people losing their jobs, there is a danger that the users will resist the introduction of the system.

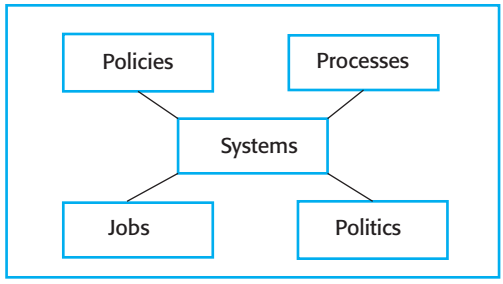


Figure 19.4
Organizational elements

2. *Job changes* New systems may deskill the users in an environment or cause them to change the way they work. If so, users may actively resist the introduction of the system into the organization. Professional staff, such as doctors or teachers, may resist system designs that require them to change their normal way of working. The people involved may feel that their professional expertise is being eroded and that their status in the organization is being reduced by the system.
3. *Organizational policies* The proposed system may not be completely consistent with organizational policies (e.g., on privacy). This may require system changes, policy changes, or process changes to bring the system and policies into line.
4. *Organizational politics* The system may change the political power structure in an organization. For example, if an organization is dependent on a complex system, those who control access to that system have a great deal of political power. Alternatively, if an organization reorganizes itself into a different structure, this may affect the requirements and use of the system.

Sociotechnical systems are complex systems, which means that it is practically impossible to have a complete understanding, in advance, of their behavior. This complexity leads to three important characteristics of sociotechnical systems:

1. They have emergent properties that are properties of the system as a whole, rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. Some of these relationships only come into existence when the system is integrated from its components, so the emergent properties can only be evaluated at that time. Security and dependability are examples of important emergent system properties.
2. They are nondeterministic, so that when presented with a specific input, they may not always produce the same output. The system's behavior depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behavior.
3. The system's success criteria are subjective rather than objective. The extent to which the system supports organizational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships

Property	Description
Reliability	System reliability depends on component reliability, but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, and modify or replace these components.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators, and its operating environment.
Volume	The volume of a system (the total space occupied) depends on how the component assemblies are arranged and connected.

Figure 19.5 Examples of emergent properties

and conflicts between organizational objectives, and how people in the organization interpret these objectives. New management may reinterpret the organizational objectives that a system was designed to support so that a "successful" system may then be seen as no longer fit for its intended purpose.

Sociotechnical considerations are often critical in determining whether or not a system has successfully met its objectives. Unfortunately, taking these into account is very difficult for engineers who have little experience of social or cultural studies. To help understand the effects of systems on organizations, various sociotechnical systems methodologies have been proposed. My paper on sociotechnical systems design discusses the advantages and disadvantages of these sociotechnical design methodologies (Baxter and Sommerville 2011).

19.1.1 Emergent properties

The complex relationships between the components in a system mean that a system is more than simply the sum of its parts. It has properties that are properties of the system as a whole. These "emergent properties" (Checkland 1981) cannot be attributed to any specific part of the system. Rather, they only emerge once the system components have been integrated. Some emergent properties, such as weight, can be derived directly from the subsystem properties. More often, however, they emerge from a combination of subsystem properties and subsystem relationships. The system property cannot be calculated directly from the properties of the individual system components. Examples of emergent properties are shown in Figure 19.5.

There are two types of emergent properties:

1. *Functional emergent properties,* when the purpose of a system only emerges after its components are integrated. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

2. *Non-functional emergent properties,* which relate to the behavior of the system in its operational environment. Reliability, performance, safety, and security are examples of these properties. These system characteristics are critical for computer-based systems, as failure to achieve a minimum defined level in these properties usually makes the system unusable. Some users may not need some of the system functions, so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.

Emergent properties, such as reliability, depend on both the properties of individual components and their interactions or relationships. For example, the reliability of a sociotechnical system is influenced by three things:

1. *Hardware reliability* What is the probability of hardware components failing, and how long does it take to repair a failed component?
2. *Software reliability* How likely is it that a software component will produce an incorrect output? Software failure is unlike hardware failure in that software does not wear out. Failures are often transient. The system carries on working after an incorrect result has been produced.
3. *Operator reliability* How likely is it that the operator of a system will make an error and provide an incorrect input? How likely is it that the software will fail to detect this error and propagate the mistake?

Hardware, software, and operator reliability are not independent but affect each other in unpredictable ways. Figure 19.6 shows how failures at one level can be propagated to other levels in the system. Say a hardware component in a system starts to go wrong. Hardware failure can sometimes generate spurious signals that are outside the range of inputs expected by the software. The software can then behave unpredictably and produce unexpected outputs. These may confuse and consequently cause stress in the system operator.

We know that people are more likely to make mistakes when they feel stressed. So a hardware failure may be the trigger for operator errors. These mistakes can, in turn, lead to unexpected software behavior, resulting in additional demands on the processor. This could overload the hardware, causing more failures and so on. Thus, an initial, relatively minor, failure, can rapidly develop into a serious problem that could lead to a complete shutdown of the system.

The reliability of a system depends on the context in which that system is used. However, the system's environment cannot be completely specified, and it is often impossible for the system designers to limit the environment for operational systems. Different systems operating within an environment may react to problems in unpredictable ways, thus affecting the reliability of all of these systems.

For example, say a system is designed to operate at normal room temperature. To allow for variations and exceptional conditions, the electronic components of a system are designed to operate within a certain range of temperatures, say, from 0 degrees to 40 degrees Celsius. Outside this temperature range, the components will

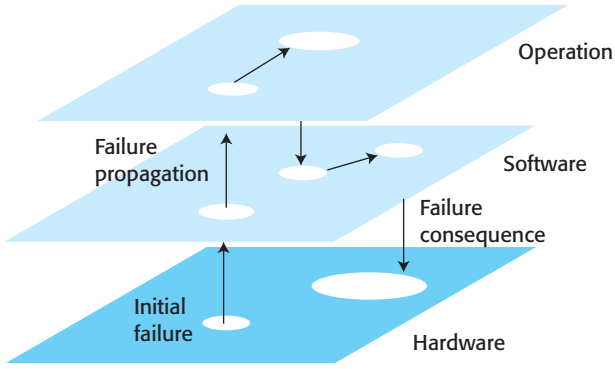


Figure 19.6 Failure propagation

behave in an unpredictable way. Now assume that this system is installed close to an air conditioner. If this air conditioner fails and vents hot gas over the electronics, then the system may overheat. The components, and hence the whole system may then fail.

If this system had been installed elsewhere in that environment, this problem would not have occurred. When the air conditioner worked properly, there were no problems. However, because of the physical closeness of these machines, an unanticipated relationship existed between them that led to system failure.

Like reliability, emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, are not directly measurable. Here, you are not simply concerned with attributes that relate to the behavior of the system but also with unwanted or unacceptable behavior.

A secure system is one that does not allow unauthorized access to its data. Unfortunately, it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these "shall not" properties after the system is operational. That is, you only know that a system is insecure when someone manages to penetrate the system.

19.1.2 Non-determinism

A deterministic system is one that is absolutely predictable. If we ignore issues of concurrency, software systems that run on reliable hardware are deterministic. When they are presented with a sequence of inputs they will always produce the same sequence of outputs. Of course, there is no such thing as completely reliable hardware, but hardware is usually reliable enough to think of hardware systems as deterministic.

People, on the other hand, are non-deterministic. When presented with exactly the same input (say a request to complete a task), their responses will depend on their emotional and physical state, the person making the request, other people in the environment, and whatever else they are doing. Sometimes they will be happy to do the work, and, at other times, they will refuse; sometimes they will perform a task well, and sometimes they will do it badly.

Sociotechnical systems are nondeterministic partly because they include people and partly because changes to the hardware, software, and data in these systems are

so frequent. The interactions between these changes are complex, and so the behavior of the system is unpredictable. Users do not know when and why changes have been made, so they see the system as nondeterministic.

For example, say a system is presented with a set of 20 test inputs. It processes these inputs and the results are recorded. At some later time, the same 20 test inputs are processed, and the results are compared to the previous stored results. Five of them are different. Does this mean that there have been five failures? Or are the differences simply reasonable variations in the system's behavior? You can only find this out by looking at the results in more depth and making judgments about the way the system has handled each input.

Non-determinism is often seen as a bad thing, and it is felt that designers should try to avoid nondeterministic behavior wherever possible. In fact, in sociotechnical systems, non-determinism has important benefits. It means that the behavior of a system is not fixed for all time but can change depending on the system's environment. For example, operators may observe that a system is showing signs of failure. Instead of using the system normally, they can change their behavior to diagnose and recover from the detected problems.

19.1.3 Success criteria

Generally, complex sociotechnical systems are developed to tackle "wicked problems" (Rittel and Webber 1973). A wicked problem is a problem that is so complex and that involves so many related entities that there is no definitive problem specification. Different stakeholders see the problem in different ways, and no one has a full understanding of the problem as a whole. The true nature of the problem may only emerge as a solution is developed.

An extreme example of a wicked problem is emergency planning to deal with the aftermath of an earthquake. No one can accurately predict where the epicenter of an earthquake will be, what time it will occur, or what effect it will have on the local environment. It is impossible to specify in detail how to deal with the problem. System designers have to make assumptions, but understanding what is required emerges only after the earthquake has happened.

This makes it difficult to define the success criteria for a system. How do you decide if a new system contributes to the business goals of the company that paid for the system? The judgment of success is not usually made against the original reasons for procuring and developing the system. Rather, it is based on whether or not the system is effective at the time it is deployed. As the business environment can change very quickly, the business goals may have changed significantly during the development of the system.

The situation is even more complex when there are multiple conflicting goals that are interpreted differently by different stakeholders. For instance, the system on which the Mentcare system is based was designed to support two separate business goals:

1. To improve the quality of care for sufferers from mental illness.
2. To improve the cost-effectiveness of treatments by providing managers with detailed reports of care provided and the costs of that care.

Unfortunately, these proved to be conflicting goals because the information that was needed to satisfy the reporting goal meant that doctors and nurses had to provide additional information, over and above the health records that they normally maintained. This reduced the quality of care for patients as it meant that clinical staff had less time to talk with them. From a doctor's perspective, this system was not an improvement on the previous manual system, but from a manager's perspective, it was.

Thus, any success criteria that are established in the early stages of the systems engineering process have to be regularly reconsidered during system development and use. You cannot evaluate these criteria objectively as they depend on the system's effect on its environment and its users. A system may apparently meet its requirements as originally specified but be practically useless because of changes in the environment where it is used.

19.2 Conceptual design

Once an idea for a system has been suggested, conceptual design is the very first thing that you do in the systems engineering process. In the conceptual design phase, you take that initial idea, investigate its feasibility, and develop it to create an overall vision of a system that could be developed. You then have to describe the envisaged system so that nonexperts, such as system users, senior company decision makers, or politicians, can understand what you are proposing.

There is an obvious overlap between conceptual design and requirements engineering. As part of the conceptual design process, you have to imagine how the proposed system will be used. This may involve discussions with potential users and other stakeholders, focus groups, and observations of how existing systems are used. The goal of these activities is to understand how users work, what is important to them, and what practical constraints on the system there might be.

The importance of establishing a vision of a proposed system is rarely mentioned in the software design and requirements literature. However, this vision has been part of the systems engineering process for military systems for many years. Fairley et al. (Fairley, Thayer, and Bjorke 1994) discuss the idea of concept analysis and the documentation of the results of concept analysis in a "Concept of Operations" (ConOps) document. This idea of developing a ConOps document is now widely used for large-scale systems, and you can find many examples of ConOps documents on the web.

Unfortunately, as is so often the case with military and government systems, good ideas can become mired in bureaucracy and inflexible standards. This is exactly what happened with ConOps, and a ConOps document standard was proposed (IEEE, 2007). As Mostashari et al. say (Mostashari et al. 2012), this tends to lead to long and unreadable documents, which do not serve their intended purpose. They propose a more agile approach to the development of a ConOps document with a shorter and more flexible document as the output of the process.

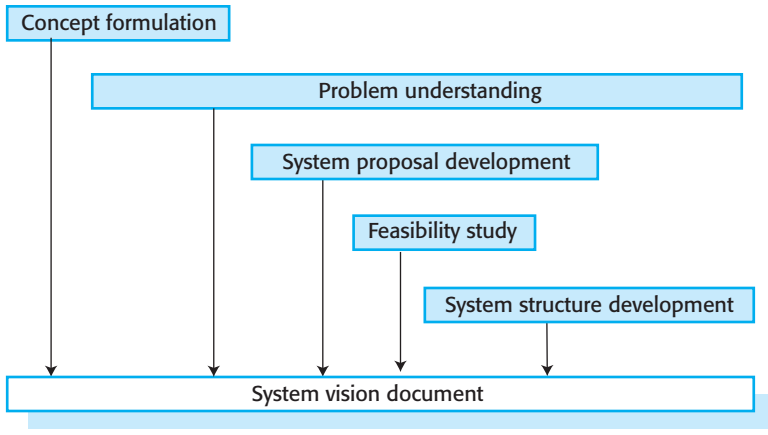


Figure 19.7 Conceptual design activities

I don't like the term *Concept of Operations* partly because of its military connotations and partly because I think that a conceptual design document is not just about system operation. It should also present the system engineer's understanding of why the system is being developed, an explanation of why the design proposals are appropriate, and, sometimes, an initial organization for the system. As Fairley says, "It should be organized to tell a story," that is, written so that people without a technical background can understand the proposals that are being made.

Figure 19.7 shows activities that may be part of the conceptual design process. Conceptual design should always be a team process that involves people from different backgrounds. I was part of the conceptual design team for the digital learning environment, introduced in Chapter 1. For the digital learning environment, the design team included teachers, education researchers, software engineers, system administrators, and system managers.

Concept formulation is the first stage of the process where you try to refine an initial statement of needs and work out what type of system would be best to meet the needs of system stakeholders. Initially, we were tasked with proposing an intranet for information sharing across schools that was easier to use than the current system. However, after discussions with teachers, we discovered that this was not really what was required. The existing system was awkward to use, but people had found workarounds. What was really required was a flexible digital learning environment that could be adapted by adding subject and age-specific tools and content that are freely available on the Internet.

We discovered this because the concept formulation activity overlapped with the activity of problem understanding. To understand a problem, you need to discuss with users and other stakeholders how they do their work. You need to find out what is important to them, what are the barriers that stop them from doing what they want to do, and their ideas of what changes are required. You need to be open-minded (it is their problem, not yours) and to be prepared to change your ideas when the reality does not match your initial vision.

In the system proposal development stage, the conceptual design team set out their ideas for alternative systems and these are the basis for a feasibility study to decide which of the ideas are worth further development. In a feasibility study, you should look at comparable systems that have been developed elsewhere and technological issues (e.g., use of mobile devices) that may affect use of the system. Then you need to assess whether or not the system could be implemented using current hardware and software technologies.

I have found that an additional useful activity is to develop an outline structure or architecture for the system. This activity is helpful both for making a feasibility assessment and for providing a basis for more detailed requirements engineering and architectural design. Furthermore, as the majority of systems are now assembled from existing systems and components, an initial architecture means that the key parts of the system have been identified and can be procured separately. This approach is often better than procuring a system as a monolithic unit from a single supplier.

For the digital learning environment, we decided on a layered service architecture (shown in Figure 1.8). All components in the system should be considered to be replaceable services. In this way, users can replace a standard service with their preferred alternative and so adapt the system to the ages and interests of the students learning with the system.

All of these activities generate information that is used to develop the system vision document. This is a critical document that senior decision makers use to decide whether or not further development of the system should go ahead. It is also used to develop further documents such as a risk analysis and budget estimate, which are also important inputs to the decision-making process.

Managers use the system vision document to understand the system; a procurement team uses it to define a tender document; and requirements engineers use it as a basis for refining the system requirements. Because these different people need different levels of detail, I suggest that the document should be structured into two parts:

1. A short summary for senior decision makers that presents the key points of the problem and the proposed system. It should be written so that readers can immediately see how the system will be used and the benefits that it will provide.
2. A number of appendices that develop the ideas in more detail and that can be used in the system procurement and requirements engineering activities.

It is challenging to write a summary of the system vision inasmuch as the readers are busy people who are unlikely to have a technical background. I have found that using user stories is very effective, providing a tangible vision of system use that nontechnical people can relate to. Stories should be short and personalized and should be a feasible description of the use of the system, as shown in Figure 19.8. There is another example of a user story from the same system in Chapter 4 (Figure 4.9).

Digital art

Jill is an S2 pupil at a secondary school in Dundee. She has a smartphone of her own, and the family has a shared Samsung tablet and a Dell laptop computer. At school, Jill signs on to the school computer and is presented with a personalized Glow+ environment, which includes a range of services, some chosen by her teachers and some she has chosen herself from the Glow app library.

She is working on a Celtic art project, and she uses Google to research a range of art sites. She sketches out some designs on paper and then uses the camera on her phone to photograph what she has done; she uploads this using the school wifi to her personal Glow+ space. Her homework is to complete the design and write a short commentary on her ideas.

At home, she uses the family tablet to sign on to Glow+, and she then uses an artwork app to process her photograph and to extend the work, add color, and so on. She finishes this part of the work, and to complete it she moves to her home laptop to type up her commentary. She uploads the finished work to Glow+ and sends a message to her art teacher that it is available for review. Her teacher looks at the project in a free period before Jill's next art class using a school tablet, and, in class, she discusses the work with Jill.

After the discussion, the teacher and Jill decide that the work should be shared, and so they publish it to the school web pages that show examples of students' work. In addition, the work is included in Jill's e-portfolio—her record of schoolwork from age 3 to 18.

Figure 19.8 A user story used in a system vision document

User stories are effective because, as already noted, readers can relate to them; in addition, they can show the capabilities of the proposed system in an easily accessible way. Of course, these are only part of a system vision, and the summary must also include a high-level description of the basic assumptions made and the ways in which the system will deliver value to the organization.

19.3 System procurement

System procurement or system acquisition is a process whose outcome is a decision to buy one or more systems from system suppliers. At this stage, decisions are made on the scope of a system that is to be purchased, system budgets and timescales, and high-level system requirements. Using this information, further decisions are then made on whether to procure a system, the type of system required, and the supplier or suppliers of the system. The drivers for these decisions are:

1. *The replacement of other organizational systems* If the organization has a mixture of systems that cannot work together or that are expensive to maintain, then procuring a replacement system, with additional capabilities, may lead to significant business benefits.
2. *The need to comply with external regulations* Increasingly, businesses are regulated and have to demonstrate compliance with externally defined regulations (e.g., Sarbanes–Oxley accounting regulations in the United States). Compliance may require the replacement of noncompliant systems or the provision of new systems specifically to monitor compliance.

3. *External competition* If a business needs to compete more effectively or maintain a competitive position, managers may decide to buy new systems to improve business efficiency or effectiveness. For military systems, the need to improve capability in the face of new threats is an important reason for procuring new systems.
4. *Business reorganization* Businesses and other organizations frequently restructure with the intention of improving efficiency and/or customer service. Reorganizations lead to changes in business processes that require new systems support.
5. *Available budget* The budget that is available is an obvious factor in determining the scope of new systems that can be procured.

In addition, new government systems are often procured to reflect political changes and political policies. For example, politicians may decide to buy new surveillance systems, which they claim will counter terrorism. Buying such systems shows voters that they are taking action.

Large complex systems are usually engineered using a mixture of off-the-shelf and specially built components. They are often integrated with existing legacy systems and organizational databases. When legacy systems and off-the-shelf systems are used, new custom software may be needed to integrate these components. The new software manages the component interfaces so that these components can interoperate. The need to develop this "glueware" is one reason why the savings from using off-the-shelf components are sometimes not as great as anticipated.

Three types of systems or system components may have to be procured:

1. Off-the-shelf applications that may be used without change and that need only minimal configuration for use.
2. Configurable application or ERP systems that have to be modified or adapted for use either by modifying the code or by using inbuilt configuration features, such as process definitions and rules.
3. Custom systems that have to be specially designed and implemented for use.

Each of these components tends to follow a different procurement process. Figure 19.9 illustrates the main features of the procurement process for these types of system. Key issues that affect procurement processes are:

1. Organizations often have an approved and recommended set of application software that has been checked by the IT department. It is usually possible to buy or acquire open-source software from this set directly without the need for detailed justification. For example, in the iLearn system, we recommended that Wordpress should be made available for student and staff blogs. If microphones are needed, off-the-shelf hardware can be bought. There are no detailed requirements, and the users adapt to the features of the chosen application.
2. Off-the-shelf components do not usually match requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing

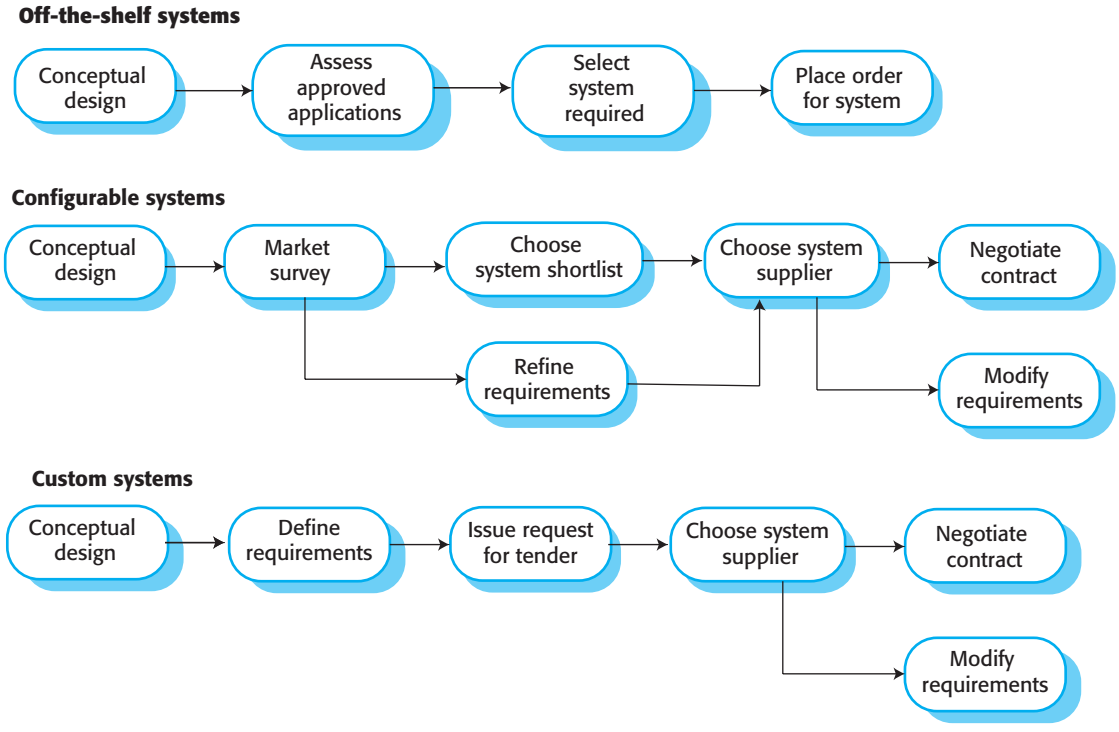


Figure 19.9 System procurement processes

a system means that you have to find the closest match between the system requirements and the facilities offered by off-the-shelf systems. ERP and other large-scale application systems usually fall into this category. You may then have to modify the requirements to fit in with the system assumptions. This can have knock-on effects on other subsystems. You also usually have an extensive configuration process to tailor and adapt the application or ERP system to the buyer's working environment.

3. When a system is to be built specially, the specification of requirements is part of the contract for the system being acquired. It is therefore a legal as well as a technical document. The requirements document is critical, and procurement processes of this type usually take a considerable amount of time.

4. For public sector systems in particular, there are detailed rules and regulations that affect the procurement of systems. For example, in the European Union, all public sector systems over a certain price must be open to tender by any supplier in Europe. This requires detailed tender documents to be drawn up and the tender to be advertised across Europe for a fixed period of time. Not only does this rule slow down the procurement process, it also tends to inhibit agile development. It forces the system buyer to develop requirements so that all companies have enough information to bid for the system contract.

5. For application systems that require change or for custom systems, there is usually a contract negotiation period when the customer and supplier negotiate the terms and conditions for development of the system. Once a system has been

selected, you may negotiate with the supplier on costs, license conditions, possible changes to the system, and other contractual issues. For custom systems, negotiations are likely to involve payment schedules, reporting, acceptance criteria, requirements change requests, and costs of system changes. During this process, requirements changes may be agreed that will reduce the overall costs and avoid some development problems.

Complex sociotechnical systems are rarely developed "in house" by the buyer of the system. Rather, external systems companies are invited to bid for the systems engineering contract. The customer's business is not systems engineering, so its employees do not have the skills needed to develop the systems themselves. For complex hardware/software systems, it may be necessary to use a group of suppliers, each with a different type of expertise.

For large systems, such as an air traffic management system, a group of suppliers may form a consortium to bid for a contract. The consortium should include all of the capabilities required for this type of system. For an ATC system, this would include computer hardware suppliers, software companies, peripheral suppliers, and suppliers of specialist equipment such as radar systems.

Customers do not usually wish to negotiate with multiple suppliers, so the contract is usually awarded to a principal contractor, who coordinates the project. The principal contractor coordinates the development of different subsystems by subcontractors. The subcontractors design and build parts of the system to a specification that is negotiated with the principal contractor and the customer. Once completed, the principal contractor integrates these components and delivers them to the customer.

Decisions made at the procurement stage of the systems engineering process are critical for later stages in that process. Poor procurement decisions often lead to problems such as late delivery of a system and development of systems that are unsuited to their operational environment. If the wrong system or the wrong supplier is chosen, then the technical processes of system and software engineering become more complex.

For example, I studied a system "failure" where a decision was made to choose an ERP system because this would "standardize" operations across the organization. These operations were very diverse, and it turned out there were good reasons for this. Standardization was practically impossible. The ERP system could not be adapted to cope with this diversity. It was ultimately abandoned after incurring costs of around £10 million.

Decisions and choices made during system procurement have a profound effect on the security and dependability of a system. For example, if a decision is made to procure an off-the-shelf system, then the organization has to accept that they have no influence over the security and dependability requirements of this system. System security depends on decisions made by system vendors. In addition, off-the-shelf systems may have known security weaknesses or may require complex configuration. Configuration errors, where entry points to the system are not properly secured, are a significant source of security problems.

On the other hand, a decision to procure a custom system means that a lot of effort must be devoted to understanding and defining security and dependability requirements. If a company has limited experience in this area, this is quite a difficult thing to do. If the

required level of dependability as well as acceptable system performance is to be achieved, then the development time may have to be extended and the budget increased.

Many bad procurement decisions stem from political rather than technical causes. Senior management may wish to have more control and so demand that a single system is used across an organization. Suppliers may be chosen because they have a long-standing relationship with a company rather than because they offer the best technology. Managers may wish to maintain compatibility with existing systems because they feel threatened by new technologies. As I discuss in Chapter 20, people who do not understand the required system are often responsible for procurement decisions. Engineering issues do not necessarily play a major part in their decision-making process.

19.4 System development

System development is a complex process in which the elements that are part of the system are developed or purchased and then integrated to create the final system. The system requirements are the bridge between the conceptual design and the development processes. During conceptual design, business and high-level functional and non-functional system requirements are defined. You can think of this as the start of development, hence the overlapping processes shown in Figure 19.1. Once contracts for the system elements have been agreed, more detailed requirements engineering takes place.

Figure 19.10 is a model of the systems development process. Systems engineering processes usually follow a "waterfall" process model similar to the one that I discussed in Chapter 2. Although the waterfall model is inappropriate for most types of software development, higher-level systems engineering processes are plan-driven processes that still follow this model.

Plan-driven processes are used in systems engineering because different elements of the system are independently developed. Different contractors are working concurrently on separate subsystems. Therefore, the interfaces to these elements have to be designed before development begins. For systems that include hardware and other equipment, changes during development can be very expensive or, sometimes, practically impossible. It is essential therefore, that the system requirements are fully understood before hardware development or building work begins.

One of the most confusing aspects of systems engineering is that companies use different terminology for each stage of the process. Sometimes, requirements engineering is part of the development process, and sometimes it is a separate activity. However, after conceptual design, there are seven fundamental development activities:

1. *Requirements engineering* is the process of refining, analyzing, and documenting the high-level and business requirements identified in the conceptual design. I have covered the most important requirements engineering activities in Chapter 4.

2. *Architectural design* overlaps significantly with the requirements engineering process. The process involves establishing the overall architecture of the system,

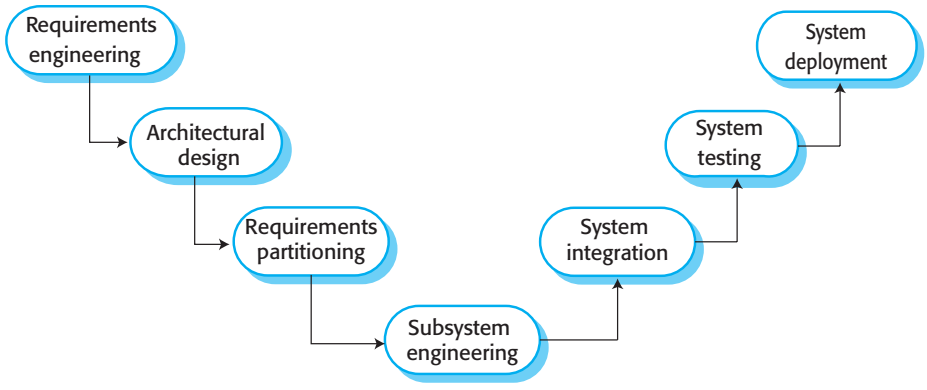


Figure 19.10 The systems development process

identifying the different system components, and understanding the relationships between them.

3. *Requirements partitioning* is concerned with deciding which subsystems (identified in the system architecture) are responsible for implementing the system requirements. Requirements may have to be allocated to hardware, software, or operational processes and prioritized for implementation. Ideally, you should allocate requirements to individual subsystems so that the implementation of a critical requirement does not need subsystem collaboration. However, this is not always possible. At this stage you also decide on the operational processes and on how these are used in the requirements implementation.

4. *Subsystem engineering* involves developing the software components of the system, configuring off-the-shelf hardware and software, designing, if necessary, special-purpose hardware, defining the operational processes for the system, and re-designing essential business processes.

5. *System integration* is the process of putting together system elements to create a new system. Only then do the emergent system properties become apparent.

6. *System testing* is an extended activity where the whole system is tested and problems are exposed. The subsystem engineering and system integration phases are reentered to repair these problems, tune the performance of the system, and implement new requirements. System testing may involve both testing by the system developer and acceptance/user testing by the organization that has procured the system.

7. *System deployment* is the process of making the system available to its users, transferring data from existing systems, and establishing communications with other systems in the environment. The process culminates with a "go live," after which users start to use the system to support their work.

Although the overall process is plan-driven, the processes of requirements development and system design are inextricably linked. The requirements and the high-level

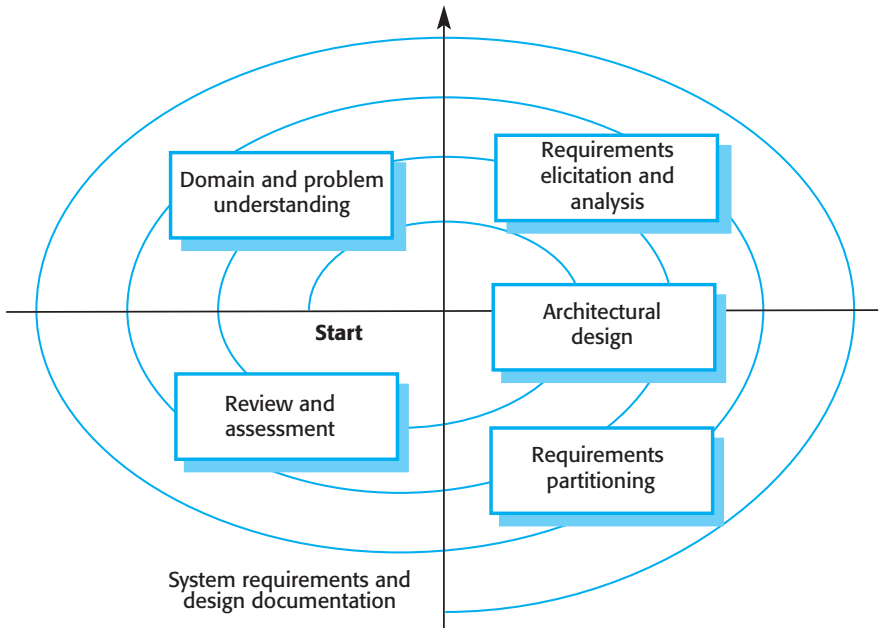


Figure 19.11 Requirements and design spiral

design are developed concurrently. Constraints posed by existing systems may limit design choices, and these choices may be specified in the requirements. You may have to do some initial design to structure and organize the requirements engineering process. As the design process continues, you may discover problems with existing requirements and new requirements may emerge. Consequently, you can think of these linked processes as a spiral, as shown in Figure 19.11.

The spiral reflects the reality that requirements affect design decisions and vice versa, and so it makes sense to interleave these processes. Starting in the center, each round of the spiral may add detail to the requirements and the design. As subsystems are identified in the architecture, decisions are made on the responsibilities of these subsystems for providing the system requirements. Some rounds of the spiral may focus on requirements, others on design. Sometimes new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

For almost all systems, many possible designs meet the requirements. These cover a range of solutions that combine hardware, software, and human operations. The solution that you choose for further development may be the most appropriate technical solution that meets the requirements. However, wider organizational and political considerations may influence the choice of solution. For example, a government client may prefer to use national rather than foreign suppliers for its system, even if national products are technically inferior.

These influences usually take effect in the review and assessment phase of the spiral model where designs and requirements may be accepted or rejected. The process ends when a review decides that the requirements and high-level design are sufficiently detailed for subsystems to be specified and designed.

Subsystem engineering involves designing and building the system's hardware and software components. For some types of systems, such as spacecraft, all hardware and software components may be designed and built during the development process. However, in most systems, some components are bought rather than developed. It is usually much cheaper to buy existing products than to develop special-purpose components. However, if you buy large off-the-shelf systems, such as ERP systems, there is a significant cost in configuring these systems for use in their operational environment.

Subsystems are usually developed in parallel. When problems that cut across subsystem boundaries are encountered, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive. Often "workarounds" that compensate for the problem must be found. These workarounds usually involve software changes to implement new requirements.

During systems integration, you take the independently developed subsystems and put them together to make up a complete system. This integration can be achieved using a "big bang" approach, where all the subsystems are integrated at the same time. However, for technical and managerial reasons, an incremental integration process where subsystems are integrated one at a time is the best approach:

1. It is usually impossible to schedule the development of all the subsystems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many subsystems are simultaneously integrated, an error that arises during testing may be in any of these subsystems. When a single subsystem is integrated with an already working system, errors that occur are probably in the newly integrated subsystem or in the interactions between the existing subsystems and the new subsystem.

As an increasing number of systems are built by integrating off-the-shelf hardware and software application systems, the distinction between implementation and integration is becoming blurred. In some cases, there is no need to develop new hardware or software. Essentially, systems integration is the implementation phase of the system.

During and after the integration process, the system is tested. This testing should focus on testing the interfaces between components and the behavior of the system as a whole. Inevitably, testing also reveals problems with individual subsystems that have to be repaired. Testing takes a long time, and a common problem in system development is that the testing team may run out of either budget or time. This problem can lead to the delivery of error-prone systems that need be repaired after they have been deployed.

Subsystem faults that are a consequence of invalid assumptions about other subsystems are often exposed during system integration. This may lead to disputes between the contractors responsible for implementing different subsystems. When problems are discovered in subsystem interaction, the contractors may argue about which subsystem is faulty. Negotiations on how to solve the problems can take weeks or months.

The final stage of the system development process is system delivery and deployment. The software is installed on the hardware and is readied for operation. This may

involve more system configuration to reflect the local environment where it is used, the transfer of data from existing systems, and the preparation of user documentation and training. At this stage, you may also have to reconfigure other systems in the environment to ensure that the new system interoperates with them.

Although system deployment is straightforward in principle, it is often more difficult than anticipated. The user environment may be different from that anticipated by the system developers. Adapting the system to make it work in an unexpected environment can be difficult. The existing system data may require extensive clean-up, and parts of it may involve more effort than expected. The interfaces to other systems may not be properly documented. You may find that the planned operational processes have to be changed because they are not compatible with the operational processes for other systems. User training is often difficult to arrange, with the consequence that, initially at least, users are unable to access the capabilities of the system. System deployment can therefore take much longer and cost much more than anticipated.

19.5 System operation and evolution

Operational processes are the processes that are involved in using the system as intended by its designers. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs, and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system. Undetected problems may arise at this stage because the system specification may contain errors or omissions. While the system may perform to specification, its functions may not meet the real operational needs. Consequently, the operators may not use the system as its designers intended.

Although the designers of operational processes may have based their process designs on extensive user studies, there is always a period of "domestication" (Stewart and Williams 2005) when users adapt to the new system and work out practical processes of how to use it. While user interface design is important, studies have shown that, given time, users can adapt to complex interfaces. As they become experienced, they prefer ways of using the system quickly rather than easily. This means that when designing systems, you should not simply cater for inexperienced users but you should design the user interface to be adaptable for experienced users.

Some people think that system operators are a source of problems in a system and that we should move toward automated systems where operator involvement is minimized. In my opinion, there are two problems with this approach:

1. It is likely to increase the technical complexity of the system because it has to be designed to cope with all anticipated failure modes. This increases the costs and

time required to build the system. Provision also has to be made to bring in people to deal with unanticipated failures.

2. People are adaptable and can cope with problems and unexpected situations. Thus, you do not have to anticipate everything that could possibly go wrong when you are specifying and designing the system.

People have a unique capability of being able to respond effectively to the unexpected, even when they have never had direct experience of these unexpected events or system states. Therefore, when things go wrong, the system operators can often recover the situation by finding workarounds and using the system in nonstandard ways. Operators also use their local knowledge to adapt and improve processes. Normally, the actual operational processes are different from those anticipated by the system designers.

Consequently, you should design operational processes to be flexible and adaptable. The operational processes should not be too constraining; they should not require operations to be done in a particular order; and the system software should not rely on a specific process being followed. Operators usually improve the process because they know what does and does not work in a real situation.

A problem that may only emerge after the system goes into operation is the operation of the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another. More subtle problems might arise because different systems have different user interfaces. Introducing a new system may increase the operator error rate, as the operators use user interface commands for the wrong system.

19.5.1 System evolution

Large, complex systems usually have a long lifetime. Complex hardware/software systems may remain in use for more than 20 years, even though both the original hardware and software technologies used are obsolete. There are several reasons for this longevity, as shown in Figure 19.12.

Over their lifetime, large complex systems change and evolve to correct errors in the original system requirements and to implement new requirements that have emerged. The system's computers are likely to be replaced with new, faster machines. The organization that uses the system may reorganize itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system. Hence, evolution is a process that runs alongside normal system operational processes. System evolution involves reentering the development process to make changes and extensions to the system's hardware, software, and operational processes.

System evolution, like software evolution (discussed in Chapter 9), is inherently costly for several reasons:

1. Proposed changes have to be analyzed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.

Factor	Rationale
Investment cost	The costs of a systems engineering project may be tens or even hundreds of millions of dollars. These costs can only be justified if the system can deliver value to an organization for many years.
Loss of expertise	As businesses change and restructure to focus on their core activities, they often lose engineering expertise. This may mean that they lack the ability to specify the requirements for a new system.
Replacement cost	The cost of replacing a large system is very high. Replacing an existing system can be justified only if this leads to significant cost savings over the existing system.
Return on investment	If a fixed budget is available for systems engineering, spending on new systems in some other area of the business may lead to a higher return on investment than replacing an existing system.
Risks of change	Systems are an inherent part of business operations, and the risks of replacing existing systems with new systems cannot be justified. The danger with a new system is that things can go wrong in the hardware, software, and operational processes. The potential costs of these problems for the business may be so high that they cannot take the risk of system replacement.
System dependencies	Systems are interdependent and replacing one of these systems may lead to extensive changes in other systems.

Figure 19.12 Factors that influence system lifetimes

2. Because subsystems are never completely independent, changes to one subsystem may have side-effects that adversely affect the performance or behavior of other subsystems. Consequent changes to these subsystems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure becomes corrupted by change, so the costs of making further changes increases.

Systems that have been in use for many years are often reliant on obsolete hardware and software technology. These "legacy systems" (discussed in Chapter 9) are sociotechnical computer-based systems that have been developed using technology that is now obsolete. However, they don't just include legacy hardware and software. They also rely on legacy processes and procedures—old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components.

Changes made to a system during system evolution are often a source of problems and vulnerabilities. If the people implementing the changes are different from those who developed the system, they may be unaware that a design decision was taken for dependability and security reasons. Therefore, they may change the system and lose some safeguards that were deliberately implemented when the system was built. Furthermore, as testing is so expensive, complete retesting may be impossible after every system change. Consequently, testing may not discover the adverse side-effects of changes that introduce or expose faults in other system components.

KEY POINTS

- Systems engineering is concerned with all aspects of specifying, buying, designing, and testing complex sociotechnical systems.
- Sociotechnical systems include computer hardware, software, and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- The emergent properties of a system are characteristics of the system as a whole rather than of its component parts. They include properties such as performance, reliability, usability, safety, and security.
- The fundamental systems engineering processes are conceptual systems design, system procurement, system development, and system operation.
- Conceptual systems design is a key activity where high-level system requirements and a vision of the operational system is developed.
- System procurement covers all of the activities involved in deciding what system to buy and who should supply that system. Different procurement processes are used for off-the-shelf application systems, configurable COTS systems, and custom systems.
- System development processes include requirements specification, design, construction, integration, and testing.
- When a system is put into use, the operational processes and the system itself inevitably change to reflect changes to the business requirements and the system's environment.

FURTHER READING

"Airport 95: Automated Baggage System." An excellent, readable case study of what can go wrong with a systems engineering project and how software tends to get the blame for wider systems failures. (*ACM Software Engineering Notes*, 21, March 1996). http://doi.acm.org/10.1145/227531.227544

"Fundamentals of Systems Engineering." This is the introductory chapter in NASA's systems engineering handbook. It presents an overview of the systems engineering process for space systems. Although these are mostly technical systems, there are sociotechnical issues to be considered. Dependability is obviously critically important. (In *NASA Systems Engineering Handbook*, NASA-SP 2007-6105, 2007). http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080008301_2008008500.pdf

The LSCITS Socio-technical Systems Handbook. This handbook introduces sociotechnical systems in an accessible way and provides access to more detailed papers on sociotechnical topics. (Various authors, 2012). http://archive.cs.st-andrews.ac.uk/STSE-Handbook

Architecting systems: Concepts, Principles and Practice. This is a refreshingly different book on systems engineering that does not have the hardware focus of many "traditional" systems engineering books.

The author, who is an experienced systems engineer, draws on examples from a wide range of systems and recognizes the importance of sociotechnical as well as technical issues. (H. Sillitto, College Publications, 2014).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

19.1. Give two examples of government functions that are supported by complex sociotechnical systems and explain why, in the foreseeable future, these functions cannot be completely automated.

19.2. Explain briefly why the involvement of a range of professional disciplines is essential in systems engineering.

19.3. Complex sociotechnical systems lead to three important characteristics. What are they? Explain each in brief.

19.4. What is a "wicked problem"? Explain why the development of a national medical records system should be considered a "wicked problem."

19.5. A multimedia virtual museum system offering virtual experiences of ancient Greece is to be developed for a consortium of European museums. The system should provide users with the facility to view 3-D models of ancient Greece through a standard web browser and should also support an immersive virtual reality experience. Develop a conceptual design for such a system, highlighting its key characteristics and essential high-level requirements.

19.6. Explain why you need to be flexible and adapt system requirements when procuring large off-the-shelf software systems, such as ERP systems. Search the web for discussions of the failures of such systems and explain, from a sociotechnical perspective, why these failures occurred. A possible starting point is: http://blog.360cloudsolutions.com/blog/bid/94028/Top-Six-ERP-Implementation-Failures

19.7. Why is system integration a particularly critical part of the systems development process? Suggest three sociotechnical issues that may cause difficulties in the system integration process.

19.8. Why is system evolution inherently costly?

19.9. What are the arguments for and against considering system engineering as a profession in its own right, like electrical engineering or software engineering?

19.10. You are an engineer involved in the development of a financial system. During installation, you discover that this system will make a significant number of people redundant. The people in the environment deny you access to essential information to complete the system installation. To what extent should you, as a systems engineer, become involved in this situation? Is it your professional responsibility to complete the installation as contracted? Should you simply abandon the work until the procuring organization has sorted out the problem?

REFERENCES

Baxter, G., and I. Sommerville. 2011. "Socio-Technical Systems: From Design Methods to Systems Engineering." *Interacting with Computers* 23 (1): 4–17. doi:10.1016/j.intcom.2010.07.003.

Checkland, P. 1981. *Systems Thinking, Systems Practice*. Chichester, UK: John Wiley & Sons.

Fairley, R. E., R. H. Thayer, and P. Bjorke. 1994. "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications." In *1st Int. Conf. on Requirements Engineering,* 40–7. Colorado Springs, CO. doi:10.1109/ICRE.1994.292405.

IEEE. 2007. "IEEE Guide for Information Technology. System Definition—Concept of Operations (ConOps) Document." *Electronics*. Vol. 1998. doi:10.1109/IEEESTD.1998.89424. http://ieeexplore.ieee.org/servlet/opac?punumber=6166

Mostashari, A., S. A. McComb, D. M. Kennedy, R. Cloutier, and P. Korfiatis. 2012. "Developing a Stakeholder-Assisted Agile CONOPS Development Process." *Systems Engineering* 15 (1): 1–13. doi:10.1002/sys.20190.

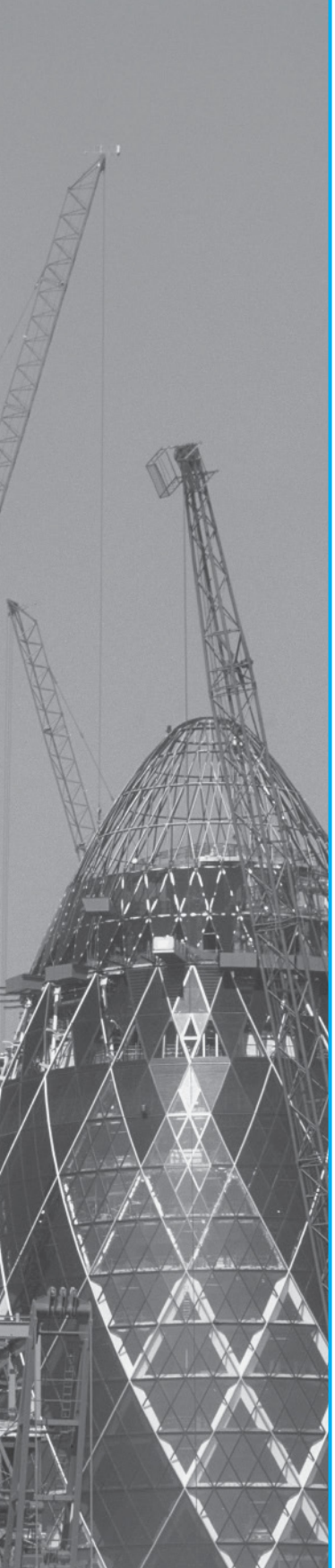
Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Stevens, R., P. Brook, K. Jackson, and S. Arnold. 1998. *Systems Engineering: Coping with Complexity.* London: Prentice-Hall.

Stewart, J., and R. Williams. 2005. "The Wrong Trousers? Beyond the Design Fallacy: Social Learning and the User." In *User Involvement in Innovation Processes. Strategies and Limitations from a Socio-Technical Perspective,* edited by H. Rohrache, 39–71. Berlin: Profil-Verlag.

Thayer, R. H. 2002. "Software System Engineering: A Tutorial." *IEEE Computer* 35 (4): 68–73. doi:10.1109/MC.2002.993773.

White, S., M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey. 1993. "Systems Engineering of Computer-Based Systems." *IEEE Computer* 26 (11): 54–65. doi:10.1109/ECBS.1994.331687.



20
Systems of systems

Objectives

The objectives of this chapter are to introduce the idea of a system of systems and to discuss the challenges of building complex systems of software systems. When you have read this chapter, you will:

- understand what is meant by a system of systems and how it differs from an individual system;
- understand systems of systems classification and the differences between different types of systems of systems;
- understand why conventional methods of software engineering that are based on reductionism are inadequate for developing systems of systems;
- have been introduced to the systems of systems engineering process and architectural patterns for systems of systems.

Contents

20.1 System complexity
20.2 Systems of systems classification
20.3 Reductionism and complex systems
20.4 Systems of systems engineering
20.5 Systems of systems architecture

We need software engineering because we create large and complex software systems. The discipline emerged in the 1960s because the first attempts to build large software systems mostly went wrong. Creating software was much more expensive than expected, took longer than planned, and the software itself was often unreliable. To address these problems, we have developed a range of software engineering techniques and technologies, which have been remarkably successful. We can now build systems that are much larger, more complex, much more reliable, and more effective than the software systems of the 1970s.

However, we have not "solved" the problems of large system engineering. Software project failures are still common. For example, there have been serious problems and delays in the implementation of government health care systems in both the United States and the UK. The root cause of these problems is, as it was in the 1960s, that we are trying to build systems that are larger and more complex than before. We are attempting to build these "mega-systems" using methods and technology that were never designed for this purpose. As I discuss later in the chapter, I believe that current software engineering technology cannot scale up to cope with the complexity that is inherent in many of the systems now being proposed.

The increase in size of software systems since the introduction of software engineering has been remarkable. Today's large systems may be a hundred or even a thousand times larger than the "large" systems of the 1960s. Northrop and her colleagues (Northrop et al. 2006) suggested in 2006 that we would shortly see the development of systems with a billion lines of code. Almost 10 years after this prediction, I suspect such systems are already in use.

Of course, we do not start with nothing and then write a billion lines of code. As I discussed in Chapter 15, the real success story of software engineering has been software reuse. It is only because we have developed ways of reusing software across applications and systems that large-scale development is possible. Very large-scale systems now and in the future will be built by integrating existing systems from different providers to create systems of systems (SoS).

What do we mean when we talk about a system of systems? As Hitchens says (Hitchins 2009), from a general systems perspective, there is no difference between a system and a system of systems. Both have emergent properties and can be composed from subsystems. However, from a software engineering perspective, I think there is a useful distinction between these terms. This distinction is sociotechnical rather than technical:

> *A system of systems is a system that contains two or more independently managed elements.*

This means that there is no single manager for all of the parts of the system of systems and that different parts of a system are subject to different management and control policies and rules. As we shall see, distributed management and control has a profound effect on the overall complexity of the system.

This definition of systems of systems says nothing about the size of systems of systems. A relatively small system that includes services from different providers is

a system of systems. Some of the problems of SoS engineering apply to such small systems, but the real challenges emerge when the constituent systems are themselves large-scale systems.

Much of the work in the area of systems of systems has come from the defense community. As the capability of software systems increased in the late 20th century, it became possible to coordinate and control previously independent military systems, such as naval and ground-based air and ship defense systems. The system might include tens or hundreds of separate elements, with software systems keeping track of these elements and providing controllers with information that allows them to be deployed most effectively.

This type of system of systems is outside the scope of a software engineering book. Instead, I focus here on systems of systems where the system elements are software systems rather than hardware such as aircraft, military vehicles, or radars. Systems of software systems are created by integrating separate software systems, and, at the time of writing, most software SoS include a relatively small number of separate systems. Each constituent system is usually a complex system in its own right. However, it is predicted that, over the next few years, the size of software SoS is likely to grow significantly as more and more systems are integrated to make use of the capabilities that they offer.

Examples of systems of systems of software systems are:

1. A cloud management system that handles local private cloud management and management of servers on public clouds such as Amazon and Microsoft.
2. An online banking system that handles loan requests and that connects to a credit reference system provided by credit reference agencies to check the credit of applicants.
3. An emergency information system that integrates information from police, ambulance, fire, and coast guard services about the assets available to deal with civil emergencies such as flooding and large-scale accidents.
4. The digital learning environment (iLearn) that I introduced in Chapter 1. This system provides a range of learning support by integrating separate software systems such as Microsoft Office 365, virtual learning environments such as Moodle, simulation modeling tools, and content such as newspaper archives.

Maier (Maier 1998) identified five essential characteristics of systems of systems:

1. *Operational independence of elements* Parts of the system are not simply components but can operate as useful systems in their own right. The systems within the SoS evolve independently.
2. *Managerial independence of elements* Parts of the system are "owned" and managed by different organizations or by different parts of a larger organization. Therefore different rules and policies apply to the management and evolution of

these systems. As I have suggested, this is the key factor that distinguishes a system of systems from a system.

3. *Evolutionary development* SoS are not developed in a single project but evolve over time from their constituent systems.

4. *Emergence* SoS have emergent characteristics that only become apparent after the SoS has been created. Of course, as I have discussed in Chapter 19, emergence is a characteristic of all systems, but it is particularly important in SoS.

5. *Geographical distribution of elements* The elements of a SoS are often geographically distributed across different organizations. This is important technically because it means that an externally-managed network is an integral part of the SoS. It is also important managerially as it increases the difficulties of communication between those involved in making system management decisions and adds to the difficulties of maintaining system security.[†]

I would like to add two further characteristics to Maier's list that are particularly relevant to systems of software systems:

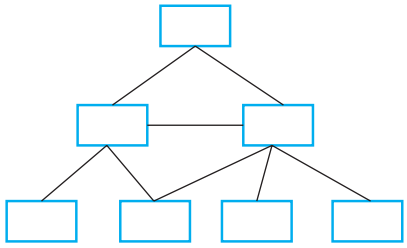
1. *Data intensive* A software SoS typically relies on and manages a very large volume of data. In terms of size, this may be tens or even hundreds of times larger than the code of the constituent systems itself.

2. *Heterogeneity* The different systems in a software SoS are unlikely to have been developed using the same programming languages and design methods. This is a consequence of the very rapid pace of evolution of software technologies. Companies frequently update their development methods and tools as new, improved versions become available. In a 20-year lifetime of a large SoS, technologies may change four or five times.

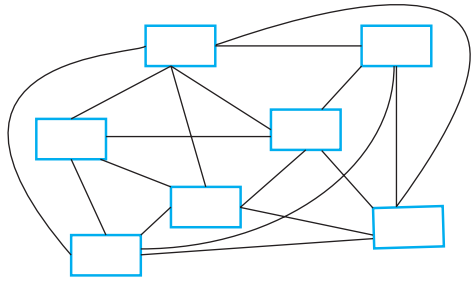
As I discuss in Section 20.1, these characteristics mean that SoS can be much more complex than systems with a single owner and manager. I believe that our current software engineering methods and techniques cannot scale to cope with this complexity. Consequently, problems with the very large and complex systems that we are now developing are inevitable. We need a completely new set of abstractions, methods, and technologies for software systems of systems engineering.

This need has been recognized independently by a number of different authorities. In the UK, a report published in 2004 (Royal Academy of Engineering 2004) led to the establishment of a national research and training initiative in large-scale complex IT systems (Sommerville et al. 2012). In the United States, the Software Engineering Institute reported on Ultra-Large Scale Systems in 2006 (Northrop et al. 2006). From the systems engineering community, Stevens (Stevens 2010) discusses the problems of constructing "mega-systems" in transport, health care, and defense.

[†]Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." Systems Engineering 1 (4): 267–284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.



System (a)



System (b)

Figure 20.1 Simple and complex systems

20.1 System complexity

I suggested in the introduction that the engineering problems that arise when constructing systems of software systems are due to the inherent complexity of these systems. In this section, I explain the basis of system complexity and discuss the different types of complexity that arise in software SoS.

All systems are composed of parts (elements) with relationships between these elements of the system. For example, the parts of a program may be objects, and the parts of each object may be constants, variables, and methods. Examples of relationships include "calls" (method A calls method B), "inherits-from" (object X inherits the methods and attributes of object Y), and "part of" (method A is part of object X).

The complexity of any system depends on the number and types of relationships between system elements. Figure 20.1 shows examples of two systems. System (a) is a relatively simple system with only a small number of relationships between its elements. By contrast, System (b), with the same number of elements, is a more complex system because it has many more element–element relationships.

The type of relationship also influences the overall complexity of a system. Static relationships are relationships that are planned and analyzable from static depictions of the system. Therefore, the "uses" relationship in a software system is a static relationship. From either the software source code or a UML model of a system, you can work out how any one software component uses other components.

Dynamic relationships are relationships that exist in an executing system. The "calls" relationship is a dynamic relationship because, in any system with if-statements, you cannot tell whether or not one method will call another method. It depends on the runtime inputs to the system. Dynamic relationships are more complex to analyze as you need to know the system inputs and data used as well as the source code of the system.

As well as system complexity, we also have to consider the complexity of the processes used to develop and maintain the system once it has gone into use. Figure 20.2 illustrates these processes and their relationship with the developed system.

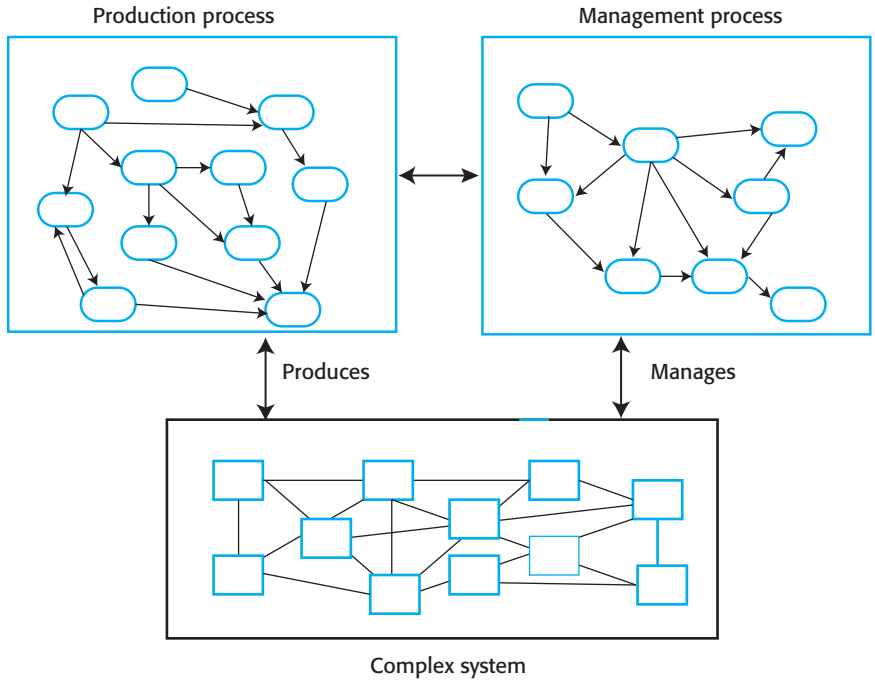


Figure 20.2 Production and management processes

As systems grow in size, they need more complex production and management processes. Complex processes are themselves complex systems. They are difficult to understand and may have undesirable emergent properties. They are more time consuming than simpler processes, and they require more documentation and coordination between the people and the organizations involved in the system development. The complexity of the production process is one of the main reasons why projects go wrong, with software delivered late and overbudget. Therefore, large systems are always at risk of cost and time overruns.

Complexity is important for software engineering because it is the main influence on the understandability and the changeability of a system. The more complex a system, the more difficult it is to understand and analyze. Given that complexity is a function of the number of relationships between elements of a system, it is inevitable that large systems are more complex than small systems. As complexity increases, there are more and more relationships between elements of the system and an increased likelihood that changing one part of a system will have undesirable effects elsewhere.

Several different types of complexity are relevant to sociotechnical systems:

1. The *technical complexity* of the system is derived from the relationships between the different components of the system itself.
2. The *managerial complexity* of the system is derived from the complexity of the relationships between the system and its managers (i.e., what can managers change in the system) and the relationships between the managers of different parts of the system.

3. The *governance complexity* of a system depends on the relationships between the laws, regulations, and policies that affect the system and the relationships between the decision-making processes in the organizations responsible for the system. As different parts of the system may be in different organizations and in different countries, different laws, rules, and policies may apply to each system within the SoS.

Governance and managerial complexity are related, but they are not the same thing. Managerial complexity is an operational issue—what can and can't actually be done with the system. Governance complexity is associated with the higher level of decision-making processes in organizations that affect the system. These decision-making processes are constrained by national and international laws and regulations.

For example, say a company decides to allow its staff to access its systems using their own mobile devices rather than company-issued laptops. The decision to allow this is a governance decision because it changes the policy of the company. As a result of this decision, management of the system becomes more complex as managers have to ensure that the mobile devices are configured properly so that company data is secure. The technical complexity of the system also increases as there is no longer a single implementation platform. Software may have to be modified to work on laptops, tablets and phones.

As well as technical complexity, the characteristics of systems of systems may also lead to significantly increased managerial and governance complexity. Figure 20.3 summarizes how the different SoS characteristics primarily contribute to different types of complexity:

1. *Operational independence* The constituent systems in the SoS are subject to different policies and rules (governance complexity) and ways of managing the system (managerial complexity).
2. *Managerial independence* The constituent systems in the SoS are managed by different people in different ways. They have to coordinate to ensure that management changes are consistent (managerial complexity). Special software may be needed to support consistent management and evolution (technical complexity).
3. Evolutionary development contributes to the technical complexity of a SoS because different parts of the system are likely to be built using different technologies.
4. Emergence is a consequence of complexity. The more complex a system, the more likely it is that it will have undesirable emergent properties. These properties increase the technical complexity of the system as software has to be developed or changed to compensate for them.
5. Geographical distribution increases the technical, managerial, and governance complexity in a SoS. Technical complexity is increased because software is required to coordinate and synchronize remote systems; managerial complexity is increased because it is more difficult for managers in different countries to coordinate their actions; governance complexity is increased because different

SoS characteristic	Technical complexity	Managerial complexity	Governance complexity
Operational independence		X	X
Managerial independence	X	X	
Evolutionary development	X		
Emergence	X		
Geographical distribution	X	X	X
Data-intensive	X		X
Heterogeneity	X		

Figure 20.3 SoS characteristics and system complexity

parts of the systems may be located in different jurisdictions and so are subject to different laws and regulations.

6. Data-intensive systems are technically complex because of the relationships between the data items. The technical complexity is also likely to be increased to cope with data errors and incompleteness. Governance complexity may be increased because of different laws governing the use of data.

7. The heterogeneity of a system contributes to its technical complexity because of the difficulties of ensuring that different technologies used in different parts of the system are compatible.

Large-scale systems of systems are now unimaginably complex entities that cannot be understood or analyzed as a whole. As I discuss in Section 20.3, the large number of interactions between the parts and the dynamic nature of these interactions means that conventional engineering approaches do not work well for complex systems. It is complexity that is the root cause of problems in projects to develop large software-intensive systems, not poor management or technical failings.

20.2 Systems of systems classification

Earlier, I suggested that the distinguishing feature of a system of systems was that two or more of its elements were independently managed. Different people with different priorities have the authority to take day-to-day operational decisions about changes to the system. As their work is not necessarily aligned, conflicts can arise that require a significant amount of time and effort to resolve. Systems of systems, therefore, always have some degree of managerial complexity.

However, this broad definition of SoS covers a very wide range of system types. It includes systems that are owned by a single organization but are managed by different

parts of that organization. It also includes systems whose constituent systems are owned and managed by different organizations that may, at times, compete with each other. Maier (Maier 1998) devised a classification scheme for SoS based on their governance and management complexity:

1. *Directed systems.* Directed SoS are owned by a single organization and are developed by integrating systems that are also owned by that organization. The system elements may be independently managed by parts of the organization. However, there is an ultimate governing body within the organization that can set priorities for system management. It can resolve disputes between the managers of different elements of the system. Directed systems therefore have some managerial complexity but no governance complexity. A military command-and-control system that integrates information from airborne and ground-based systems is an example of a directed SoS.
2. *Collaborative systems.* Collaborative SoS are systems with no central authority to set management priorities and resolve disputes. Typically, elements of the system are owned and governed by different organizations. However, all of the organizations involved recognize the mutual benefits of joint governance of the system. They therefore usually set up a voluntary governance body that makes decisions about the system. Collaborative systems have both managerial complexity and a limited degree of governance complexity. An integrated public transport information system is an example of a collaborative system of systems. Bus, rail, and air transport providers agree to link their systems to provide passengers with up-to-date information.
3. *Virtual systems.* Virtual systems have no central governance, and the participants may not agree on the overall purpose of the system. Participant systems may enter or leave the SoS. Interoperability is not guaranteed but depends on published interfaces that may change. These systems have a very high degree of both managerial and governance complexity. An example of a virtual SoS is an automated high-speed algorithmic trading system. These systems from different companies automatically buy and sell stock from each other, with trades taking place in fractions of a second.

Unfortunately, I think that the names that Maier has used do not really reflect the distinctions between these different types of systems. As Maier himself says, there is always some collaboration in the management of the system elements. So, "collaborative systems" is not really a good name. The term *directed systems* implies top-down authority. However, even within a single organization, the need to maintain good working relationships between the people involved means that governance is agreed to rather than imposed.

In "virtual" SoS, there may be no formal mechanisms for collaboration, but the system has some mutual benefit for all participants. Therefore, they are likely to collaborate informally to ensure that the system can continue to operate. Furthermore, Maier's use of the term *virtual* could be confusing because "virtual" has now come to mean "implemented by software," as in virtual machines and virtual reality.

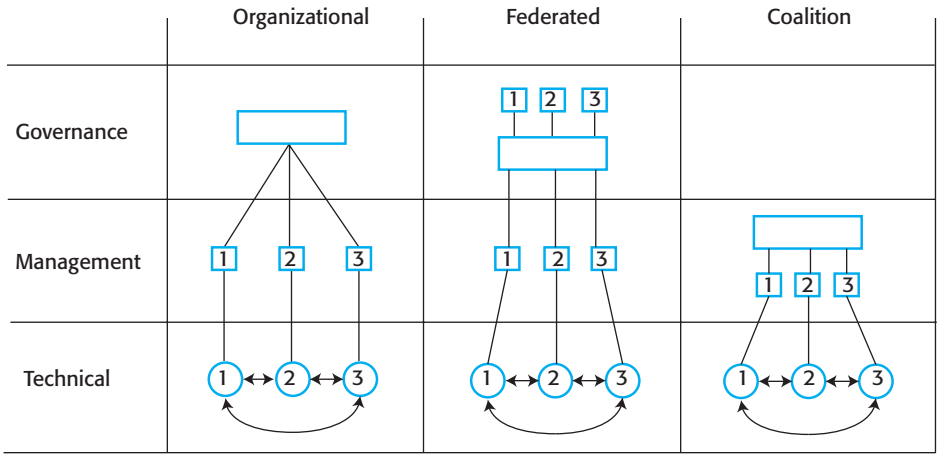


Figure 20.4 SoS collaboration

Figure 20.4 illustrates the collaboration in these different types of system. Rather than use Maier's names, I have used what I hope are more descriptive terms:

1. *Organizational systems of systems* are SoS where the governance and management of the system lies within the same organization or company. These correspond to Maier's "directed SoS." Collaboration between system owners is managed by the organization. The SoS may be geographically distributed, with different parts of the system subject to different national laws and regulations. In Figure 20.4, Systems 1, 2, and 3 are independently managed, but the governance of these systems is centralized.
2. *Federated systems* are SoS where the governance of the SoS depends on a voluntary participative body in which all of the system owners are represented. In Figure 20.4, this is shown by the owners of Systems 1, 2, and 3 participating in a single governance body. The system owners agree to collaborate and believe that decisions made by the governance body are binding. They implement these decisions in their individual management policies, although implementations may differ because of national laws, regulations, and culture.
3. *System of system coalitions* are SoS with no formal governance mechanisms but where the organizations involved informally collaborate and manage their own systems to maintain the system as a whole. For example, if one system provides a data feed to others, the managers of that system will not change the format of the data without notice. Figure 20.4 shows that there is no governance at the organizational level but that informal collaboration exists at the management level.

This governance-based classification scheme provides a means of identifying the governance requirements for a SoS. By classifying a system according to this model, you can check if the appropriate governance structures exist and if these are the ones you really need. Setting up these structures across organizations is a political process and inevitably takes a long time. It is therefore helpful to understand the governance

problem early in the process and take actions to ensure that appropriate governance is in place. It may be the case that you need to adopt a governance model that moves a system from one class to another. Moving the governance model to the left in Figure 20.4 usually reduces complexity.

As I have suggested, the school digital learning environment (iLearn) is a system of systems. As well as the digital learning system itself, it is connected to school administration systems and to network management systems. These network management systems are used for Internet filtering, which stops students from accessing undesirable material on the Internet.

iLearn is a relatively simple technical system, but it has a high level of governance complexity. This complexity arises because of the way that education is funded and managed. In many countries pre-university education is funded and organized at a local level rather than at a national level. States, cities, or counties are responsible for schools in their area and have autonomy in deciding school funding and policies. Each local authority maintains its own school administration system and network management system.

In Scotland, there are 32 local authorities with responsibility for education in their area. School administration is outsourced to one of three providers and iLearn must connect to their systems. However, each local authority has its own network management policies with separate network management systems involved.

The development of a digital learning system is a national initiative, but to create a digital learning environment, it has to be integrated with network management and school administration systems. It is therefore a system of systems with administration and network management systems, as well as the systems within iLearn such as Office 365 and Wordpress. There is no common governance process across authorities, so, according to the classification scheme, this is a coalition of systems. In practice, this means that it cannot be guaranteed that students in different places can access the same tools and content, because of different Internet filtering policies.

When we produced the conceptual model for the system, we made a strong recommendation that common policies should be established across local authorities on administrative information provision and Internet filtering. In essence, we suggested that the system should be a federated system rather than a coalition of systems. This suggestion requires a new governance body to be established to agree on common policies and standards for the system.

20.3 Reductionism and complex systems

I have already suggested that our current software engineering methods and technologies cannot cope with the complexity that is inherent in modern systems of systems. Of course, this idea is not new: Progress in all engineering disciplines has always been driven by challenging and difficult problems. New methods and tools are developed in response to failures and difficulties with existing approaches.

In software engineering, we have seen the incredibly rapid development of the discipline to help manage the increasing size and complexity of software systems. This effort has been very successful indeed. We can now build systems that are orders of magnitude larger and more complex than those of the 1960s and 1970s.

As with other engineering disciplines, the approach that has been the basis of complexity management in software engineering is called *reductionism*. Reductionism is a philosophical position based on the assumptions that any system is made up of parts or subsystems. It assumes that the behavior and properties of the system as a whole can be understood and predicted by understanding the individual parts and the relationships between these parts. Therefore, to design a system, the parts making up that system are identified, constructed separately, and then assembled into the complete system. Systems can be thought of as hierarchies, with the important relationships between parent and child nodes in the hierarchy.

Reductionism has been and continues to be the fundamental underpinning approach to all kinds of engineering. We can identify common abstractions across the same types of system and design and build these separately. They can then be integrated to create the required system. For example, the abstractions in an automobile might be a body shell, a drive train, an engine, a fuel system, and so on. There are a relatively small number of relationships between these abstractions, so it is possible to specify interfaces and design and build each part of the system separately.

The same reductionist approach has been the basis of software engineering for almost 50 years. Top-down design, where you start with a very high-level model of a system and break this down to its components is a reductionist approach. This is the basis of all software design methods, such as object-oriented design. Programming languages include abstractions, such as procedures and objects that directly reflect reductionist system decomposition.

Agile methods, although they may appear quite different from top-down systems design, are also reductionist. They rely on being able to decompose a system into parts, implement these parts separately, and then integrate these to create the system. The only real difference between agile methods and top-down design is that the system is decomposed into components incrementally rather than all at once.

Reductionist methods are most successful when there are relatively few relationships or interactions between the parts of a system and it is possible to model these relationships in a scientific way. This is generally true for mechanical and electrical systems where there are physical linkages between the system components. It is less true for electronic systems and certainly not the case for software systems, where there may be many more static and dynamic relationships between system components.

The distinctions between software and hardware components was recognized in the 1970s. Design methods emphasized the importance of limiting and controlling the relationships between the parts of a system. These methods suggested that components should be tightly integrated with loose coupling between these components. Tight integration meant that most of the relationships were internal to a component, and loose coupling meant that there were relatively few component–component

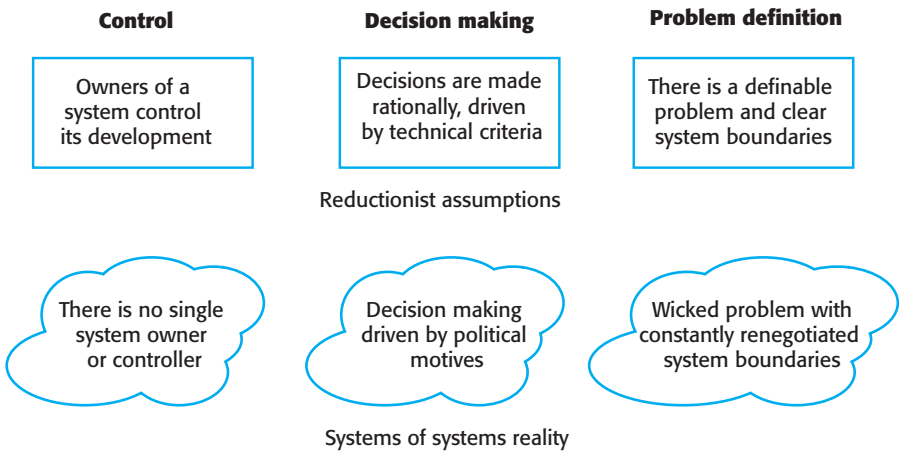


Figure 20.5 Reductionist assumptions and complex system reality

relationships. The need for tight integration (data and operations) and loose coupling was the driver for the development of object-oriented software engineering.

Unfortunately, controlling the number and types of relationship is practically impossible in large systems, especially systems of systems. Reductionism does not work well when there are many relationships in a system and when these relationships are difficult to understand and analyze. Therefore, any type of large system development is likely to run into difficulties.

The reasons for these potential difficulties are that the fundamental assumptions inherent to reductionism are inapplicable for large and complex systems (Sommerville et al. 2012). These assumptions are shown in Figure 20.5 and apply in three areas:

1. *System ownership and control* Reductionism assumes that there is a controlling authority for a system that can resolve disputes and make high-level technical decisions that will apply across the system. As we have seen, because there are multiple bodies involved in their governance, this is simply not true for systems of systems.

2. *Rational decision making* Reductionism assumes that interactions between components can be objectively assessed by, for example, mathematical modeling. These assessments are the driver for system decision making. Therefore, if one particular design of a vehicle, say, offers the best fuel economy without a reduction in power, then a reductionist approach assumes that this will be the design chosen.

3. *Defined system boundaries* Reductionism assumes that the boundaries of a system can be agreed to and defined. This is often straightforward: There may be a physical shell defining the system as in a car, a bridge has to cross a given stretch of water, and so on. Complex systems are often developed to address wicked problems (Rittel and Webber 1973). For such problems, deciding on what is part of the system and what is outside it is usually a subjective judgment, with frequent disagreements between the stakeholders involved.

These reductionist assumptions break down for all complex systems, but when these systems are software-intensive, the difficulties are compounded:

1. Relationships in software systems are not governed by physical laws. We cannot produce mathematical models of software systems that will predict their behavior and attributes. We therefore have no scientific basis for decision making. Political factors are usually the driver of decision making for large and complex software systems.
2. Software has no physical limitations; hence there are no limits on where the boundaries of a system should be drawn. Different stakeholders will argue for the boundaries to be placed in such a way that is best for them. Furthermore, it is much easier to change software requirements than hardware requirements. The boundaries and the scope of a system are likely to change during its development.
3. Linking software systems from different owners is relatively easy; hence we are more likely to try and create a SoS where there is no single governing body. The management and evolution of the different systems involved cannot be completely controlled.

For these reasons, I believe that the problems and difficulties that are commonplace in large software systems engineering are inevitable. Failures of large government projects such as the health automation projects in the UK and the United States are a consequence of complexity rather than technical or project management failures.

Reductionist approaches such as object-oriented development have been very successful in improving our ability to engineer many types of software system. They will continue to be useful and effective in developing small and medium-sized systems whose complexity can be controlled and which may be parts of a software SoS. However, because of the fundamental assumptions underlying reductionism, "improving" these methods will not lead to an improvement in our ability to engineer complex systems of systems. Rather, we need new abstractions, methods, and tools that recognize the technical, human, social, and political complexities of SoS engineering. I believe that these new methods will be probabilistic and statistical and that tools will rely on system simulation to support decision making. Developing these new approaches is a major challenge for software and systems engineering in the 21st century.

20.4 Systems of systems engineering

Systems of systems engineering is the process of integrating existing systems to create new functionality and capabilities. Systems of systems are not designed in a top-down way. Rather, they are created when an organization recognizes that they can add value to existing systems by integrating these into a SoS. For example, a city government might wish to reduce air pollution at particular hot-spots in the city. To do so, it might integrate its traffic management system with a national real-time pollution monitoring systems. This then allows for the traffic management system to alter its strategy to reduce pollution by changing traffic light sequences, speed limits and so on.

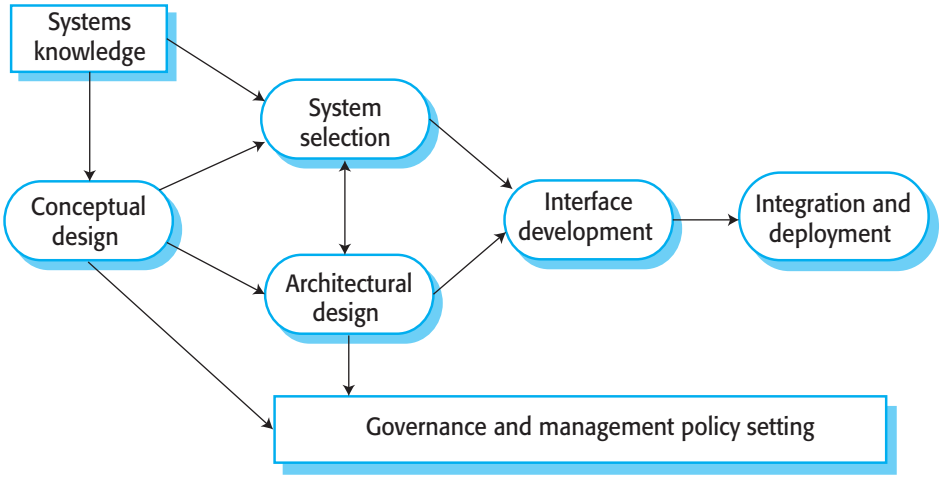


Figure 20.6 An SoS engineering process

The problems of software SoS engineering have much in common with the problems of integrating large-scale application systems that I discussed in Chapter 15 (Boehm and Abts 1999). To recap, these were:

1. Lack of control over system functionality and performance.
2. Differing and incompatible assumptions made by the developers of the different systems.
3. Different evolution strategies and timetables for the different systems.
4. Lack of support from system owners when problems arise.

Much of the effort in building systems of software systems comes from addressing these problems. It involves deciding on the system architecture, developing software interfaces that reconcile differences between the participating systems, and making the system resilient to unforeseen changes that may occur.

Software systems of systems are large and complex entities, and the processes used for their development vary widely depending on the type of systems involved, the application domain, and the needs of the organizations involved in developing the SoS. However, as shown in Figure 20.6, five general activities are involved in SoS development processes:

1. *Conceptual design* I introduced the idea of conceptual design in Chapter 19, which covers systems engineering. Conceptual design is the activity of creating a high-level vision for a system, defining essential requirements, and identifying constraints on the overall system. In SoS engineering, an important input to the conceptual design process is knowledge of the existing systems that may participate in the SoS.
2. *System selection* During this activity, a set of systems for inclusion in the SoS is chosen. This process is comparable to the process of choosing application

systems for reuse, covered in Chapter 15. You need to assess and evaluate existing systems to choose the capabilities that you need. When you are selecting application systems, the selection criteria are largely commercial; that is, which systems offer the most suitable functionality at a price you are prepared to pay?

However, political imperatives and issues of system governance and management are often the key factors that influence what systems are included in a SoS. For example, some systems may be excluded from consideration because an organization does not wish to collaborate with a competitor. In other cases, organizations that are contributing to a federation of systems may have systems in place and insist that these are used, even though they are not necessarily the best systems.

3. *Architectural design* In parallel with system selection, an overall architecture for the SoS has to be developed. Architectural design is a major topic in its own right that I cover in Section 20.5.
4. *Interface development* The different systems involved in a SoS usually have incompatible interfaces. Therefore, a major part of the software engineering effort in developing a SoS is to develop interfaces so that constituent systems can interoperate. This may also involve the development of a unified user interface so that SoS operators do not have to deal with multiple user interfaces as they use the different systems in the SoS.
5. *Integration and deployment* This stage involves making the different systems involved in the SoS work together and interoperate through the developed interfaces. System deployment means putting the system into place in the organizations concerned and making it operational.

In parallel with these technical activities, there needs to be a high-level activity concerned with establishing policies for the governance of the system of systems and defining management guidelines to implement these policies. Where there are several organizations involved, this process can be prolonged and difficult. It may involve organizations changing their own policies and processes. It is therefore important to start governance discussions at an early stage in the SoS development process.

20.4.1 Interface development

The constituent systems in a SoS are usually developed independently for some specific purpose. Their user interface is tailored to that original purpose. These systems may or may not have application programming interfaces (APIs) that allow other systems to interface directly to them. Therefore, when these systems are integrated into a SoS, software interfaces have to be developed, which allows the constituent systems in the SoS to interoperate.

In general, the aim in SoS development is for systems to be able to communicate directly with each other without user intervention. If these systems already offer a service-based interface, as discussed in Chapter 18, then this communication can be implemented using this approach. Interface development involves describing how to

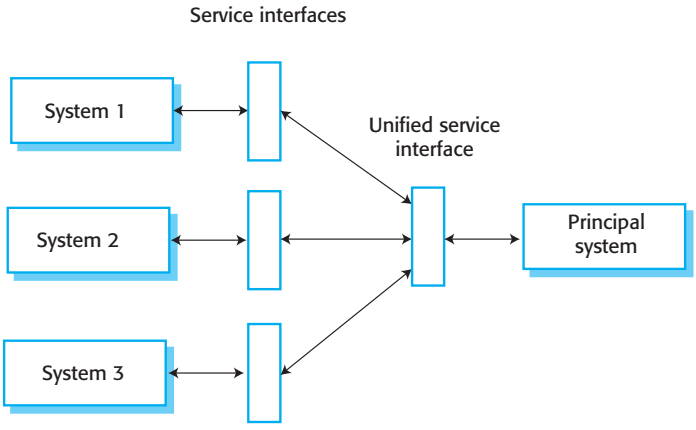


Figure 20.7 Systems with service interfaces

use the interfaces to access the functionality of each system. The systems involved can communicate directly with each other. System coalitions, where all of the systems involved are peers, are likely to use this type of direct interaction as it does not require prearranged agreements on system communication protocols.

More commonly, however, the constituent systems in a SoS either have their own specialized API or only allow their functionality to be accessed through their user interfaces. You therefore have to develop software that reconciles the differences between these interfaces. It is best to implement these interfaces as service-based interfaces, as shown in Figure 20.7 (Sillitto 2010).

To develop service-based interfaces, you have to examine the functionality of existing systems and define a set of services to reflect that functionality. The interface then provides these services. The services are implemented either by calls to the underlying system API or by mimicking user interaction with the system. One of the systems in the SoS is usually a principal or coordinating system that manages the interactions between the constituent systems. The principal system acts as a service broker, directing service calls between the different systems in the SoS. Each system therefore does not need to know which other system is providing a called service.

User interfaces for each system in a SoS are likely to be different. The principal system must have some overall user interfaces that handle user authentication and provide access to the features of the underlying system. However, it is usually expensive and time consuming to implement a unified user interface to replace the individual interfaces of the underlying systems.

A unified user interface (UI) makes it easier for new users to learn to use the SoS and reduces the likelihood of user error. However, whether or not unified UI development is cost-effective depends on a number of factors:

1. *The interaction assumptions of the systems in the SoS* Some systems may have a process-driven model of interaction where the system controls the interface and prompts the user for inputs. Others may give control to the user, so that the user chooses the sequence of interactions with the system. It is practically impossible to unify different interaction models.

2. *The mode of use of the SoS* In many cases, SoS are used in such a way that most of the interactions of users at a site are with one of the constituent systems. They use other systems only when additional information is required. For example, air traffic controllers may normally use a radar system for flight information and only access a flight plan database when additional information is required. A unified interface is a bad idea in these situations because it would slow down interaction with the most commonly used system. However, if the operators interact with all of the constituent systems, then a unified UI may be the best way forward.

3. *The "openness" of the SoS* If the SoS is open, so that new systems may be added to it when it is in use, then unified UI development is impractical. It is impossible to anticipate what the UI of new systems will be. Openness also applies to the organizations using the SoS. If new organizations can become involved, then they may have existing equipment and their own preferences for user interaction. They may therefore prefer not to have a unified UI.

In practice, the limiting factor in UI unification is likely to be the budget and time available for UI development. UI development is one of the most expensive systems engineering activities. In many cases, there is simply not enough project budget available to pay for the creation of a unified SoS user interface.

20.4.2 Integration and deployment

System integration and deployment are usually separate activities. A system is integrated from its components by an integration and testing team, validated, and then released for deployment. The components are managed so that changes are controlled and the integration team can be confident that the required version is included in the system. However, for SoS, such an approach may not be possible. Some of the component systems may already be deployed and in use, and the integration team cannot control changes to these systems.

For SoS, therefore, it makes sense to consider integration and deployment to be part of the same process. This approach reflects one of the design guidelines that I discuss in the following section, which is that an incomplete system of systems should be usable and provide useful functionality. The integration process should begin with systems that are already deployed, with new systems added to the SoS to provide coherent additions to the functionality of the overall system.

It often makes sense to plan the deployment of the SoS to reflect this, so that SoS deployment takes place in a number of stages. For example, Figure 20.8 illustrates a three-stage deployment process for the iLearn digital learning environment:

1. The initial deployment provides authentication, basic learning functionality, and integration with school administration systems.

2. Stage 2 of the deployment adds an integrated storage system and a set of more specialized tools to support subject-specific learning. These tools might include

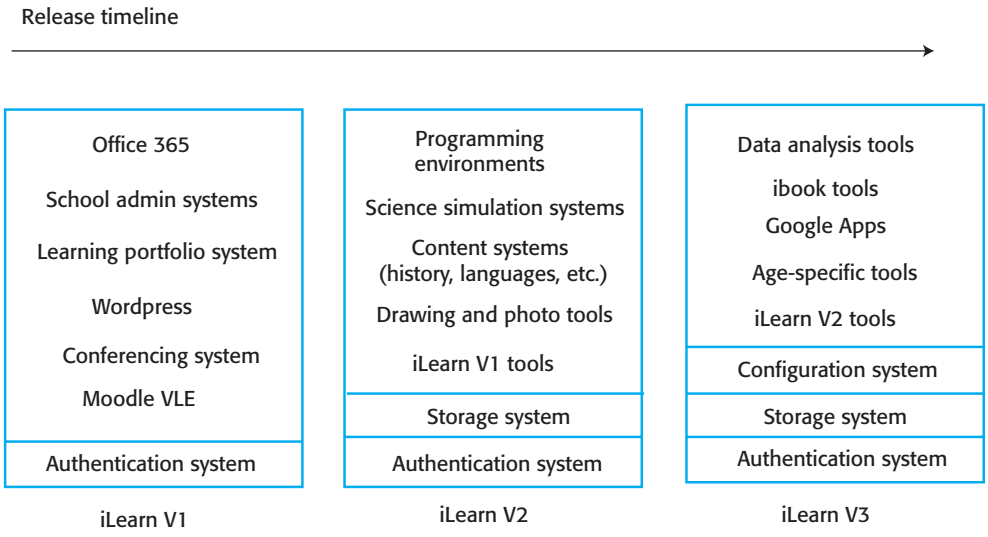


Figure 20.8 Release sequence For the iLearn SoS

archives for history, simulation systems for science, and programming environments for computing.

3. Stage 3 adds features for user configuration and the ability of users to add new systems to the iLearn environment. This stage allows different versions of the system to be created for different age groups, further specialized tools, and alternatives to the standard tools to be included.

As in any large systems engineering project, the most time-consuming and expensive part of system integration is system testing. Testing systems of systems is difficult and expensive for three reasons:

1. There may not be a detailed requirements specification that can be used as a basis for system testing. It may not be cost-effective to develop a SoS requirements document because the details of the system functionality are defined by the systems that are included.
2. The constituent systems may change in the course of the testing process, so tests may not be repeatable.
3. If problems are discovered, it may not be possible to fix the problems by requiring one or more of the constituent systems to be changed. Rather, some intermediate software may have to be introduced to solve the problem.

To help address some of these problems, I believe that SoS testing should take on board some of the testing techniques developed in agile methods:

1. Agile methods do not rely on having a complete system specification for system acceptance testing. Rather, stakeholders are closely engaged with the testing process

and have the authority to decide when the overall system is acceptable. For SoS, a range of stakeholders should be involved in the testing process if possible, and they can comment on whether or not the system is ready for deployment.

2. Agile methods make extensive use of automated testing. This makes it much easier to rerun tests to discover if unexpected system changes have caused problems for the SoS as a whole.

Depending on the type of system, you may have to plan the installation of equipment and user training as part of the deployment process. If the system is being installed in a new environment, equipment installation is straightforward. However, if it is intended to replace an existing system, there may be problems in installing new equipment if it is not compatible with the equipment that is in use. There may not be the physical space for the new equipment to be installed alongside the working system. There may be insufficient electrical power, or users may not have time to be involved because they are busy using the current system. These nontechnical issues can delay the deployment process and slow down the adoption and use of the SoS.

20.5 Systems of systems architecture

Perhaps the most crucial activity of the systems of systems engineering process is architectural design. Architectural design involves selecting the systems to be included in the SoS, assessing how these systems will interoperate, and designing mechanisms that facilitate interaction. Key decisions on data management, redundancy, and communications are made. In essence, the SoS architect is responsible for realizing the vision set out in the conceptual design of the system. For organizational and federated systems, in particular, decisions made at this stage are crucial to the performance, resilience, and maintainability of the system of systems.

Maier (Maier 1998) discusses four general principles for the architecting of complex systems of systems:

1. Design systems so that they can deliver value if they are incomplete. Where a system is composed of several other systems, it should not just be useful if all of its components are working properly. Rather, there should be several "stable intermediate forms" so that a partial system works and can do useful things.

2. Be realistic about what can be controlled. The best performance from a SoS may be achieved when an individual or group exerts control over the overall system and its constituents. If there is no control, then delivering value from the SoS is difficult. However, attempts to overcontrol the SoS are likely to lead to resistance from the individual system owners and consequent delays in system deployment and evolution.

3. Focus on the system interfaces. To build a successful system of systems, you have to design interfaces so that the system elements can interoperate. It is

important that these interfaces are not too restrictive so that the system elements can evolve and continue to be useful participants in the SoS.

4. Provide collaboration incentives. When the system elements are independently owned and managed, it is important each system owner have incentives to continue to participate in the system. These may be financial incentives (pay per use or reduced operational costs), access incentives (you share your data and I'll share mine), or community incentives (participate in a SoS and you get a say in the community).

Sillitto (Sillitto 2010) has added to these principles and suggests additional important design guidelines. These include the following:

1. Design a SoS as node and web architecture. Nodes are sociotechnical systems that include data, software, hardware, infrastructure (technical components), and organizational policies, people, processes, and training (sociotechnical). The web is not just the communications infrastructure between nodes, but it also provides a mechanism for informal and formal social communications between the people managing and running the systems at each node.
2. Specify behavior as services exchanged between nodes. The development of service-oriented architectures now provides a standard mechanism for system operability. If a system does not already provide a service interface, then this interface should be implemented as part of the SoS development process.
3. Understand and manage system vulnerabilities. In any SoS, there will be unexpected failures and undesirable behavior. It is critically important to try to understand vulnerabilities and design the system to be resilient to such failures.

The key message that emerges from both Maier's and Sillitto's work is that SoS architects have to take a broad perspective. They need to look at the system as a whole, taking into account both technical and sociotechnical considerations. Sometimes the best solution to a problem is not more software but changes to the rules and policies that govern the operation of the system.

Architectural frameworks such as MODAF (MOD 2008) and TOGAF (TOGAF is a registered trademark of The Open Group 2011) have been suggested as a means of supporting the architectural design of systems of systems. Architectural frameworks were originally developed to support enterprise systems architectures, which are portfolios of separate systems. Enterprise systems may be organizational systems of systems, or they may have a simpler management structure so that the system portfolio can be managed as a whole. Architectural frameworks are intended for the development of organizational systems of systems where there is a single governance authority for the entire SoS.

An architectural framework recognizes that a single model of an architecture does not present all of the information needed for architectural and business analysis. Rather, frameworks propose a number of architectural views that should be created and maintained to describe and document enterprise systems. Frameworks have much in common and tend to reflect the language and history of the organizations

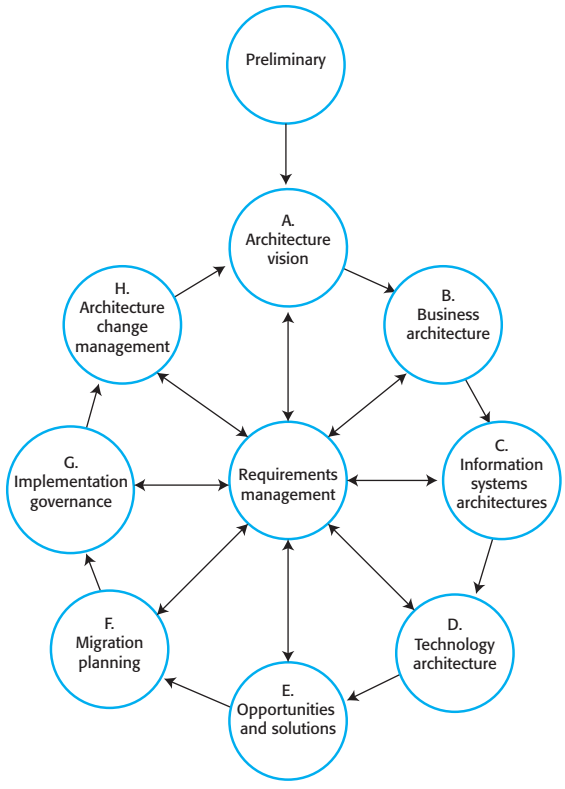


Figure 20.9 The TOGAF architecture development method (TOGAF® Version 9.1, © 1999–2011. The Open Group.)

involved. For example, MODAF and DODAF are comparable frameworks from the UK Ministry of Defence (MOD) and the U.S. Department of Defense (DOD).

The TOGAF framework has been developed by the Open Group as an open standard and is intended to support the design of a business architecture, a data architecture, an application architecture, and a technology architecture for an enterprise. At its heart is the Architecture Development Method (ADM), which consists of a number of discrete phases. These are shown in Figure 20.9, taken from the TOGAF reference documentation (Open Group 2011).

All architectural frameworks involve the production and management of a large set of architectural models. Each of the activities shown in Figure 20.8 leads to the production of system models. However, this is problematic for two reasons:

1. Initial model development takes a long time and involves extensive negotiations between system stakeholders. This slows the development of the overall system.
2. It is time-consuming and expensive to maintain model consistency as changes are made to the organization and the constituent systems in a SoS.

Architecture frameworks are fundamentally reductionist, and they largely ignore sociotechnical and political issues. While they do recognize that problems are difficult to define and are open-ended, they assume a degree of control and governance

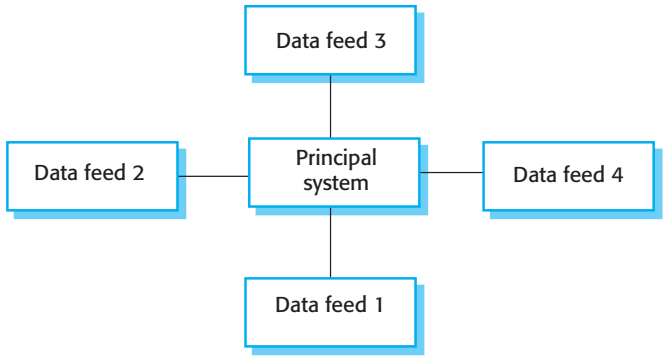


Figure 20.10 Systems as data feeds

that is impossible to achieve in many systems of systems. They are a useful checklist to remind architects of things to think about in the architectural design process. However, I think that the overhead involved in model management and the reductionist approach taken by frameworks limits their usefulness in SoS architectural design.

20.5.1 Architectural patterns for systems of systems

I have described architectural patterns for different types of system in Chapters 6, 17, and 21. In short, an architectural pattern is a stylized architecture that can be recognized across a range of different systems. Architectural patterns are a useful way of stimulating discussions about the most appropriate architecture for a system and for documenting and explaining the architectures used. This section covers a number of "typical" patterns in systems of software systems. As with all architectural patterns, real systems are usually based on more than one of these patterns.

The notion of architectural patterns for systems of systems is still at an early stage of development. Kawalsky (Kawalsky et al. 2013) discusses the value of architectural patterns in understanding and supporting SoS design, with a focus on patterns for command and control systems. I find that patterns are effective in illustrating SoS organization, without the need for detailed domain knowledge.

Systems as data-feeds

In this architectural pattern (Figure 20.10), there is a principal system that requires data of different types. This data is available from other systems, and the principal system queries these systems to get the data required. Generally, the systems that provide data do not interact with each other. This pattern is often observed in organizational or federated systems where some governance mechanisms are in place.

For example, to license a vehicle in the UK, you need to have both valid insurance and a roadworthiness certificate. When you interact with the vehicle licensing system, it interacts with two other systems to check that these documents are valid. These systems are:

1. An *"insured vehicles" system*, which is a federated system run by car insurance companies that maintains information about all current car insurance policies.

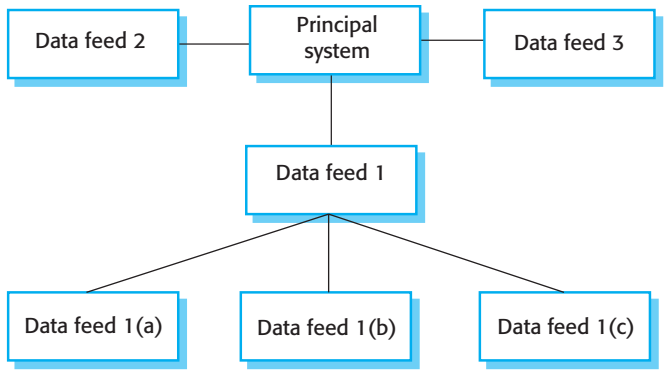


Figure 20.11 Systems as data feeds with a unifying interface

2. An "*MOT certificate" system*, which is used to record all roadworthiness certificates issued by testing agencies licensed by the government.

The "systems as data feeds" architecture is an appropriate architecture to use when it is possible to identify entities in a unique way and create relatively simple queries about these entities. In the licensing system, vehicles can be uniquely identified by their registration number. In other systems, it may be possible to identify entities such as pollution monitors by their GPS coordinates.

A variant of the "systems as data feeds" architecture arises when a number of systems provide data that are similar but not identical. Therefore, the architecture has to include an intermediate layer as shown in Figure 20.11. The role of this intermediate layer is to translate the general query from the principal system into the specific query required by the individual information system.

For example, the iLearn environment interacts with school administration systems from three different providers. All of these systems provide the same information about students (names, personal information, etc.) but have different interfaces. The databases have different organizations, and the format of the data returned differs from one system to another. The unifying interface here detects where the user of the system is based and, using this regional information, knows which administrative system should be accessed. It then converts a standard query into the appropriate query for that system.

Problems that can arise in systems that use this pattern are primarily interface problems when the data feeds are unavailable or are slow to respond. It is important to ensure that timeouts are included in the system so that a failure of a data feed does not compromise the response time of the system as a whole. Governance mechanisms should be in place to ensure that the format of provided data is not changed without the agreement of all system owners.

Systems in a container

Systems in a container are systems of systems where one of the systems acts as a virtual container and provides a set of common services such as an authentication and a storage service. Conceptually, other systems are then placed into this container

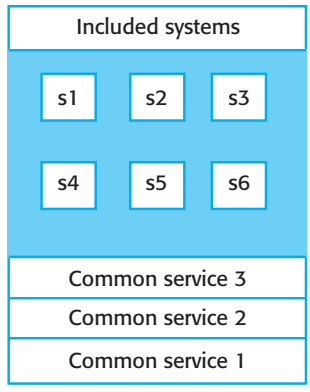


Figure 20.12 Systems in a container

to make their functionality accessible to system users. Figure 20.12 illustrates a container system with three common services and six included systems. The systems that are included may be selected from an approved list of systems and need not be aware that they are included in the container. This pattern of SoS is most often observed in federated systems or system coalitions.

The iLearn environment is a system in a container. There are common services that support authentication, storage of user data, and system configuration. Other functionality comes from choosing existing systems such as a newspaper archive or a virtual learning environment and integrating these into the container.

Of course, you don't place systems into a real container to implement these systems of systems. Rather, for each approved system, there is a separate interface that allows it to be integrated with the common services. This interface manages the translation of the common services provided by the container and the requirements of the integrated system. It may also be possible to include systems that are not approved. However, these will not have access to the common services provided by the container.

Figure 20.13 illustrates this integration. This graphic is a simplified version of iLearn that provides three common services:

1. An authentication service that provides a single sign-in to all approved systems. Users do not have to maintain separate credentials for these systems.
2. A storage service for user data. This service can be seamlessly transferred to and from approved systems.
3. A configuration service that is used to include or remove systems from the container.

This example shows a version of iLearn for Physics. As well as an office productivity system (Office 365) and a VLE (Moodle), this system includes simulation and data analysis systems. Other systems—YouTube and a science encyclopedia—are also part of this system. However, these are not "approved," and so no container interface is available. Users must log on to these systems separately and organize their own data transfers.

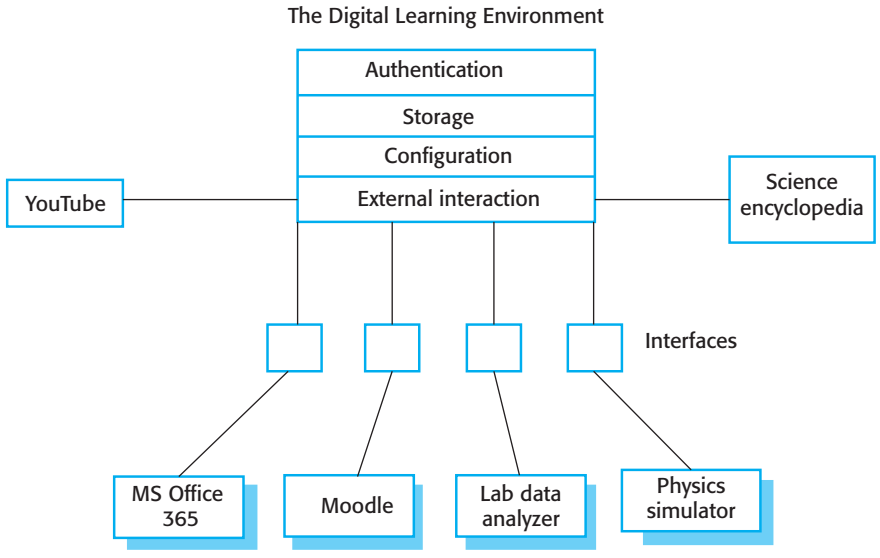


Figure 20.13 The DLE as a container system

There are two problems with this type of SoS architecture:

1. A separate interface must be developed for each approved system so that common services can be used with these systems. This means that only a relatively small number of approved systems can be supported.
2. The owners of the container system have no influence on the functionality and behavior of the included systems. Systems may stop working, or they may be withdrawn at any time.

However, the main benefit of this architecture is that it allows for incremental development. An early version of the container system can be based on "unapproved" systems. Interfaces to these can be developed in later versions so that they are more closely integrated with the container services.

Trading systems

Trading systems are systems of systems where there is no single principal system but processing may take place in any of the constituent systems. The systems involved trade information among themselves. There may be one-to-one or one-to-many interactions between these systems. Each system publishes its own interface, but there may not be any interface standards that are followed by all systems. This system is shown in Figure 20.14. Trading systems may be federated systems or system coalitions.

An example of a trading SoS is a system of systems for algorithmic trading of stocks and shares. Brokers all have their own separate systems that can automatically buy and sell stock from other systems. They set prices and negotiate individually with these systems. Another example of a trading system is a travel aggregator that shows price comparisons and allows travel to be booked directly by a user.

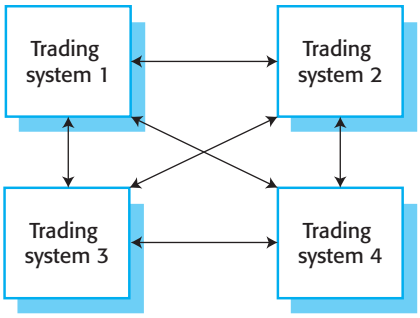


Figure 20.14 A trading system of systems

Trading systems may be developed for any type of marketplace, with the information exchanged being information about the goods being traded and their prices. Although trading systems are systems in their own right and could conceivably be used for individual trading, they are most useful in an automated trading context where the systems negotiate directly with each other.

The major problem with this type of system is that there is no governance mechanism, so any of the systems involved may change at any time. Because these changes may contradict the assumptions made by other systems, trading cannot continue. Sometimes the owners of the systems in the coalition wish to be able to continue trading with other systems and so may make informal arrangements to ensure that changes to one system do not make trading impossible. In other cases, such as a travel aggregator, an airline may deliberately change its system so that it is unavailable and so force bookings to be made directly with it.

KEY POINTS

- Systems of systems are systems where two or more of the constituent systems are independently managed and governed.
- Three types of complexity are important for systems of systems—technical complexity, managerial complexity, and governance complexity.
- System governance can be used as the basis for a classification scheme for SoS. This leads to three classes of SoS, namely, organizational systems, federated systems, and system coalitions.
- Reductionism as an engineering method breaks down because of the inherent complexity of systems of systems. Reductionism assumes clear system boundaries, rational decision making, and well-defined problems. None of these are true for systems of systems.
- The key stages of the SoS development process are conceptual design, system selection, architectural design, interface development, and integration and deployment. Governance and management policies must be designed in parallel with these activities.

- Architectural patterns for systems of systems are a means of describing and discussing typical architectures for SoS. Important patterns are systems as data feeds, systems in a container, and trading systems.

FURTHER READING

"Architecting Principles for Systems of Systems." A now-classic paper on systems of systems that introduces a classification scheme for SoS, discusses its value, and proposes a number of architectural principles for SoS design. (M. Maier, *Systems Engineering*, 1 (4), 1998).

Ultra-large Scale Systems: The Software Challenge of the Future This book, produced for the U.S. Department of Defense in 2006, introduces the notion of ultra-large-scale systems, which are systems of systems with hundreds of nodes. It discusses the issues and challenges in developing such systems. (L. Northrop et al., Software Engineering Institute, 2006). http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

"Large-scale Complex IT Systems." This paper discusses the problems of large-scale complex IT systems that are systems of systems and expands on the ideas here on the breakdown of reductionism. It proposes a number of research challenges in the area of SoS. (I. Sommerville et al., Communications of the ACM, 55 (7), July 2012). http://dx.doi.org/ 10.1145/2209249.2209268

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

20.1. Explain why managerial and operational independence are the key distinguishing characteristics of systems of systems when compared to other large, complex systems.

20.2. Briefly explain any four essential characteristics of systems of systems.

20.3. The classification of SoS presented in Section 20.2 suggests a governance-based classification scheme. Giving reasons for your answer, identify the classifications for the following systems of systems:

(a) A health care system that provides unified access to all patient health records from hospitals, clinics, and primary care.
(b) The World Wide Web
(c) A government system that provides access to a range of welfare services such as pensions, disability benefits, and unemployment benefits.

Are there any problems with the suggested classification for any of these systems?

20.4. Explain what is meant by reductionism and why it is effective as a basis for many kinds of engineering.

20.5. Define systems of systems engineering. List the problems of software SoS engineering that are also common to problems of integrating large-scale application systems.

20.6. How beneficial is a unified user interface in the interface design of SoS? What are the factors on which the cost-effectiveness of a unified user interface is dependent?

20.7. Sillitto suggests that communications between nodes in a SoS are not just technical but should also include informal sociotechnical communications between the people involved in the system. Using the iLearn SoS as an example, suggest where these informal communications may be important to improve the effectiveness of the system.

20.8. Suggest the closest-fit architectural pattern for the systems of systems introduced in Exercise 20.3.

20.9. The trading system pattern assumes that there is no central authority involved. However, in areas such as equity trading, trading systems must follow regulatory rules. Suggest how this pattern might be modified to allow a regulator to check that these rules have been followed. This should not involve all trades going through a central node.

20.10. You work for a software company that has developed a system that provides information about consumers and that is used within a SoS by a number of other retail businesses. They pay you for the services used. Discuss the ethics of changing the system interfaces without notice to coerce users into paying higher charges. Consider this question from the point of view of the company's employees, customers, and shareholders.

REFERENCES

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Hitchins, D. 2009. "System of Systems—The Ultimate Tautology." http://www.hitchins.net/profs-stuff/profs-blog/system-of-systems---the.html

Kawalsky, R., D. Joannou, Y. Tian, and A. Fayoumi. 2013. “Using Architecture Patterns to Architect and Analyze Systems of Systems.” In *Conference on Systems Engineering Research (CSER 13)*, 283–292. doi:10.1016/j.procs.2013.01.030.

Maier, M. W. 1998. “Architecting Principles for Systems-of-Systems.” *Systems Engineering* 1 (4): 267–284. doi:10.1002/(SICI)1520–6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

MOD, UK. 2008. “MOD Architecture Framework.” https://www.gov.uk/mod-architecture-framework

Northrop, Linda, R. P. Gabriel, M. Klein, and D. Schmidt. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh: Software Engineering Institute. http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

Open Group. 2011. “Open Group Standard TOGAF Version 9.1.” http://pubs.opengroup.org/architecture/togaf91-doc/arch/

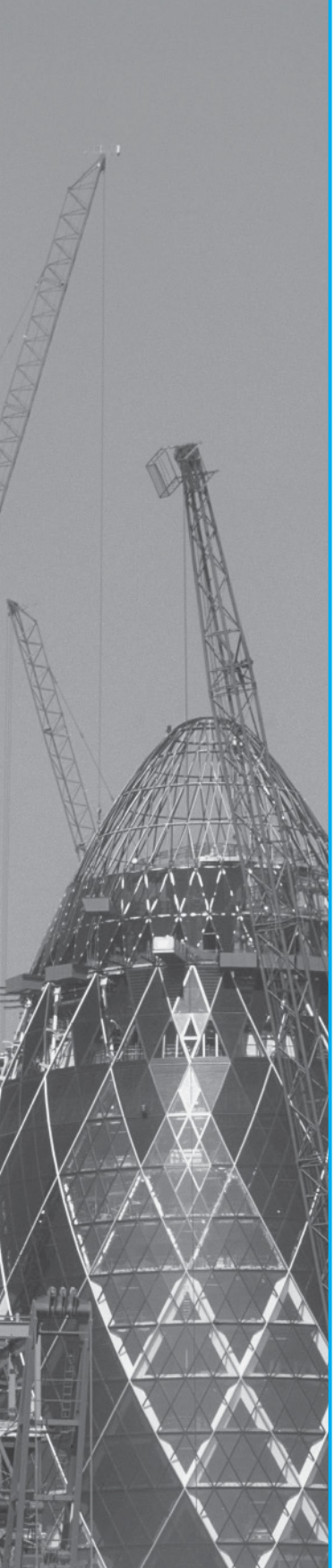
Rittel, H., and M. Webber. 1973. “Dilemmas in a General Theory of Planning.” *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Royal Academy of Engineering. 2004. “Challenges of Complex IT Projects.” London. http://www.bcs.org/upload/pdf/complexity.pdf

Sillitto, H. 2010. “Design Principles for Ultra-Large-Scale Systems.” In *Proceedings of the 20th International Council for Systems Engineering International Symposium*. Chicago.

Sommerville, I., D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige. 2012. “Large-Scale Complex IT Systems.” *Comm. ACM* 55 (7): 71–77. doi:10.1145/2209249.2209268.

Stevens, R. 2010. *Engineering Mega-Systems: The Challenge of Systems Engineering in the Information Age*. Boca Raton, FL: CRC Press.



21

Real-time software engineering

Objectives

The objective of this chapter is to introduce some of the characteristic features of embedded real-time software engineering. When you have read this chapter, you will:

- understand the concept of embedded software, which is used to control systems that react to external events in their environment;
- have been introduced to a design process for real-time systems, where the software systems are organized as a set of cooperating processes;
- understand three architectural patterns that are commonly used in embedded real-time systems design;
- understand the organization of real-time operating systems and the role that they play in an embedded, real-time system.

Contents

21.1 Embedded systems design
21.2 Architectural patterns for real-time software
21.3 Timing analysis
21.4 Real-time operating systems

Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants. These computers interact directly with hardware devices. Their software must react to events generated by the hardware and often issue control signals in response to these events. These signals result in an action, such as the initiation of a phone call, the movement of a character on the screen, the opening of a valve, or the display of the system status. The software in these systems is embedded in system hardware, often in read-only memory. It responds, in real time, to events from the system's environment. By real time, I mean that the software system has a deadline for responding to external events. If this deadline is missed, then the overall hardware–software system will not operate correctly.

Embedded software is very important economically because almost every electrical device now includes software. There are therefore many more embedded software systems than other types of software systems. Ebert and Jones (Ebert and Jones 2009) estimated that there were about 30 embedded microprocessor systems per person in developed countries. This figure was increasing between 10% and 20% per year. This suggests that, by 2020, there will be more than 100 embedded systems per person.

Responsiveness in real time is the critical difference between embedded systems and other software systems, such as information systems, web-based systems, or personal software systems, whose main purpose is data processing. For non–real-time systems, the correctness of a system can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system. In response to an input, a corresponding output should be generated by the system and, often, some data should be stored. For example, if you choose a create command in a patient information system, then the correct system response is to create a new patient record in a database and to confirm that this has been done. Within reasonable limits, it does not matter how long this takes.

However, in a real-time system, the correctness depends both on the response to an input and the time taken to generate that response. If the system takes too long to respond, then the required response may be ineffective. For example, if embedded software controlling a car's braking system is too slow, then an accident may occur because it is impossible to stop the car in time.

Therefore, time is fundamental in the definition of a real-time software system:

> *A real-time software system is a system whose correct operation depends on both the results produced by the system and the time at which these results are produced. A "soft real-time system" is a system whose operation is degraded if results are not produced according to the specified timing requirements. If results are not produced according to the timing specification in a "hard real-time system," this is considered to be a system failure.*

Timely response is an important factor in all embedded systems, but not all embedded systems require a very fast response. For example, the insulin pump software that I have used as an example in several chapters of this book is an embedded system. However, while the system needs to check the glucose level at periodic intervals, it does not need to

respond very quickly to external events. The wilderness weather station software is also an embedded system, but, again, it does not require a fast response to external events.

As well as the need for real-time response, there are other important differences between embedded systems and other types of software system:

1. Embedded systems generally run continuously and do not terminate. They start when the hardware is switched on, and execute until the hardware is switched off. Techniques for reliable software engineering, as discussed in Chapter 11, may therefore have to be used to ensure continuous operation. The real-time system may include update mechanisms that support dynamic reconfiguration so that the system can be updated while it is in service.
2. Interactions with the system's environment are unpredictable. In interactive systems, the pace of the interaction is controlled by the system. By limiting user options, the events and commands to be processed are known in advance. By contrast, real-time embedded systems must be able to respond to expected and unexpected events at any time. This leads to a design for real-time systems based on concurrency, with several processes executing in parallel.
3. Physical limitations may affect the design of a system. Examples of limitations include restrictions on the power available to the system and the physical space taken up by the hardware. These limitations may generate requirements for the embedded software, such as the need to conserve power and so prolong battery life. Size and weight limitations may mean that the software has to take over some hardware functions because of the need to limit the number of chips used in the system.
4. Direct hardware interaction may be necessary. In interactive systems and information systems, a layer of software (the device drivers) hides the hardware from the operating system. This is possible because you can only connect a few types of device to these systems, such as keyboards, mice, and displays. By contrast, embedded systems may have to interact with a wide range of hardware devices that do not have separate device drivers.
5. Issues of safety and reliability may dominate the system design. Many embedded systems control devices whose failure may have high human or economic costs. Therefore, dependability is critical, and the system design has to ensure safety-critical behavior at all times. This often leads to a conservative approach to design where tried and tested techniques are used instead of newer techniques that may introduce new failure modes.

Real-time embedded systems can be thought of as reactive systems; that is, they must react to events in their environment (Berry 1989; Lee 2002). Response times are often governed by the laws of physics rather than chosen for human convenience. This is in contrast to other types of software where the system controls the speed of the interaction. For example, the word processor that I am using to write this book can check spelling and grammar, and there are no practical limits on the time taken to do so.

21.1 Embedded system design

During the design process for embedded software, software designers have to consider in detail the design and performance of the system hardware. Part of the system design process may involve deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems that are embedded in consumer products, such as the systems in cell phones, the costs and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used. For some systems, special-purpose hardware may have to be designed and built.

A top-down software design process, in which the design starts with an abstract model that is decomposed and developed in a series of stages, is impractical for most real-time systems. Low-level decisions on hardware, support software, and system timing must be considered early in the process. These limit the flexibility of system designers. Additional software functionality, such as battery and power management, may have to be included in the system.

Given that embedded systems are reactive systems that react to events in their environment, the most general approach to embedded, real-time software design is based on a stimulus-response model. A stimulus is an event occurring in the software system's environment that causes the system to react in some way; a response is a signal or message that the software sends to its environment.

You can define the behavior of a real-time system by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. For example, Figure 21.1 shows possible stimuli and system responses for a burglar alarm system (discussed in Section 21.2.1).

Stimuli fall into two classes:

1. *Periodic stimuli* These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
2. *Aperiodic stimuli* These occur irregularly and unpredictably and are usually signaled, using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

Stimuli come from sensors in the system's environment, and responses are sent to actuators, as shown in Figure 21.2. These actuators control equipment, such as a pump, which then makes changes to the system's environment. The actuators themselves may also generate stimuli. The stimuli from actuators often indicate that some problem with the actuator has occurred, which must be handled by the system.

A general design guideline for real-time systems is to have separate control processes for each type of sensor and actuator (Figure 21.3). For each type of sensor, there may be a sensor management process that handles data collection from these sensors. Data-processing processes compute the required responses for the stimuli received by the system. Actuator control processes are associated with each actuator

Stimulus	Response
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police, run power supply test.

Figure 21.1 Stimuli and responses for a burglar alarm system

and manage the operation of that actuator. This model allows data to be collected quickly from the sensor (before it is overwritten by the next input) and enables processing and the associated actuator response to be carried out later.

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize the system architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time software systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform on which the real-time system executes may include a real-time operating system (discussed in Section 21.4). The functions provided by this operating system are accessed through the runtime support system for the real-time programming language that is used.

There is no standard embedded system design process. Rather, different processes are used that depend on the type of system, available hardware, and the organization that is developing the system. The following activities may be included in a real-time software design process:

1. *Platform selection* In this activity, you choose an execution platform for the system, that is, the hardware and the real-time operating system to be used. Factors that influence these choices include the timing constraints on the system, limitations on power available, the experience of the development team, and the price target for the delivered system.
2. *Stimuli/response identification* This involves identifying the stimuli that the system must process and the associated response or responses for each stimulus.

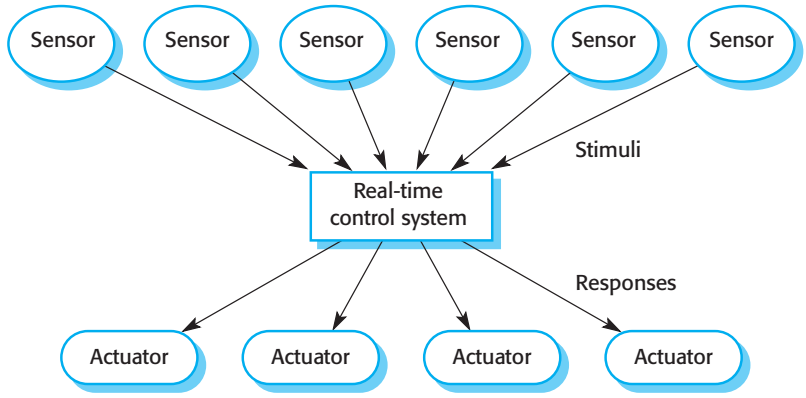


Figure 21.2 A general model of an embedded real-time system

3. *Timing analysis* For each stimulus and associated response, you identify the timing constraints that apply to both stimulus and response processing. These constraints are used to establish the deadlines for the processes in the system.
4. *Process design* Process design involves aggregating the stimulus and response processing into a number of concurrent processes. A good starting point for designing the process architecture is the architectural patterns that I describe in Section 20.2. You then optimize the process architecture to reflect the specific requirements that you have to implement.
5. *Algorithm design* For each stimulus and response, you design algorithms to carry out the required computations. Algorithm designs may have to be developed relatively early in the design process to indicate the amount of processing required and the time needed to complete that processing. This is especially important for computationally intensive tasks, such as signal processing.
6. *Data design* You specify the information that is exchanged by processes and the events that coordinate information exchange, and design data structures to manage this information exchange. Several concurrent processes may share these data structures.
7. *Process scheduling* You design a scheduling system that will ensure that processes are started in time to meet their deadlines.

The specific activities and the activity sequence in a real-time system design process depend on the type of system being developed, its novelty, and its environment.

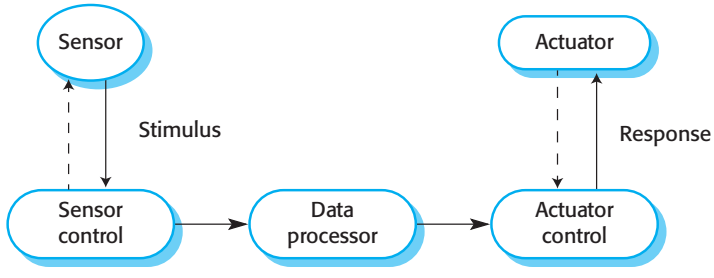


Figure 21.3 Sensor and actuator processes

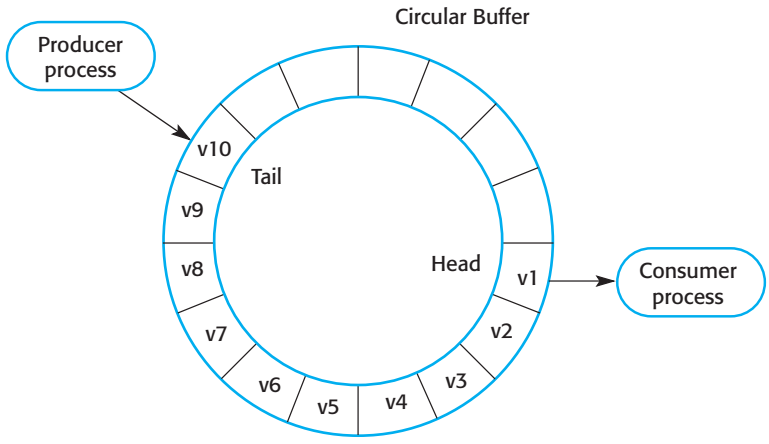


Figure 21.4 Producer/consumer processes sharing a circular buffer

In some cases, for new systems, you may be able to follow a fairly abstract approach where you start with the stimuli and associated processing, and decide on the hardware and execution platforms late in the process. In other cases, the choice of hardware and operating system is made before the software design starts. You then have to design the software to take account of the constraints imposed by the system hardware.

Processes in a real-time system have to be coordinated and share information. Process coordination mechanisms ensure mutual exclusion to shared resources. When one process is modifying a shared resource, other processes should not be able to change that resource. Mechanisms for ensuring mutual exclusion include semaphores, monitors, and critical regions. These process synchronization mechanisms are described in most operating system books (Silberschaltz, Galvin, and Gagne 2013; Stallings 2014).

When designing the information exchange between processes, you have to take into account that these processes may be running at different speeds. One process is producing information, and the other process is consuming that information. If the producer is running faster than the consumer, new information could overwrite a previously read information item before the consumer process has read the original information. If the consumer process is running faster than the producer process, the same item could be read twice.

To avoid this problem, you should implement information exchange using a shared buffer and use mutual exclusion mechanisms to control access to that buffer. This means that information can't be overwritten before it has been read and that information cannot be read twice. Figure 21.4 illustrates the organization of a shared buffer. This is usually implemented as a circular queue, using a list data structure. Mismatches in speed between the producer and consumer processes can be accommodated without having to delay process execution.

The producer process always enters data in the buffer location at the end of the queue (represented as v10 in Figure 21.4). The consumer process always retrieves information from the head of the queue (represented as v1 in Figure 21.4). After the consumer process has retrieved the information, the tail of the queue is adjusted to point at the next item (v2). After the producer process has added information, the tail of the queue is adjusted to point at the next free slot in the queue.

Obviously, it is important to ensure that the producer and consumer process do not attempt to access the same item at the same time (i.e., when Head = Tail). If they do, the value of the item is unpredictable. The system also has to ensure that the producer process does not add items to a full buffer and that the consumer process does not try to take items from an empty buffer.

To do this, you implement the circular buffer as a process with Get and Put operations to access the buffer. The Put operation is called by the producer process and the Get operation by the consumer process. Synchronization primitives, such as semaphores or critical regions, are used to ensure that the operation of Get and Put are synchronized, so that they don't access the same location simultaneously. If the buffer is full, the Put process has to wait until a slot is free; if the buffer is empty, the Get process has to wait until an entry has been made.

Once you have chosen the execution platform for the system, designed a process architecture, and decided on a scheduling policy, you have to check that the system will meet its timing requirements. You can perform this check through static analysis of the system using knowledge of the timing behavior of components, or through simulation. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the execution platform, or all of these may then have to be redesigned to improve the performance of the system.

Timing constraints or other requirements may sometimes mean that it is best to implement some system functions, such as signal processing, in hardware. Modern hardware components, such as FPGAs (field-programmable gate arrays), are flexible and can be adapted to different functions. Hardware components deliver much better performance than the equivalent software. System processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimization.

21.1.1 Real-time system modeling

The events that a real-time system must react to often cause the system to move from one state to another. For this reason, state models, which I introduced in Chapter 5, are used to describe real-time systems. A state model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may cause a transition to a different state. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received.

State models are an integral part of real-time system design methods. The UML supports the development of state models based on Statecharts (Harel 1987, 1988). Statecharts are formal state machine models that support hierarchical states, so that groups of states can be considered as a single entity. Douglass discusses the use of the UML in real-time systems development (Douglass 1999).

I have already illustrated this approach to system modeling in Chapter 5 where I used an example of a model of a simple microwave oven. Figure 21.5 is another example of a state model that shows the operation of a fuel delivery software system embedded in a petrol (gas) pump. The rounded rectangles represent system states, and the arrows represent stimuli that force a transition from one state to another.

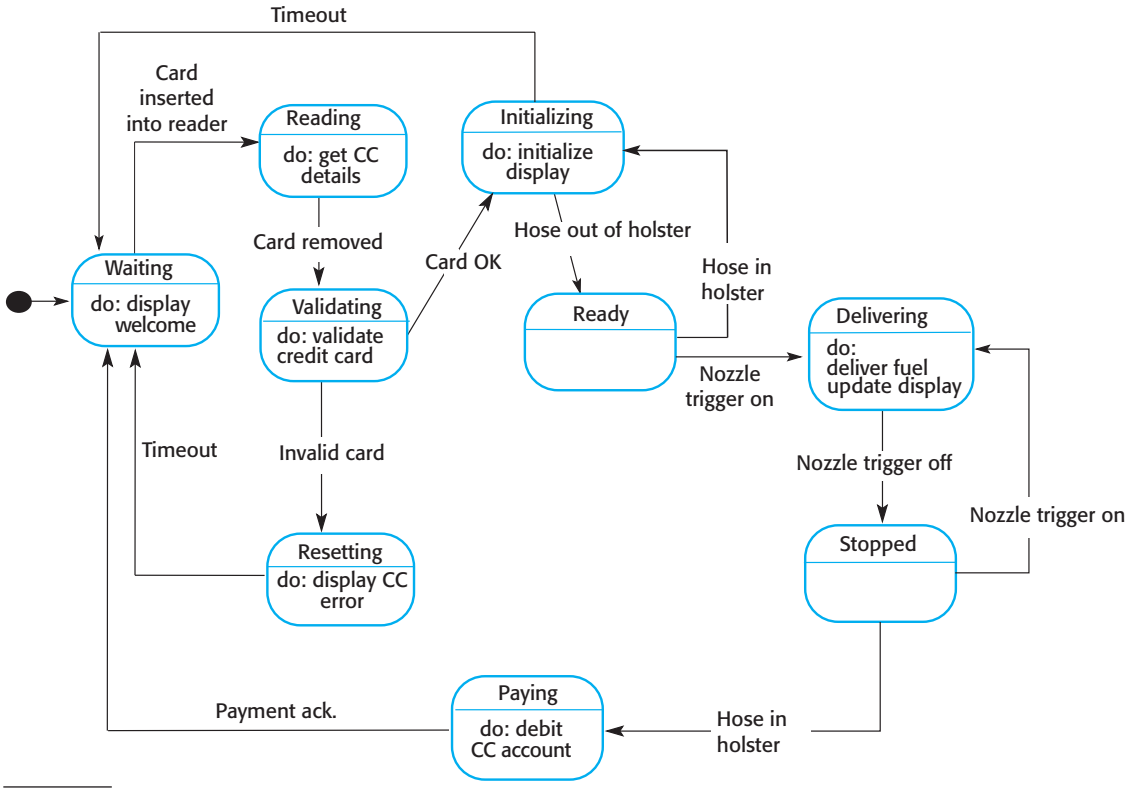


Figure 21.5 State machine model of a petrol (gas) pump

The names chosen in the state machine diagram are descriptive. The associated information indicates actions taken by the system actuators or information that is displayed. Notice that this system never terminates but idles in a waiting state when the pump is not operating.

The fuel delivery system is designed to allow unattended operation, with the following sequence of actions:

1. The buyer inserts a credit card into a card reader built into the pump. This causes a transition to a Reading state where the card details are read and the buyer is then asked to remove the card.
2. Removal of the card triggers a transition to a Validating state where the card is validated.
3. If the card is valid, the system initializes the pump and, when the fuel hose is removed from its holster, transitions to the Delivering state, where is ready to deliver fuel. Activating the trigger on the nozzle causes fuel to be pumped; this stops when the trigger is released (for simplicity, I have ignored the pressure switch that is designed to stop fuel spillage).



Real-time Java

The Java programming language has been modified to make it suitable for real-time systems development. These modifications include asynchronous communications, the addition of time, including absolute and relative time, a new thread model where threads cannot be interrupted by garbage collection, and a new memory management model that avoids the unpredictable delays that can result from garbage collection.

http://software-engineering-book.com/web/real-time-java/

4. After the fuel delivery is complete and the buyer has replaced the hose in its holster, the system moves to a Paying state where the user's account is debited.
5. After payment, the pump software returns to the Waiting state.

State models are used in model-driven engineering, which I discussed in Chapter 5, to define the operation of a system. They can be transformed automatically or semiautomatically to an executable program.

21.1.2 Real-time programming

Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages. Hard real-time systems, running on limited hardware, are still sometimes programmed in assembly language so that tight deadlines can be met. Systems programming languages, such as C, which allow efficient code to be generated, are widely used.

The advantage of using a systems programming language like C is that it allows the development of efficient programs. However, these languages do not include constructs to support concurrency or the management of shared resources. Concurrency and resource management are implemented through calls to primitives provided by the real-time operating system for mutual exclusion. Because the compiler cannot check these calls, programming errors are more likely. Programs are also often more difficult to understand because the language does not include real-time features. As well as understanding the program, the reader also has to know how real-time support is provided using system calls.

Because real-time systems must meet their timing constraints, you may not be able to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data representations and accessing attribute values through operations defined with the object. There is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The consequent loss of performance may make it impossible to meet real-time deadlines.

A version of Java has been developed for embedded systems development (Burns and Wellings 2009; Bruno and Bollella 2009). This language includes a modified thread mechanism, which allows threads to be specified that will not be interrupted

by the language garbage collection mechanism. Asynchronous event handling and timing specification has also been included. However, at the time of writing, this specification has mostly been used on platforms that have significant processor and memory capacity (e.g., a cell phone) rather than simpler embedded systems, with more limited resources. These systems are still usually implemented in C.

21.2 Architectural patterns for real-time software

Architectural patterns are abstract, stylized descriptions of good design practice. They capture knowledge about the organization of system architectures, when these architectures should be used, and their advantages and disadvantages. You use an architectural pattern to understand an architecture and as starting point for creating your own, specific architectural design.

The difference between real-time and interactive software means that there are distinct architectural patterns for real-time embedded systems. Real-time systems' patterns are process-oriented rather than object- or component-oriented. In this section, I discuss three real-time architectural patterns that are commonly used:

1. *Observe and React* This pattern is used when a set of sensors are routinely monitored and displayed. When the sensors show that some event has occurred (e.g., an incoming call on a cell phone), the system reacts by initiating a process to handle that event.
2. *Environmental Control* This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment. In response to environmental changes detected by the sensor, control signals are sent to the system actuators.
3. *Process Pipeline* This pattern is used when data has to be transformed from one representation to another before it can be processed. The transformation is implemented as a sequence of processing steps, which may be carried out concurrently. This allows for very fast data processing, because a separate core or processor can execute each transformation.

These patterns can of course be combined, and you will often see more than one of them in a single system. For example, when the Environmental Control pattern is used, it is very common for the actuators to be monitored using the Observe and React pattern. In the event of an actuator failure, the system may react by displaying a warning message, shutting down the actuator, switching in a backup system, and so forth.

The patterns that I cover are architectural patterns that describe the overall structure of an embedded system. Douglass (Douglass 2002) describes lower-level, real-time design patterns that support more detailed design decision making. These patterns include design patterns for execution control, communications, resource allocation, and safety and reliability.

Name	Observe and React
Description	The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, if necessary, take actions in response to the exceptional value.
Stimuli	Values from sensors attached to the system.
Responses	Outputs to display, alarm triggers, signals to reacting systems.
Processes	Observer, Analysis, Display, Alarm, Reactor.
Used in	Monitoring systems, alarm systems.

Figure 21.6 The Observe and React pattern

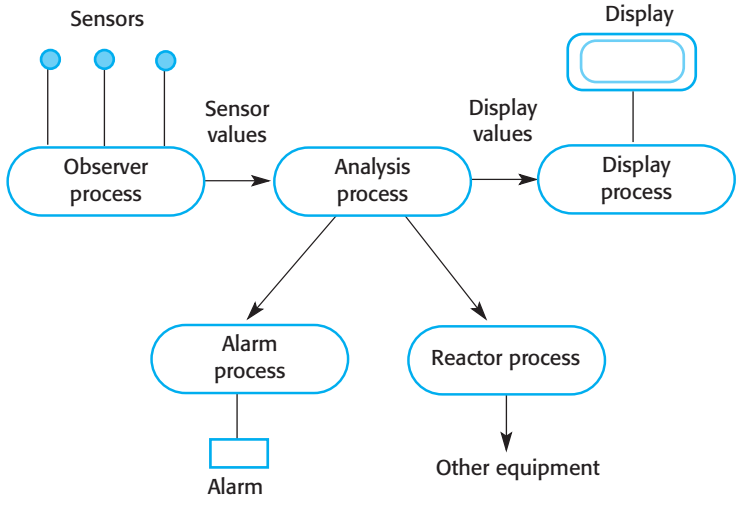


Figure 21.7 The Observe and React process structure

These architectural patterns should be the starting point for an embedded systems design; however, they are not design templates. If you use them as such, you will probably end up with an inefficient process architecture. You have to optimize the process structure to ensure that you do not have too many processes. You also should ensure that there is a clear correspondence between the processes and the sensors and actuators in the system.

21.2.1 Observe and react

Monitoring systems are an important class of embedded real-time systems. A monitoring system examines its environment through a set of sensors and usually displays the state of the environment in some way. This could be on a built-in screen, on special-purpose instrument displays, or on a remote display. If the system detects some exceptional event or sensor state, the monitoring system takes some action.

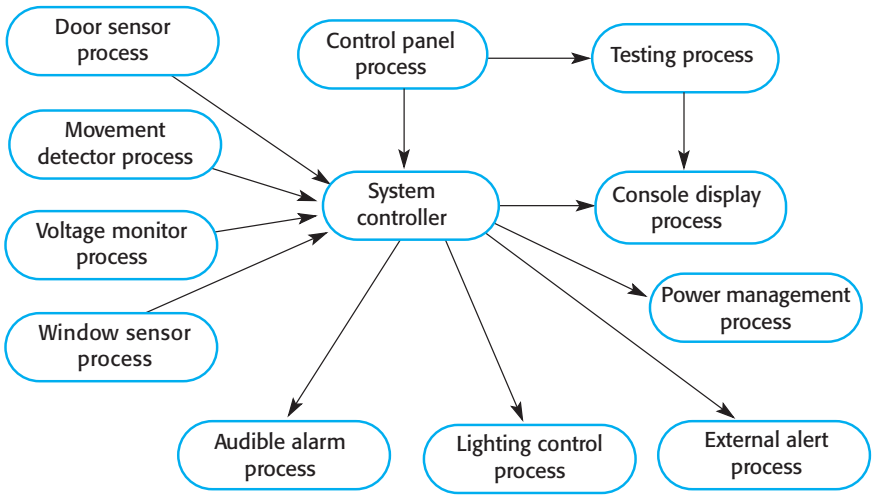


Figure 21.8 The process structure of a burglar alarm system

This often involves raising an alarm to draw an operator's attention to the event. Sometimes the system may initiate some other preventative action, such as shutting down the system to preserve it from damage.

The Observe and React pattern (Figures 21.6 and 21.7) is commonly used in monitoring systems. The values of sensors are observed, and the system initiates actions that depend on these sensor values. Monitoring systems may be composed of several instantiations of the Observe and React pattern, one for each type of sensor in the system. Depending on the system requirements, you may then optimize the design by combining processes (e.g., you may use a single display process to display the information from all of the different types of sensor).

As an example of the use of this pattern, consider the design of a burglar alarm system to be installed in an office building:

> *A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensors. These sensors include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that can detect when a window has been opened.*
>
> *When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The sensor system is normally powered by mains power but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the mains voltage. If a voltage drop is detected, the system assumes that intruders have interrupted the power supply, so an alarm is raised.*

A process architecture for the alarm system is shown in Figure 21.8. The arrows represent signals sent from one process to another. This system is a "soft" real-time system that does not have stringent timing requirements. The sensors only need to detect

Name	Environmental Control
Description	The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators, which then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.
Stimuli	Values from sensors attached to the system and the state of the system actuators.
Responses	Control signals to actuators display information.
Processes	Monitor, Control, Display, Actuator driver, Actuator monitor.
Used in	Control systems.

Figure 21.9 The Environmental Control pattern

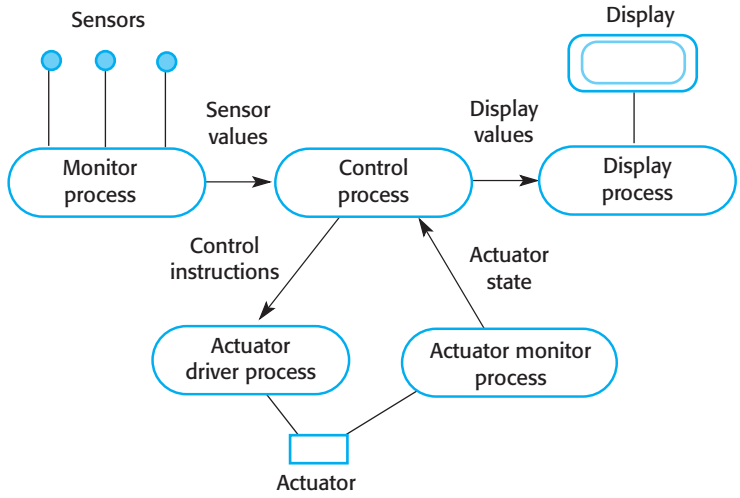


Figure 21.10 The Environmental Control process structure

the presence of people rather than high-speed events, so they only need to be polled 2 or 3 times per second. I cover the timing requirements for this system in Section 21.3.

I have already introduced the stimuli and responses in this alarm system in Figure 21.1. These responses are used as a starting point for the system design. The Observe and React pattern is used in this design. There are observer processes associated with each type of sensor and reactor processes for each type of reaction. A single analysis process checks the data from all of the sensors. The display processes in the pattern are combined into a single display process.

21.2.2 Environmental Control

The most widespread use of real-time embedded software is in control systems. In these systems, the software controls the operation of equipment, based on stimuli

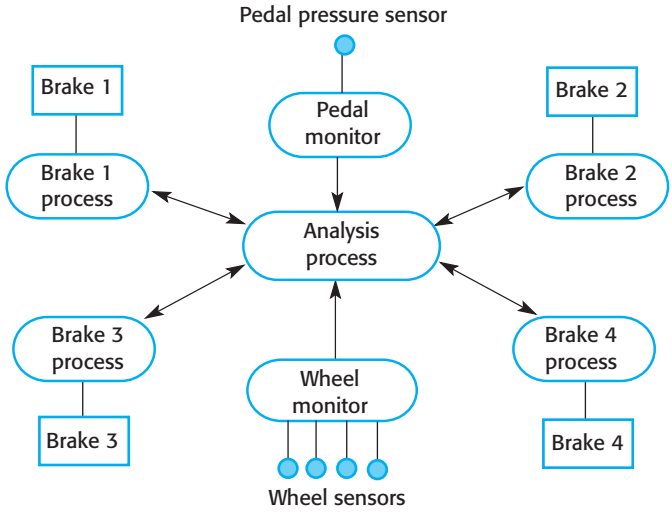


Figure 21.11 Control system architecture for an anti-skid braking system

from the equipment's environment. For example, an anti-skid braking system in a car monitors the car's wheels and brake system (the system's environment). It looks for signs that the wheels are skidding when brake pressure is applied. If this is the case, the system adjusts the brake pressure to stop the wheels locking and reduce the likelihood of a skid.

Control systems may make use of the Environmental Control pattern, which is a general control pattern that includes sensor and actuator processes. This pattern is described in Figure 21.9, with the process architecture shown in Figure 21.10. A variant of this pattern leaves out the display process. This variant is used in situations where user intervention is not required or where the rate of control is so high that a display would not be meaningful.

This pattern can be the basis for a control system design with an instantiation of the Environmental Control pattern for each actuator (or actuator type) being controlled. You then optimize the design to reduce the number of processes. For example, you may combine actuator monitoring and actuator control processes, or you may have a single monitoring and control process for several actuators. The optimizations that you choose depend on the timing requirements. You may need to monitor sensors more frequently than you send control signals, in which case it may be impractical to combine control and monitoring processes. There may also be direct feedback between the actuator control and the actuator monitoring process. This allows fine-grain control decisions to be made by the actuator control process.

You can see how this pattern is used in Figure 21.11, which shows an example of a controller for a car braking system. The starting point for the design is associating an instance of the pattern with each actuator type in the system. In this case, there are four actuators, with each controlling the brake on one wheel. The individual sensor processes are combined into a single wheel-monitoring process that monitors the sensors on all

Name	Process Pipeline
Description	A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage, or the pipeline may terminate in an actuator.
Stimuli	Input values from the environment or some other process
Responses	Output values to the environment or a shared buffer
Processes	Producer, Buffer, Consumer
Used in	Data acquisition systems, multi-media systems

Figure 21.12 The Process Pipeline pattern

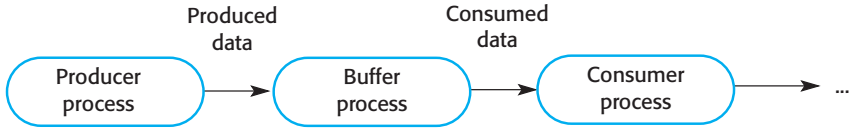


Figure 21.13 Process Pipeline process structure

wheels. This monitors the state of each wheel to check if the wheel is turning or locked. A separate process monitors the pressure on the brake pedal exerted by the car driver.

The system includes an anti-skid feature, which is triggered if the sensors indicate that a wheel is locked when the brake has been applied. This means that there is insufficient friction between the road and the tire; in other words, the car is skidding. If the wheel is locked, the driver cannot steer that wheel. To counteract this effect, the system sends a rapid sequence of on/off signals to the brake on that wheel, which allows the wheel to turn and control to be regained.

The Wheel monitor process monitors whether or not each wheel is turning. If a wheel is skidding (not turning), it informs the Analysis process. This then signals the processes associated with the wheels that are skidding to initiate anti-skid braking.

21.2.3 Process pipeline

Many real-time systems are concerned with collecting analog data from the system's environment. They then digitize that data for analysis and processing by the system. The system may also convert digital data to analog data, which it then sends to its environment. For example, a software radio accepts incoming packets of digital data representing the radio transmission and transforms the data into a sound signal that people can listen to.

The data processing involved in many of these systems has to be carried out very quickly. Otherwise, incoming data may be lost and outgoing signals may be broken up because essential information is missing. The Process Pipeline pattern makes this rapid processing possible by breaking down the required data processing into a sequence of separate transformations. Each of these transformations is implemented

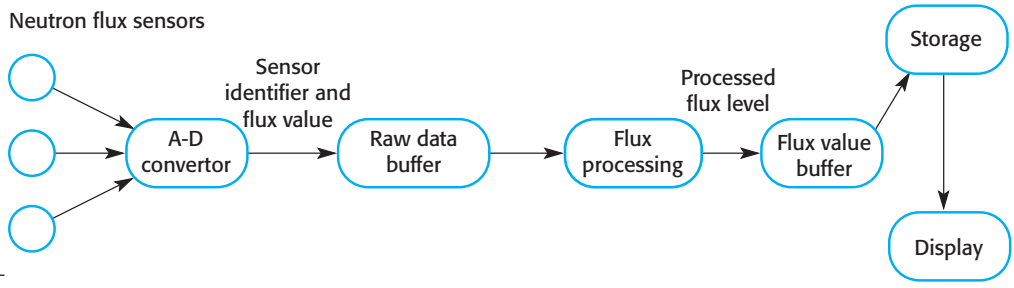


Figure 21.14 Neutron flux data acquisition

by an independent process. This architecture is efficient for systems that use multiple processors or multicore processors. Each process in the pipeline can be associated with a separate processor or core, so that the processing steps can be carried out in parallel.

Figure 21.12 is a brief description of the data pipeline pattern, and Figure 21.13 shows the process architecture for this pattern. Notice that the processes involved produce and consume information. The processes exchange information using synchronized buffers, as I explained in Section 21.1. Producer and consumer processes can thereby operate at different speeds without data losses.

An example of a system that may use a process pipeline is a high-speed data acquisition system. Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in situations where the sensors are collecting large volumes of data from the system's environment and it isn't possible or necessary to process that data in real time. Rather, it is collected and stored for later analysis. Data acquisition systems are often used in scientific experiments and process control systems where physical processes, such as chemical reactions, are very rapid. In these systems, the sensors may be generating data very quickly, and the data acquisition system has to ensure that a sensor reading is collected before the sensor value changes.

Figure 21.14 is a simplified model of a data acquisition system that might be part of the control software in a nuclear reactor. This system collects data from sensors monitoring the neutron flux (the density of neutrons) in the reactor. The sensor data is placed in a buffer from which it is extracted and processed. The average flux level is displayed on an operator's display and stored for future processing.

21.3 Timing analysis

As I discussed in the introduction to this chapter, the correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced. Therefore, timing analysis is an important activity in the embedded, real-time software development process. In such an analysis, you calculate how often each process in the system must be executed to ensure that all inputs

are processed and all system responses are produced in a timely way. The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.

Timing analysis for real-time systems is particularly difficult when the system has to deal with a mixture of periodic and aperiodic stimuli and responses. Because aperiodic stimuli are unpredictable, you have to make assumptions about the probability of these stimuli occurring and therefore requiring service at any particular time. These assumptions may be incorrect, and system performance after delivery may not be adequate. Cooling's book (Cooling 2003) discusses techniques for real-time system performance analysis that takes aperiodic events into account.

As computers have become faster, it has become possible in many systems to design using only periodic stimuli. When processors were slow, aperiodic stimuli had to be used to ensure that critical events were processed before their deadline, as delays in processing usually involved some loss to the system. For example, the failure of a power supply in an embedded system may mean that the system has to shut down attached equipment in a controlled way, within a very short time (say 50 milliseconds). This could be implemented as a "power fail" interrupt. However, it can also be implemented using a periodic process that runs frequently and checks the power. As long as the time between process invocations is short, there is still time to perform a controlled shutdown of the system before the lack of power causes damage. For this reason, I only discuss timing issues for periodic processes.

When you are analyzing the timing requirements of embedded real-time systems and designing systems to meet these requirements, you have to consider three key factors:

1. *Deadlines* The times by which stimuli must be processed and some response produced by the system. If the system does not meet a deadline, then, if it is a hard real-time system, this is a system failure; in a soft real-time system, it results in degraded system service.
2. *Frequency* The number of times per second that a process must execute so that you are confident that it can always meet its deadlines.
3. *Execution time* The time required to process a stimulus and produce a response. Execution time is not always the same because of the conditional execution of code, delays waiting for other processes, and so on. Therefore, you may have to consider both the average execution time of a process and the worst-case execution time for that process. The worst-case execution time is the maximum time that the process takes to execute. In a hard real-time system, you may have to make assumptions based on the worst-case execution time to ensure that deadlines are not missed. In soft real-time systems, you can base your calculations on the average execution time.

To continue the example of a power supply failure, let's calculate the worst-case execution time for a process that switches equipment power from mains

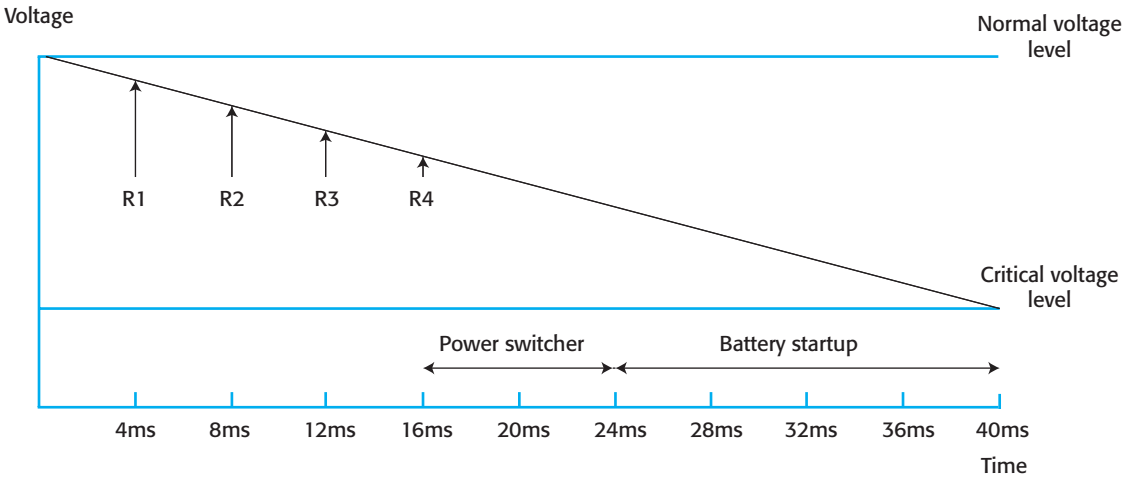


Figure 21.15
Power failure timing analysis

power to a battery backup. Figure 21.15 presents a timeline showing the events in the system:

1. Assume that, after a mains power failure event, it takes 50 milliseconds (ms) for the supplied voltage to drop to a level where the equipment may be damaged. The battery backup must therefore be activated and in operation within 50 ms. Usually, you allow for a margin of error, so you should set a shorter deadline of 40 ms because of physical variations in the equipment. This means that all equipment must be running on the battery backup power supply within 40 ms.
2. However, the battery backup system cannot be instantaneously activated. It takes 16 ms from starting the backup power supply to the supply being fully operational. This means that the time available to detect the power failure and start the battery backup system is 24 ms.
3. There is a process that is scheduled to run 250 times per second, that is, every 4 ms. This process assumes that there is a power supply problem if a significant drop in voltage occurs between readings and is sustained for three readings. This time is allowed so that temporary fluctuations do not cause a switch to the battery backup system.
4. In the above timeline, the power fails immediately after a reading has been taken. Therefore, reading R1 is the start reading for the power fail check. The voltage continues to drop for readings R2–R4, so a power failure is assumed. This is the worst possible case, where a power failure event occurs immediately after a sensor check, so 16 ms have elapsed since that event.
5. At this stage, the process that switches to the battery backup is started. Because the battery backup takes 16 ms to become operational, the worst-case execution time for this process is 8 ms, so that the 40 ms deadline can be reached.

Stimulus/Response	Timing requirements
Audible alarm	The audible alarm should be switched on within half a second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Door alarm	Each door alarm should be polled twice per second.
Lights switch	The lights should be switched on within half a second of an alarm being raised by a sensor.
Movement detector	Each movement detector should be polled twice per second.
Power failure	The switch to backup power must be completed within a deadline of 50 ms.
Voice synthesizer	A synthesized message should be available within 2 seconds of an alarm being raised by a sensor.
Window alarm	Each window alarm should be polled twice per second.

Figure 21.16 Timing requirements for the burglar alarm system

The starting point for timing analysis in a real-time system is the timing requirements, which should set out the deadlines for each required response in the system. Figure 21.16 shows possible timing requirements for the office building burglar alarm system discussed in Section 21.2.1. To simplify this example, let us ignore stimuli generated by system testing procedures and external signals to reset the system in the event of a false alarm. This means there are only two types of stimulus processed by the system:

1. Power failure is detected by observing a voltage drop of more than 20%. The required response is to switch the circuit to backup power by signaling an electronic power-switching device that switches the mains power to battery backup.
2. Intruder alarm is a stimulus generated by one of the system sensors. The response to this stimulus is to compute the room number of the active sensor, set up a call to the police, initiate the voice synthesizer to manage the call, and switch on the audible intruder alarm and building lights in the area.

As shown in Figure 21.16, you should list the timing constraints for each class of sensor separately, even when (as in this case) they are the same. By considering them separately, you leave scope for future change and make it easier to compute the number of times the controlling process has to be executed each second.

Allocating the system functions to concurrent processes is the next design stage. Four types of sensors must be polled periodically, each with an associated process: the voltage sensor, door sensors, window sensors, and movement detectors. Normally, the processes associated with the sensor will execute very quickly as all

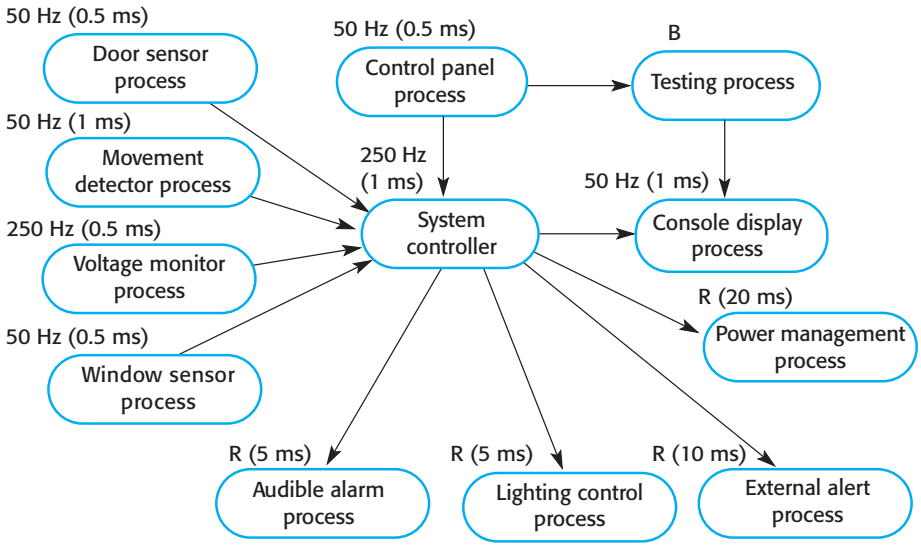


Figure 21.17
Alarm process timing

they are doing is checking whether or not a sensor has changed its status (e.g., from off to on). It is reasonable to assume that the execution time to check and assess the state of one sensor is less than 1 millisecond.

To ensure that you meet the deadlines defined by the timing requirements, you then have to decide how frequently the related processes have to run and how many sensors should be examined during each execution of the process. There are obvious trade-offs here between frequency and execution time:

1. The deadline for detecting a change of state is 0.25 second, which means that each sensor has to be checked 4 times per second. If you examine one sensor during each process execution, then if there are N sensors of a particular type, you must schedule the process 4N times per second to ensure that all sensors are checked within the deadline.
2. If you examine four sensors, say, during each process execution, then the execution time is increased to about 4 ms, but you need only run the process N times/second to meet the timing requirement.

In this case, because the system requirements define actions when two or more sensors are positive, the best strategy is to examine sensors in groups, with groups based on the physical proximity of the sensors. If an intruder has entered the building, then it will probably be adjacent sensors that are positive.

When you have completed the timing analysis, you may then annotate the process model with information about frequency of execution and their expected execution time (see Figure 21.17). Here, periodic processes are annotated with their frequency, processes that are started in response to a stimulus are annotated with R, and the testing process is a background process, annotated with B. This background process

only runs when processor time is available. In general, it is simpler to design a system so that there are a small number of process frequencies. The execution times represent the required worst-case execution times of the processes.

The final step in the design process is to design a scheduling system that will ensure that a process will always be scheduled to meet its deadlines. You can only do this if you know the scheduling approaches that are supported by the real-time operating system (OS) used (Burns and Wellings 2009). The scheduler in the real-time OS allocates a process to a processor for a given amount of time. The time can be fixed, or it may vary depending on the priority of the process.

In allocating process priorities, you have to consider the deadlines of each process so that processes with short deadlines receive processor time to meet these deadlines. For example, the voltage monitor process in the burglar alarm needs to be scheduled so that voltage drops can be detected and a switch made to backup power before the system fails. This should therefore have a higher priority than the processes that check sensor values, as these have fairly relaxed deadlines compared to their expected execution time.

21.4 Real-time operating systems

The execution platform for most application systems is an operating system that manages shared resources and provides features such as a file system and runtime process management. However, the extensive functionality in a conventional operating system takes up a great deal of space and slows down the operation of programs. Furthermore, the process management features in the system may not be designed to allow fine-grain control over the scheduling of processes.

For these reasons, standard operating systems, such as Linux and Windows, are not normally used as the execution platform for real-time systems. Very simple embedded systems may be implemented as "bare metal" systems. The systems provide their own execution support and so include system startup and shutdown, process and resource management, and process scheduling. More commonly, however, embedded applications are built on top of a real-time operating system (RTOS), which is an efficient operating system that offers the features needed by real-time systems. Examples of RTOS are Windows Embedded Compact, VxWorks, and RTLinux.

A real-time operating system manages processes and resource allocation for a real-time system. It starts and stops processes so that stimuli can be handled, and it allocates memory and processor resources. The components of an RTOS (Figure 21.18) depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

1. A real-time clock, which provides the information required to schedule processes periodically.
2. If interrupts are supported, an interrupt handler, which manages aperiodic requests for service.

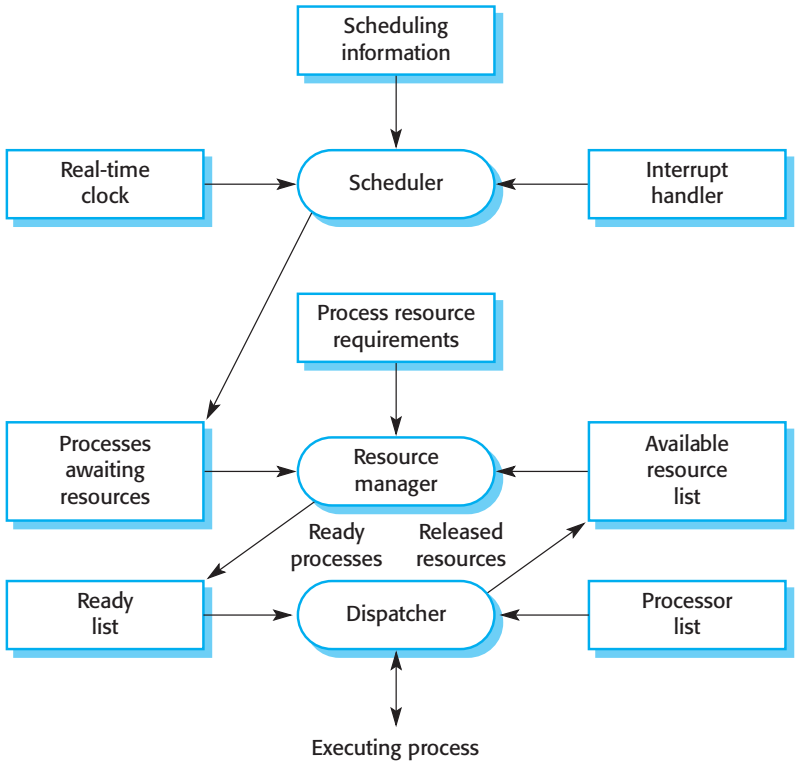


Figure 21.18 Components of a real-time operating system

3. A scheduler, which is responsible for examining the processes that can be executed and for choosing one of these processes for execution.
4. A resource manager, which allocates appropriate memory and processor resources to processes that have been scheduled for execution.
5. A dispatcher, which is responsible for starting the execution of processes.

Real-time operating systems for large systems, such as process control or telecommunication systems, may have additional facilities, namely, disk storage management, fault management facilities that detect and report system faults, and a configuration manager that supports the dynamic reconfiguration of real-time applications.

21.4.1 Process management

Real-time systems have to handle external events quickly and, in some cases, meet deadlines for processing these events. The event-handling processes must therefore be scheduled for execution in time to detect the event. They must also be allocated sufficient processor resources to meet their deadline. The process manager in an RTOS is responsible for choosing processes for execution, allocating processor and memory resources, and starting and stopping process execution on a processor.

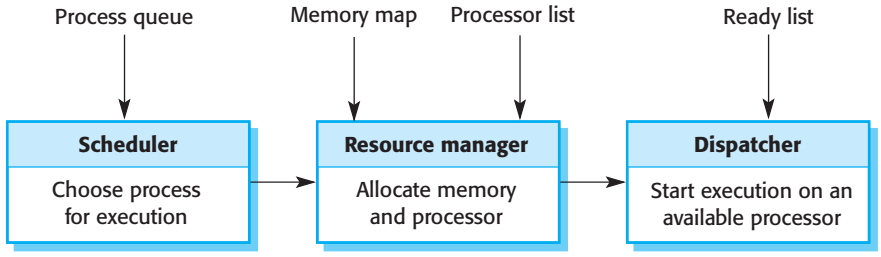


Figure 21.19 RTOS actions required to start a process

The process manager has to manage processes with different priorities. For some stimuli, such as those associated with certain exceptional events, it is essential that their processing should be completed within the specified time limits. Other processes may be safely delayed if a more critical process requires service. Consequently, the RTOS has to be able to manage at least two priority levels for system processes:

1. *Clock level* This level of priority is allocated to periodic processes.
2. *Interrupt level* This is the highest priority level. It is allocated to processes that need a very fast response. One of these processes will be the real-time clock process. This process is not required if interrupts are not supported in the system.

A further priority level may be allocated to background processes (such as a self-checking process) that do not need to meet real-time deadlines. These processes are scheduled for execution when processor capacity is available.

Periodic processes must be executed at specified time intervals for data acquisition and actuator control. In most real-time systems, there will be several types of periodic process. Using the timing requirements specified in the application program, the RTOS arranges the execution of periodic processes so that they can all meet their deadlines.

The actions taken by the operating system for periodic process management are shown in Figure 21.19. The scheduler examines the list of periodic processes and selects a process to be executed. The choice depends on the process priority, the process periods, the expected execution times, and the deadlines of the ready processes. Sometimes two processes with different deadlines should be executed at the same clock tick. In such a situation, one process must be delayed. Normally, the system will choose to delay the process with the longest deadline.

Processes that have to respond quickly to asynchronous events may be interrupt-driven. The computer's interrupt mechanism causes control to transfer to a predetermined memory location. This location contains an instruction to jump to a simple and fast interrupt service routine. The service routine disables further interrupts to avoid being interrupted itself. It then discovers the cause of the interrupt and initiates, with a high priority, a process to handle the stimulus causing the interrupt. In some high-speed data acquisition systems, the interrupt handler saves the data that the interrupt signaled was available in a buffer for later processing. Interrupts are then enabled again, and control is returned to the operating system.

At any one time several processes, all with different priorities, could be executed. The process scheduler implements system-scheduling policies that determine the order of process execution. There are two commonly used scheduling strategies:

1. *Nonpreemptive scheduling* After a process has been scheduled for execution, it runs to completion or until it is blocked for some reason, such as waiting for input. This can cause problems if there are processes with different priorities and a high-priority process has to wait for a low-priority process to finish.
2. *Preemptive scheduling* The execution of an executing process may be stopped if a higher-priority process requires service. The higher-priority process preempts the execution of the lower-priority process and is allocated to a processor.

Within these strategies, different scheduling algorithms have been developed. These include round-robin scheduling, where each process is executed in turn; rate monotonic scheduling, where the process with the shortest period (highest frequency) is given priority; and shortest deadline first scheduling, where the process in the queue with the shortest deadline is scheduled (Burns and Wellings 2009).

Information about the process to be executed is passed to the resource manager. The resource manager allocates memory and, in a multiprocessor system, also adds a processor to this process. The process is then placed on the "ready list," a list of processes that are ready for execution. When a processor finishes executing a process and becomes available, the dispatcher is invoked. It scans the ready list to find a process that can be executed on the available processor and starts its execution.

KEY POINTS

- An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is "embedded" in the hardware. Embedded systems are normally real-time systems.
- A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.
- Several standard patterns can be observed in different types of embedded system. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control, and a data-processing pattern.

- Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- A real-time operating system is responsible for process and resource management. It always includes a scheduler, which is the component responsible for deciding which process should be scheduled for execution.

FURTHER READING

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th ed. An excellent and comprehensive text that provides broad coverage of all aspects of real-time systems. (A. Burns and A. Wellings, Addison-Wesley, 2009).

"Trends in Embedded Software Engineering." This article suggests that model-driven development (as discussed in Chapter 5 of this book) will become an important approach to embedded systems development. This is part of a special issue on embedded systems, and other articles, such as the one by Ebert and Jones, are also useful reading. (*IEEE Software,* 26 (3), May–June 2009). http://dx.doi.org/10.1109/MS.2009.80

Real-time systems: Design Principles for Distributed Embedded Applications, 2nd ed. This is a comprehensive textbook on modern real-time systems that may be distributed and mobile systems. The author focuses on hard real-time systems and covers important topics such as Internet connectivity and power management. (H. Kopetz, Springer, 2013).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

21.1. Explain why responsiveness in real time is the critical difference between embedded systems and other software systems.

21.2. Identify possible stimuli and the expected responses for an embedded system that controls a home refrigerator or a domestic washing machine.

21.3. Using the state-based approach to modeling, as discussed in Section 21.1.1, model the operation of the embedded software for a voicemail system that is included in a landline phone.

Train protection system

- The system acquires information on the speed limit of a segment from a trackside transmitter, which continually broadcasts the segment identifier and its speed limit. The same transmitter also broadcasts information on the status of the signal controlling that track segment. The time required to broadcast track segment and signal information is 50 ms.
- The train can receive information from the trackside transmitter when it is within 10 m of a transmitter.
- The maximum train speed is 180 kph.
- Sensors on the train provide information about the current train speed (updated every 250 ms) and the train brake status (updated every 100 ms).
- If the train speed exceeds the current segment speed limit by more than 5 kph, a warning is sounded in the driver's cabin. If the train speed exceeds the current segment speed limit by more than 10 kph, the train's brakes are automatically applied until the speed falls to the segment speed limit. Train brakes should be applied within 100 ms of the time when the excessive train speed has been detected.
- If the train enters a track segment that is signaled with a red light, the train protection system applies the train brakes and reduces the speed to zero. Train brakes should be applied within 100 ms of the time when the red light signal is received.
- The system continually updates a status display in the driver's cabin.

Figure 21.20 Requirements for a train protection system

This should display the number of recorded messages on an LED display and should allow the user to dial-in and listen to the recorded messages.

21.4. What are the commonly used architectural patterns in real-time systems and when are they used?

21.5. Show how the Environmental Control pattern could be used as the basis of the design of a system to control the temperature in a greenhouse. The temperature should be between 10 and 30 degrees Celsius. If it falls below 10 degrees, the heating system should be switched on; if it goes above 30, the windows should be automatically opened.

21.6. Design a process architecture for an environmental monitoring system that collects data from a set of air quality sensors situated around a city. There are 5000 sensors organized into 100 neighborhoods. Each sensor must be interrogated four times per second. When more than 30% of the sensors in a particular neighborhood indicate that the air quality is below an acceptable level, local warning lights are activated. All sensors return the readings to a central computer, which generates reports every 15 minutes on the air quality in the city.

21.7. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). Details are shown in Figure 21.20. Identify the stimuli that must be processed by the on-board train control system and the associated responses to these stimuli.

21.8. Suggest a possible process architecture for this system.

21.9. If a periodic process in the on-board train protection system is used to collect data from the trackside transmitter, how often must it be scheduled to ensure that the system is guaranteed to collect information from the transmitter? Explain how you arrived at your answer.

21.10. With the help of examples, define what a real-time operating system is. Explain how it is different from a conventional operating system. What are the components included in real-time operating systems and what are their responsibilities?

REFERENCES

Berry, G. 1989. "Real-Time Programming: Special-Purpose or General-Purpose Languages." In *Information Processing*, edited by G. Ritter, 89:11–17. Amsterdam: Elsevier Science Publishers.

Bruno, E. J., and G. Bollella. 2009. *Real-Time Java Programming: With Java RTS*. Boston: Prentice-Hall.

Burns, A., and A. Wellings. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.

Cooling, J. 2003. *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.

Douglass, B. P. 1999. *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd ed.* Boston: Addison-Wesley.

———. 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.

Ebert, C., and C. Jones. 2009. "Embedded Software: Facts, Figures and Future." *IEEE Computer* 26 (3): 42–52. doi:10.1109/MC.2009.118.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

———. 1988. "On Visual Formalisms." *Comm. ACM* 31 (5): 514–530. doi:10.1145/42411.42414.

Lee, E A. 2002. "Embedded Software." In *Advances in Computers*, edited by M. Zelkowitz. Vol. 56. London: Academic Press.

Silberschaltz, A., P. B. Galvin, and G. Gagne. 2013. *Operating System Concepts, 9th ed.* New York: John Wiley & Sons.

Stallings, W. 2014. *Operating Systems: Internals and Design Principles, 8th ed.* Boston: Prentice-Hall.

This page intentionally left blank



PART 4

Software Management

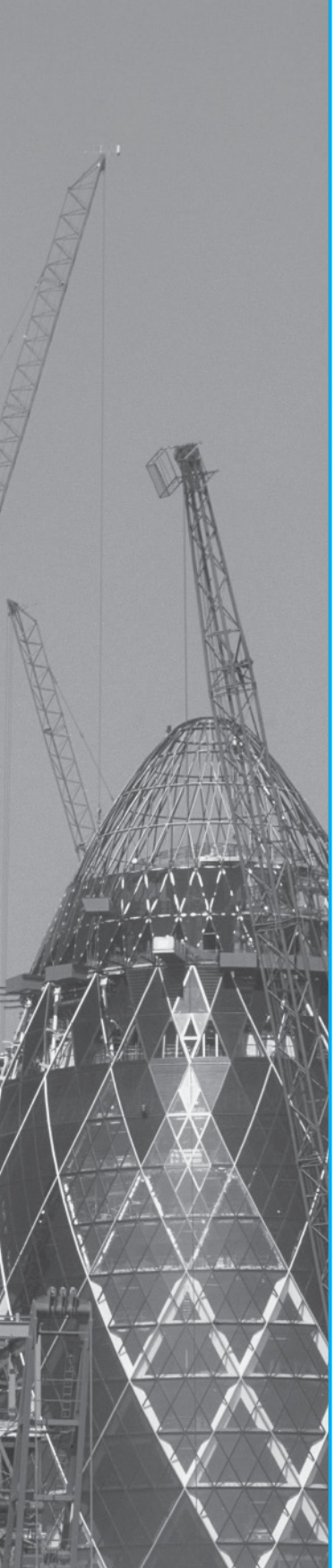
It is sometimes suggested that the key difference between software engineering and other types of programming is that software engineering is a managed process. By this, I mean that the software development takes place within an organization and is subject to a range of schedule, budget and organizational constraints. I introduce a range of management topics in this part of the book with a focus on technical management issues rather than 'softer' management issues such as people management, or the more strategic management of enterprise systems.

Chapters 22 and 23 focus on the essential project management activities, planning, risk management and people management. Chapter 22 introduces software project management and its first major section is concerned with risk management where managers identify what might go wrong and plan what they might do about it. This chapter also includes sections on people management and team working.

Chapter 23 covers project planning and estimation. I introduce bar charts as fundamental planning tools and explain why plan-driven development will remain an important development approach, in spite of the success of agile methods. I also discuss issues that influence the price charged for a system and techniques of software cost estimation. I use the COCOMO family of cost models to describe algorithmic cost modeling and explain the benefits and disadvantages of algorithmic approaches.

Chapters 24 explains the basics of software quality management, as practised in large projects. Quality management is concerned with processes and techniques for ensuring and improving the quality of software. I discuss the importance of standards in quality management, the use of reviews and inspections in the quality assurance process. The final section of this chapter covers software measurement and I discuss the benefits and problems in using metrics and software data analytics in quality management.

Finally, Chapter 25 discusses configuration management, a critical issue for all large systems. However, the need for configuration management is not always obvious to students who have only been concerned with personal software development, so I describe the various aspects of this topic here, including version management, system building, change management and release management. I explain why continuous integration or daily system building is important. An important change in this edition is the inclusion of new material on distributed version management systems, such as Git, which are being increasingly used to support software engineering by distributed teams.



22

Project management

Objectives

The objective of this chapter is to introduce software project management and two important management activities, namely, risk management and people management. When you have read the chapter you will:

- know the principal tasks of software project managers;
- have been introduced to the notion of risk management and some of the risks that can arise in software projects;
- understand factors that influence personal motivation and what these might mean for software project managers;
- understand key issues that influence team working, such as team composition, organization, and communication.

Contents

22.1 Risk management
22.2 Managing people
22.3 Teamwork

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: The software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers.

The success criteria for project management obviously vary from project to project, but, for most projects, important goals are:

- to deliver the software to the customer at the agreed time;
- to keep overall costs within budget;
- to deliver software that meets the customer's expectations;
- to maintain a coherent and well-functioning development team.

These goals are not unique to software engineering but are the goals of all engineering projects. However, software engineering is different from other types of engineering in a number of ways that make software management particularly challenging. Some of these differences are:

1. *The product is intangible* A manager of a shipbuilding or a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible—parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress by looking at the artifact that is being constructed. Rather, they rely on others to produce evidence that they can use to review the progress of the work.
2. *Large software projects are often "one-off" projects* Every large software development project is unique because every environment where software is developed is, in some ways, different from all others. Even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make experience obsolete. Lessons learned from previous projects may not be readily transferable to new projects.
3. *Software processes are variable and organization-specific* The engineering process for some types of system, such as bridges and buildings, is well understood. However, different companies use quite different software development processes. We cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project or when completely new software is being developed.

Because of these issues, it is not surprising that some software projects are late, overbudget, and behind schedule. Software systems are often new, very complex, and technically innovative. Schedule and cost overruns are also common in other

engineering projects, such as new transport systems, that are complex and innovative. Given the difficulties involved, it is perhaps remarkable that so many software projects are delivered on time and to budget.

It is impossible to write a standard job description for a software project manager. The job varies tremendously depending on the organization and the software being developed. Some of the most important factors that affect how software projects are managed are:

1. *Company size* Small companies can operate with informal management and team communications and do not need formal policies and management structures. They have less management overhead than larger organizations. In larger organizations, management hierarchies, formal reporting and budgeting, and approval processes must be followed.
2. *Software customers* If the customer is an internal customer (as is the case for software product development), then customer communications can be informal and there is no need to fit in with the customer's ways of working. If custom software is being developed for an external customer, agreement has to be reached on more formal communication channels. If the customer is a government agency, the software company must operate according to the agency's policies and procedures, which are likely to be bureaucratic.
3. *Software size* Small systems can be developed by a small team, which can get together in the same room to discuss progress and other management issues. Large systems usually need multiple development teams that may be geographically distributed and in different companies. The project manager has to coordinate the activities of these teams and arrange for them to communicate with each other.
4. *Software type* If the software being developed is a consumer product, formal records of project management decisions are unnecessary. On the other hand, if a safety-critical system is being developed, all project management decisions should be recorded and justified as these may affect the safety of the system.
5. *Organizational culture* Some organizations have a culture that is based on supporting and encouraging individuals, while others are group focused. Large organizations are often bureaucratic. Some organizations have a culture of taking risks, whereas others are risk averse.
6. *Software development processes* Agile processes typically try to operate with "lightweight" management. More formal processes require management monitoring to ensure that the development team is following the defined process.

These factors mean that project managers in different organizations may work in quite different ways. However, a number of fundamental project management activities are common to all organizations:

1. *Project planning* Project managers are responsible for planning, estimating, and scheduling project development and assigning people to tasks. They supervise

the work to ensure that it is carried out to the required standards, and they monitor progress to check that the development is on time and within budget.

2. *Risk management* Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise.
3. *People management* Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.
4. *Reporting* Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews.
5. *Proposal writing* The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded.

Project planning is an important topic in its own right, which I discuss in Chapter 23. In this chapter, I focus on risk management and people management.

22.1 Risk management

Risk management is one of the most important jobs for a project manager. You can think of a risk as something that you'd prefer not to have happen. Risks may threaten the project, the software that is being developed, or the organization. Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks (Hall 1998; Ould 1999).

Risks can be categorized according to type of risk (technical, organizational, etc.), as I explain in Section 22.1.1. A complementary classification is to classify risks according to what these risks affect:

1. Project risks affect the project schedule or resources. An example of a project risk is the loss of an experienced system architect. Finding a replacement architect with appropriate skills and experience may take a long time; consequently, it will take longer to develop the software design than originally planned.
2. Product risks affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform

as expected. This may affect the overall performance of the system so that it is slower than expected.

3. Business risks affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Of course, these risk categories overlap. An experienced engineer's decision to leave a project, for example, presents a *project risk* because the software delivery schedule will be affected. It inevitably takes time for a new project member to understand the work that has been done, so he or she cannot be immediately productive. Consequently, the delivery of the system may be delayed. The loss of a team member can also be a *product risk* because a replacement may not be as experienced and so could make programming errors. Finally, losing a team member can be a *business risk* because an experienced engineer's reputation may be a critical factor in winning new contracts.

For large projects, you should record the results of the risk analysis in a risk register along with a consequence analysis. This sets out the consequences of the risk for the project, product, and business. Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage. For small projects, formal risk recording may not be required, but the project manager should be aware of them.

The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed. However, there are also common risks that are independent of the type of software being developed. These can occur in any software development project. Some examples of these common risks are shown in Figure 22.1.

Software risk management is important because of the inherent uncertainties in software development. These uncertainties stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills. You have to anticipate risks, understand their impact on the project, the product, and the business, and take steps to avoid these risks. You may need to draw up contingency plans so that, if the risks do occur, you can take immediate recovery action.

An outline of the process of risk management is presented in Figure 22.2. It involves several stages:

1. *Risk identification* You should identify possible project, product, and business risks.
2. *Risk analysis* You should assess the likelihood and consequences of these risks.
3. *Risk planning* You should make plans to address the risk, either by avoiding it or by minimizing its effects on the project.
4. *Risk monitoring* You should regularly assess the risk and your plans for risk mitigation and revise these plans when you learn more about the risk.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of company management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
Software tool underperformance	Product	Software tools that support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Figure 22.1 Examples of common project, product, and business risks

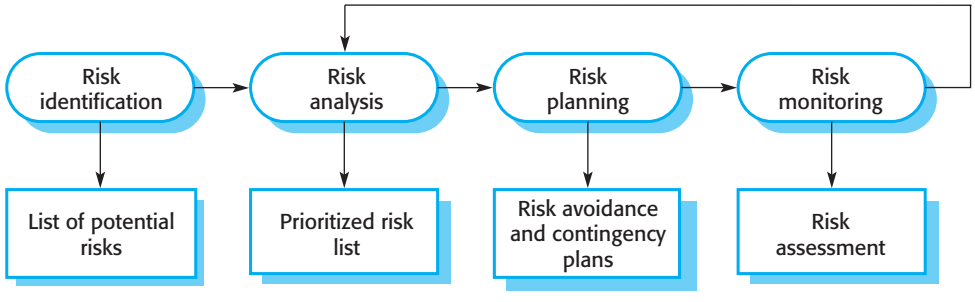


Figure 22.2 The risk management process

For large projects, you should document the outcomes of the risk management process in a risk management plan. This should include a discussion of the risks faced by the project, an analysis of these risks, and information on how you plan to manage the risk if it seems likely to be a problem.

The risk management process is an iterative process that continues throughout a project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes

available, you have to re-analyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.

Risk management in agile development is less formal. The same fundamental activities should still be followed and risks discussed, although these may not be formally documented. Agile development reduces some risks, such as risks from requirements changes. However, agile development also has a downside. Because of its reliance on people, staff turnover can have significant effects on the project, product, and business. Because of the lack of formal documentation and its reliance on informal communications, it is very hard to maintain continuity and momentum if key people leave the project.

22.1.1 Risk identification

Risk identification is the first stage of the risk management process. It is concerned with identifying the risks that could pose a major threat to the software engineering process, the software being developed, or the development organization. Risk identification may be a team process in which a team gets together to brainstorm possible risks. Alternatively, project managers may identify risks based on their experience of what went wrong on previous projects.

As a starting point for risk identification, a checklist of different types of risk may be used. Six types of risk may be included in a risk checklist:

1. Estimation risks arise from the management estimates of the resources required to build the system.
2. Organizational risks arise from the organizational environment where the software is being developed.
3. People risks are associated with the people in the development team.
4. Requirements risks come from changes to the customer requirements and the process of managing the requirements change.
5. Technology risks come from the software or hardware technologies that are used to develop the system.
6. Tools risks come from the software tools and other support software used to develop the system.

Figure 22.3 shows examples of possible risks in each of these categories. When you have finished the risk identification process, you should have a long list of risks that could occur and that could affect the product, the process, and the business. You then need to prune this list to a manageable size. If you have too many risks, it is practically impossible to keep track of all of them.

Risk type	Possible risks
Estimation	1. The time required to develop the software is underestimated. 2. The rate of defect repair is underestimated. 3. The size of the software is underestimated.
Organizational	4. The organization is restructured so that different management are responsible for the project. 5. Organizational financial problems force reductions in the project budget.
People	6. It is impossible to recruit staff with the skills required. 7. Key staff are ill and unavailable at critical times. 8. Required training for staff is not available.
Requirements	9. Changes to requirements that require major design rework are proposed. 10. Customers fail to understand the impact of requirements changes.
Technology	11. The database used in the system cannot process as many transactions per second as expected. 12. Faults in reusable software components have to be repaired before these components are reused.
Tools	13. The code generated by software code generation tools is inefficient. 14. Software tools cannot work together in an integrated way.

Figure 22.3 Examples of different types of risk

22.1.2 Risk analysis

During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk. There is no easy way to do so. You have to rely on your judgment and experience of previous projects and the problems that arose in them. It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, you should assign the risk to one of a number of bands:

1. The probability of the risk might be assessed as insignificant, low, moderate, high, or very high.
2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.

You may then tabulate the results of this analysis process using a table ordered according to the seriousness of the risk. Figure 22.4 illustrates this for the risks that I have identified in Figure 22.3. Obviously, the assessment of probability and seriousness is arbitrary here. To make this assessment, you need

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (5).	Low	Catastrophic
It is impossible to recruit staff with the skills required (6).	High	Catastrophic
Key staff are ill at critical times in the project (7).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused (12).	Moderate	Serious
Changes to requirements that require major design rework are proposed (9).	Moderate	Serious
The organization is restructured so that different managements are responsible for the project (4).	High	Serious
The database used in the system cannot process as many transactions per second as expected (11).	Moderate	Serious
The time required to develop the software is underestimated (1).	High	Serious
Software tools cannot be integrated (14).	High	Tolerable
Customers fail to understand the impact of requirements changes (10).	Moderate	Tolerable
Required training for staff is not available (8).	Moderate	Tolerable
The rate of defect repair is underestimated (2).	Moderate	Tolerable
The size of the software is underestimated (3).	High	Tolerable
Code generated by code generation tools is inefficient (13).	Moderate	Insignificant

Figure 22.4 Risk types and examples

detailed information about the project, the process, the development team, and the organization.

Of course, both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and as risk management plans are implemented. You should therefore update this table during each iteration of the risk management process.

Once the risks have been analyzed and ranked, you should assess which of these risks are most significant. Your judgment must depend on a combination of the probability of the risk arising and the effects of that risk. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.

Boehm (Boehm 1988) recommends identifying and monitoring the "top 10" risks. However, I think that the right number of risks to monitor must depend on the project. It might be 5 or it might be 15. From the risks identified in Figure 22.4, I think that it is appropriate to consider the eight risks that have catastrophic or serious consequences (Figure 22.5).

22.1.3 Risk planning

The risk planning process develops strategies to manage the key risks that threaten the project. For each risk, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in the risk occurs. You should also think about the information that you need to collect while monitoring the project so that emerging problems can be detected before they become serious.

In risk planning, you have to ask "what-if" questions that consider both individual risks, combinations of risks, and external factors that affect these risks. For example, questions that you might ask are:

1. What if several engineers are ill at the same time?
2. What if an economic downturn leads to budget cuts of 20% for the project?
3. What if the performance of open-source software is inadequate and the only expert on that open-source software leaves?
4. What if the company that supplies and maintains software components goes out of business?
5. What if the customer fails to deliver the revised requirements as predicted?

Based on the answers to these "what-if" questions, you may devise strategies for managing the risks. Figure 22.5 shows possible risk management strategies that have been identified for the key risks (i.e., those that are serious or intolerable) shown in Figure 22.4. These strategies fall into three categories:

1. *Avoidance strategies* Following these strategies means that the probability that the risk will arise is reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 22.5.
2. *Minimization strategies* Following these strategies means that the impact of the risk is reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 22.5.
3. *Contingency plans* Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 22.5.

You can see a clear analogy here with the strategies used in critical systems to ensure reliability, security, and safety, where you must avoid, tolerate, or recover from failures. Obviously, it is best to use a strategy that avoids the risk. If this is not possible, you should use a strategy that reduces the chances that the risk will have serious effects. Finally, you should have strategies in place to

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of automated code generation.

Figure 22.5 Strategies to help manage risk

cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.

22.1.4 Risk monitoring

Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed. You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable. You should also think about whether or not the effects of the risk have changed. To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are obviously dependent on the types of risk. Figure 22.6 gives some examples of factors that may be helpful in assessing these risk types.

You should monitor risks regularly at all stages in a project. At every management review, you should consider and discuss each of the key risks separately. You should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

Figure 22.6 Risk indicators

22.2 Managing people

The people working in a software organization are its greatest assets. It is expensive to recruit and retain good people, and it is up to software managers to ensure that the engineers working on a project are as productive as possible. In successful companies and economies, this productivity is achieved when people are respected by the organization and are assigned responsibilities that reflect their skills and experience.

It is important that software project managers understand the technical issues that influence the work of software development. Unfortunately, however, good software engineers are not always good people managers. Software engineers often have strong technical skills but may lack the softer skills that enable them to motivate and lead a project development team. As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

There are four critical factors that influence the relationship between a manager and the people that he or she manages:

1. *Consistency* All the people in a project team should be treated in a comparable way. No one expects all rewards to be identical, but people should not feel that their contribution to the organization is undervalued.
2. *Respect* Different people have different skills, and managers should respect these differences. All members of the team should be given an opportunity to make a contribution. In some cases, of course, you will find that people simply don't fit into a team and they cannot continue, but it is important not to jump to conclusions about them at an early stage in the project.

3. *Inclusion* People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a working environment where all views, even those of the least experienced staff, are considered.

4. *Honesty* As a manager, you should always be honest about what is going well and what is going badly in the team. You should also be honest about your level of technical knowledge and be willing to defer to staff with more knowledge when necessary. If you try to cover up ignorance or problems, you will eventually be found out and will lose the respect of the group.

Practical people management has to be based on experiences so my aim in this section and the following section on teamwork is to raise awareness of the most important issues that project managers may have to deal with.

22.2.1 Motivating people

As a project manager, you need to motivate the people who work with you so that they will contribute to the best of their abilities. In practice, motivation means organizing work and its environment to encourage people to work as effectively as possible. If people are not motivated, they will be less interested in the work they are doing. They will work slowly, be more likely to make mistakes, and will not contribute to the broader goals of the team or the organization.

To provide this encouragement, you should understand a little about what motivates people. Maslow (Maslow 1954) suggests that people are motivated by satisfying their needs. These needs are arranged in a series of levels, as shown in Figure 22.7. The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, and the need to feel secure in an environment. Social need is concerned with the need to feel part of a social grouping. Esteem need represents the need to feel respected by others, and self-realization need is concerned with personal development. People need to satisfy lower-level needs such as hunger before the more abstract, higher-level needs.

People working in software development organizations are not usually hungry, thirsty, or physically threatened by their environment. Therefore, making sure that peoples' social, esteem, and self-realization needs are satisfied is most important from a management point of view.

1. To satisfy social needs, you need to give people time to meet their co-workers and provide places for them to meet. Software companies such as Google provide social space in their offices for people to get together. This is relatively easy when all of the members of a development team work in the same place, but, increasingly, team members are not located in the same building or even the same town or state. They may work for different organizations or from home most of the time.

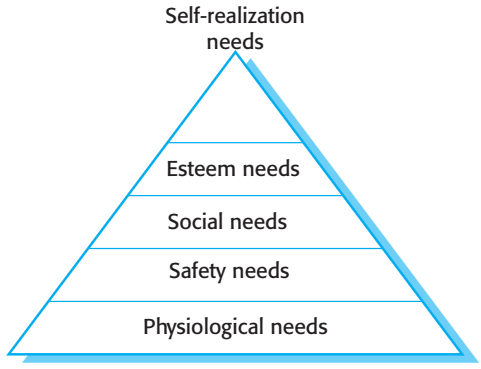


Figure 22.7 Human needs hierarchy

Social networking systems and teleconferencing can be used for remote communications, but my experience with these systems is that they are most effective when people already know each other. You should arrange some face-to-face meetings early in the project so that people can directly interact with other members of the team. Through this direct interaction, people become part of a social group and accept the goals and priorities of that group.

2. To satisfy esteem needs, you need to show people that they are valued by the organization. Public recognition of achievements is a simple and effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.

3. Finally, to satisfy self-realization needs, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks, and provide opportunities for training and development where people can enhance their skills. Training is an important motivating influence as people like to gain new knowledge and learn new skills.

Maslow's model of motivation is helpful up to a point, but I think that a problem with it is that it takes an exclusively personal viewpoint on motivation. It does not take adequate account of the fact that people feel themselves to be part of an organization, a professional group, and one or more cultures. Being a member of a cohesive group is highly motivating for most people. People with fulfilling jobs often like to go to work because they are motivated by the people they work with and the work that they do. Therefore, as a manager, you also have to think about how a group as a whole can be motivated. I discuss this and other teamwork issues in Section 22.3.

In Figure 22.8, I illustrate a problem of motivation that managers often have to face. In this example, a competent group member loses interest in the work and in the group as a whole. The quality of her work falls and becomes unacceptable. This situation has to be dealt with quickly. If you don't sort out the problem, the other group members will become dissatisfied and feel that they are doing an unfair share of the work.

Case study: Motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of six developers that can develop new products based on the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team, and creative new ideas are developed. The team decides to develop a system that a user can initiate and control the alarm system from a cell phone or tablet computer. However, some months into the project, Alice notices that Dorothy, a hardware expert, starts coming into work late, that the quality of her work is deteriorating, and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials of any problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity to use these skills. Basically, she is working as a C programmer on the alarm system software.

While she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Figure 22.8 Individual motivation

In this example, Alice tries to find out if Dorothy's personal circumstances could be the problem. Personal difficulties commonly affect motivation because people cannot therefore concentrate on their work. You may have to give them time and support to resolve these issues, although you also have to make it clear that they still have a responsibility to their employer.

Dorothy's motivation problem is one that can arise when projects develop in an unexpected direction. People who expect to do one type of work may end up doing something completely different. In those circumstances, you may decide that the team member should leave the team and find opportunities elsewhere. In this example, however, Alice decides to try to convince Dorothy that broadening her experience is a positive career step. She gives Dorothy more design autonomy and organizes training courses in software engineering that will give her more opportunities after her current project has finished.

Psychological personality type also influences motivation. Bass and Dunteman (Bass and Dunteman 1963) identified three classifications for professional workers:

1. *Task-oriented people,* who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.



The People Capability Maturity Model

The People Capability Maturity Model (P-CMM) is a framework for assessing how well organizations manage the development of their staff. It highlights best practice in people management and provides a basis for organizations to improve their people management processes. It is best suited to large rather than small, informal companies.

http://software-engineering-book.com/web/people-cmm/

2. *Self-oriented people,* who are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals. They often have longer-term goals, such as career progression, that motivate them, and they wish to be successful in their work to help realize these goals.
3. *Interaction-oriented people,* who are motivated by the presence and actions of co-workers. As more and more attention is paid to user interface design, interaction-oriented individuals are becoming more involved in software engineering.

Research has shown that interaction-oriented personalities usually like to work as part of a group, whereas task-oriented and self-oriented people usually prefer to act as individuals. Women are more likely to be interaction-oriented than men are. They are often more effective communicators. I discuss the mix of these different personality types in groups in the case study shown later in Figure 22.10.

Each individual's motivation is made up of elements of each class, but one type of motivation is usually dominant at any one time. However, individuals can change. For example, technical people who feel they are not being properly rewarded can become self-oriented and put personal interests before technical concerns. If a group works particularly well, self-oriented people can become more interaction-oriented.

22.3 Teamwork

Most professional software is developed by project teams that range in size from two to several hundred people. However, as it is impossible for everyone in a large group to work together on a single problem, large teams are usually split into a number of smaller groups. Each group is responsible for developing part of the overall system. The best size for a software engineering group is 4 to 6 members, and they should never have more than 12 members. When groups are small, communication problems are reduced. Everyone knows everyone else, and the whole group can get around a table for a meeting to discuss the project and the software that they are developing.

Putting together a group that has the right balance of technical skills, experience, and personalities is a critical management task. However, successful groups are more than simply a collection of individuals with the right balance of skills. A good group is cohesive and thinks of itself as a strong, single unit. The people involved are motivated by the success of the group as well as by their own personal goals.

In a cohesive group, members think of the group as more important than the individuals who are group members. Members of a well-led, cohesive group are loyal to the group. They identify with group goals and other group members. They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

The benefits of creating a cohesive group are:

1. *The group can establish its own quality standards* Because these standards are established by consensus, they are more likely to be observed than external standards imposed on the group.
2. *Individuals learn from and support each other* Group members learn by working together. Inhibitions caused by ignorance are minimized as mutual learning is encouraged.
3. *Knowledge is shared* Continuity can be maintained if a group member leaves. Others in the group can take over critical tasks and ensure that the project is not unduly disrupted.
4. *Refactoring and continual improvement is encouraged* Group members work collectively to deliver high-quality results and fix problems, irrespective of the individuals who originally created the design or program.

Good project managers should always try to encourage group cohesiveness. They may try to establish a sense of group identity by naming the group and establishing a group identity and territory. Some managers like explicit group-building activities such as sports and games, although these are not always popular with group members. Social events for group members and their families are a good way to bring people together.

One of the most effective ways of promoting cohesion is to be inclusive. That is, you should treat group members as responsible and trustworthy, and make information freely available. Sometimes managers feel that they cannot reveal certain information to everyone in the group. This invariably creates a climate of mistrust. An effective way of making people feel valued and part of a group is to make sure that they know what is going on.

You can see an example in the case study in Figure 22.9. Alice arranges regular informal meetings where she tells the other group members what is going on. She makes a point of involving people in the product development by asking them to come up with new ideas derived from their own family experiences. The “away

Case study: Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As her company is developing a new product, she takes the opportunity to involve all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing, and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an "away day" for the group where the team spends two days on "technology updating." Each team member prepares an update on a relevant technology and presents it to the group. This is an offsite meeting, and plenty of time is scheduled for discussion and social interaction.

Figure 22.9 Group cohesion

days" are also good ways of promoting cohesion: People relax together while they help each other learn about new technologies.

Whether or not a group is effective depends, to some extent, on the nature of the project and the organization doing the work. If an organization is in a state of turmoil with constant reorganizations and job insecurity, it is difficult for team members to focus on software development. Similarly, if a project keeps changing and is in danger of cancellation, people lose interest in it.

Given a stable organizational and project environment, the three factors that have the biggest effect on team working are:

1. *The people in the group* You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing, and documentation.
2. *The way the group is organized* A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.
3. *Technical and managerial communications* Good communication between group members, and between the software engineering team and other project stakeholders, is essential.

As with all management issues, getting the right team cannot guarantee project success. Too many other things can go wrong, including changes to the business and the business environment. However, if you don't pay attention to group composition, organization, and communications, you increase the likelihood that your project will run into difficulties.

22.3.1 Selecting group members

A manager or team leader's job is to create a cohesive group and organize that group so that they work together effectively. This task involves selecting a group with the right balance of technical skills and personalities. Sometimes people are hired from outside the organization; more often, software engineering groups are put together from current employees who have experience on other projects. Managers rarely have a completely free hand in team selection. They often have to use the people who are available in the company, even if they are not the ideal people for the job.

Many software engineers are motivated primarily by their work. Software development groups, therefore, are often composed of people who have their own ideas about how technical problems should be solved. They want to do the best job possible, so they may deliberately redesign systems that they think can be improved and add extra system features that are not in the system requirements. Agile methods encourage engineers to take the initiative to improve the software. However, sometimes this means that time is spent doing things that aren't really needed and that different engineers compete to rewrite each other's code.

Technical knowledge and ability should not be the only factor used to select group members. The "competing engineers" problem can be reduced if the people in the group have complementary motivations. People who are motivated by the work are likely to be the strongest technically. People who are self-oriented will probably be best at pushing the work forward to finish the job. People who are interaction-oriented help facilitate communications within the group. I think that it is particularly important to have interaction-oriented people in a group. They like to talk to people and can detect tensions and disagreements at an early stage, before these problems have a serious impact on the group.

In the case study in Figure 22.10, I have suggested how Alice, the project manager, has tried to create a group with complementary personalities. This particular group has a good mix of interaction- and task-oriented people, but I have already discussed, in Figure 22.8, how Dorothy's self-oriented personality has caused problems because she has not been doing the work that she expected. Fred's part-time role in the group as a domain expert might also be a problem. He is mostly interested in technical challenges, so he may not interact well with other group members. The fact that he is not always part of the team means that he may not fully relate to the team's goals.

It is sometimes impossible to choose a group with complementary personalities. If this is the case, the project manager has to control the group so that individual goals do not take precedence over organizational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely to develop when group members are given instructions without being aware of the part that their task plays in the overall project.

For example, say a software engineer takes over the development of a system and notices that possible improvements could be made to the design. If he or she implements these improvements without understanding the rationale for the original design, any changes, though well-intentioned, might have adverse implications for

Case study: Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly being promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Chun—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fiona—task-oriented
Fred—task-oriented
Hassan—interaction-oriented

Figure 22.10 Group composition

other parts of the system. If all the members of the group are involved in the design from the start, they are more likely to understand why design decisions have been made. They may then identify with these decisions rather than oppose them.

22.2.3 Group organization

The way a group is organized affects the group's decisions, the ways information is exchanged, and the interactions between the development group and external project stakeholders. Important organizational questions for project managers include the following:

1. Should the project manager be the technical leader of the group? The technical leader or system architect is responsible for the critical technical decisions made during software development. Sometimes the project manager has the skill and experience to take on this role. However, for large projects, it is best to separate technical and managerial roles. The project manager should appoint a senior engineer to be the project architect, who will take responsibility for technical leadership.
2. Who will be involved in making critical technical decisions, and how will these decisions be made? Will decisions be made by the system architect or the project manager or by reaching consensus among a wider range of team members?
3. How will interactions with external stakeholders and senior company management be handled? In many cases, the project manager will be responsible for these interactions, assisted by the system architect if there is one. However, an alternative organizational model is to create a dedicated role concerned with external liaison and appoint someone with appropriate interaction skills to that role.



Hiring the right people

Project managers are often responsible for selecting the people in the organization who will join their software engineering team. Getting the best possible people in this process is very important as poor selection decisions may be a serious risk to the project.

Key factors that should influence the selection of staff are education and training, application domain and technology experience, communication ability, adaptability, and problem solving ability.

http://software-engineering-book.com/web/people-selection/

4. How can groups integrate people who are not co-located? It is now common for groups to include members from different organizations and for people to work from home as well as in a shared office. This change has to be considered in group decision-making processes.

5. How can knowledge be shared across the group? Group organization affects information sharing as certain methods of organization are better for sharing than others. However, you should avoid too much information sharing as people become overloaded and excessive information distracts them from their work.

Small programming groups are usually organized in an informal way. The group leader gets involved in the software development with the other group members. In an informal group, the group as a whole discusses the work to be carried out, and tasks are allocated according to ability and experience. More senior group members may be responsible for the architectural design. However, detailed design and implementation is the responsibility of the team member who is allocated to a particular task.

Agile development teams are always informal groups. Agile enthusiasts claim that formal structure inhibits information exchange. Many decisions that are usually seen as management decisions (such as decisions on schedule) may be devolved to group members. However, there still needs to be a project manager who is responsible for strategic decision making and communications outside of the group.

Informal groups can be very successful, particularly when most group members are experienced and competent. Such a group makes decisions by consensus, which improves cohesiveness and performance. However, if a group is composed mostly of inexperienced or incompetent members, informality can be a hindrance. With no experienced engineers to direct the work, the result can be a lack of coordination between group members and, possibly, eventual project failure.

In hierarchical groups the group leader is at the top of the hierarchy. He or she has more formal authority than the group members and so can direct their work. There is a clear organizational structure, and decisions are made toward the top of the hierarchy and implemented by people lower down. Communications are primarily instructions from senior staff; the people at lower levels of the hierarchy have relatively little communication with the managers at the upper levels.

Hierarchical groups can work well when a well-understood problem can be easily broken down into software components that can be developed in different parts of the hierarchy. This grouping allows for rapid decision making, which is why military organizations follow this model. However, it rarely works well for complex software engineering. In software development, effective team communications at all levels is essential:

1. Changes to the software often require changes to several parts of the system, and this requires discussion and negotiation at all levels in the hierarchy.
2. Software technologies change so fast that more junior staff may know more about new technologies than experienced staff. Top-down communications may mean that the project manager does not find out about the opportunities of using these new technologies. More junior staff may become frustrated because of what they see as old-fashioned technologies being used for development.

A major challenge facing project managers is the difference in technical ability between group members. The best programmers may be up to 25 times more productive than the worst programmers. It makes sense to use these "super-programmers" in the most effective way and to provide them with as much support as possible.

At the same time, focusing on the super-programmers can be demotivating for other group members who are resentful that they are not given responsibility. They may be concerned that this will affect their career development. Furthermore, if a "super-programmer" leaves the company, the impact on a project can be huge. Therefore, adopting a group model that is based on individual experts can pose significant risks.

22.3.3 Group communications

It is absolutely essential that group members communicate effectively and efficiently with each other and with other project stakeholders. Group members must exchange information on the status of their work, the design decisions that have been made, and changes to previous design decisions. They have to resolve problems that arise with other stakeholders and inform these stakeholders of changes to the system, the group, and delivery plans. Good communication also helps strengthen group cohesiveness. Group members come to understand the motivations, strengths, and weaknesses of other people in the group.

The effectiveness and efficiency of communications are influenced by:

1. *Group size* As a group gets bigger, it gets harder for members to communicate effectively. The number of one-way communication links is $n * (n - 1)$, where n is the group size, so, with a group of eight members, there are 56 possible communication pathways. This means that it is quite possible that some people will rarely communicate with each other. Status differences between group members mean that communications are often one-way. Managers and experienced engineers tend to dominate communications with less experienced staff, who may be reluctant to start a conversation or make critical remarks.



The physical work environment

Group communications and individual productivity are both affected by the team's working environment. Individual workspaces are better for concentration on detailed technical work as people are less likely to be distracted by interruptions. However, shared workspaces are better for communications. A well-designed work environment takes both of these needs into account.

http://software-engineering-book.com/web/workspace/

2. *Group structure* People in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure. In hierarchical groups, communications tend to flow up and down the hierarchy. People at the same level may not talk to each other. This is a particular problem in a large project with several development groups. If people working on different subsystems only communicate through their managers, then there are more likely to be delays and misunderstandings.

3. *Group composition* People with the same personality types (discussed in Section 22.2) may clash, and, as a result, communications can be inhibited. Communication is also usually better in mixed-sex groups than in single-sex groups (Marshall and Heslin 1975). Women are often more interaction-oriented than men and may act as interaction controllers and facilitators for the group.

4. *The physical work environment* The organization of the workplace is a major factor in facilitating or inhibiting communications. While some companies use standard open-plan offices for their staff, others invest in providing a workspace that includes a mixture of private and group working areas. This allows for both collaborative activities and individual development that require a high level of concentration.

5. *The available communication channels* There are many different forms of communication—face to face, email messages, formal documents, telephone, and technologies such as social networking and wikis. As project teams become increasingly distributed, with team members working remotely, you need to make use of interaction technologies, such as conferencing systems, to facilitate group communications.

Project managers usually work to tight deadlines, and, consequently, they often try to use communication channels that don't take up too much of their time. They may rely on meetings and formal documents to pass on information to project staff and stakeholders and send long emails to project staff. Unfortunately, while this may be an efficient approach to communication from a project manager's perspective, it is not usually very effective. There are often good reasons why people can't attend meetings, and so they don't hear the presentation. People do not have time to read long documents and emails that are not directly relevant to their work. When several versions of the same document are produced, readers find it difficult to keep track of the changes.

Effective communication is achieved when communications are two-way and the people involved can discuss issues and information and establish a common understanding of proposals and problems. All this can be done through meetings, although these meetings are often dominated by powerful personalities. Informal discussions when a manager meets with the team for coffee are sometimes more effective.

More and more project teams include remote members, which also makes meetings more difficult. To involve them in communications, you may make use of wikis and blogs to support information exchange. Wikis support the collaborative creation and editing of documents, and blogs support threaded discussions about questions and comments made by group members. Wikis and blogs allow project members and external stakeholders to exchange information, irrespective of their location. They help manage information and keep track of discussion threads, which often become confusing when conducted by email. You can also use instant messaging and teleconferences, which can be easily arranged, to resolve issues that need discussion.

KEY POINTS

- Good software project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative, so there is no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- Risk management involves identifying and assessing major project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage, or deal with likely risks if or when they arise.
- People management involves choosing the right people to work on a project and organizing the team and its working environment so that they are as productive as possible.
- People are motivated by interaction with other people, by the recognition of management and their peers, and by being given opportunities for personal development.
- Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized, and the communication between group members.
- Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities, and available communication channels.

FURTHER READING

The Mythical Man Month: Essays on Software Engineering (Anniversary Edition). The problems of software management have remained largely unchanged since the 1960s, and this is one of the best books on the topic. It presents an interesting and readable account of the management of one of the first very large software projects, the IBM OS/360 operating system. The anniversary edition (published 20 years after the original edition in 1975) includes other classic papers by Brooks. (F. P. Brooks, 1995, Addison-Wesley).

Peopleware: Productive Projects and Teams, 2nd ed. This now classic book focuses on the importance of treating people properly when managing software projects. It is one of the few books that recognizes how the place where people work influences communications and productivity. Strongly recommended. (T. DeMarco and T. Lister, 1999, Dorset House).

Waltzing with Bears: Managing Risk on Software Projects. A very practical and easy-to-read introduction to risks and risk management. (T. DeMarco and T. Lister, 2003, Dorset House).

Effective Project Management: Traditional, Agile, Extreme. 2014 (7th ed.). This is a textbook on project management in general rather than software project management. It is based on the so-called PMBOK (Project Management Body of Knowledge) and, unlike most books on this topic, discusses PM techniques for agile projects. (R. K. Wysocki, 2014).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

22.1. Explain why the intangibility of software systems poses special problems for software project management.

22.2. Explain how company size and software size are factors that affect software project management.

22.3. Using reported instances of project problems in the literature, list management difficulties and errors that occurred in these failed programming projects. (I suggest that you start with *The Mythical Man Month*, as suggested in Further Reading.)

22.4. In addition to the risks shown in Figure 22.1, identify at least six other possible risks that could arise in software projects.

22.5. What is risk monitoring? How can risks be monitored? List a few examples of types of risks and their potential indicators.

22.6. Fixed-price contracts, where the contractor bids a fixed price to complete a system development, may be used to move project risk from client to contractor. If anything goes wrong, the contractor has to pay. Suggest how the use of such contracts may increase the likelihood that product risks will arise.

22.7. Explain why keeping all members of a group informed about progress and technical decisions in a project can improve group cohesiveness.

22.8. What qualities of a cohesive group's members make the group robust? List out the key benefits of creating a cohesive group.

22.9. Write a case study in the style used here to illustrate the importance of communications in a project team. Assume that some team members work remotely and that it is not possible to get the whole team together at short notice.

22.10. Your manager asks you to deliver software to a schedule that you know can only be met by asking your project team to work unpaid overtime. All team members have young children. Discuss whether you should accept this demand from your manager or whether you should persuade your team to give their time to the organization rather than to their families. What factors might be significant in your decision?

REFERENCES

Bass, B. M., and G. Dunteman. 1963. "Behaviour in Groups as a Function of Self, Interaction and Task Orientation." *J. Abnorm. Soc. Psychology*. 66 (4): 19–28. doi:10.1037/h0042764.

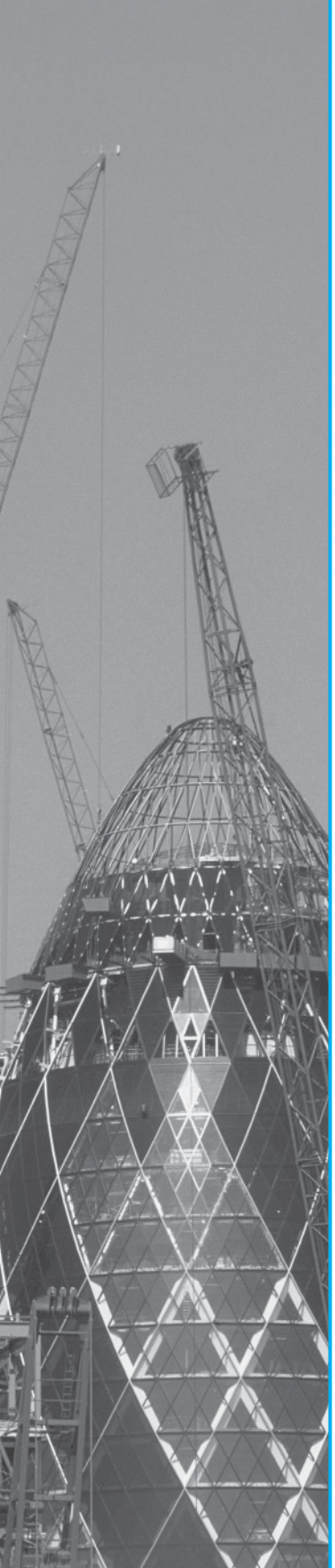
Boehm, B. W. 1988. "A Spiral Model of Software Development and Enhancement." *IEEE Computer* 21 (5): 61–72. doi:10.1109/2.59.

Hall, E. 1998. *Managing Risk: Methods for Software Systems Development*. Reading, MA: Addison-Wesley.

Marshall, J. E., and R. Heslin. 1975. "Boys and Girls Together. Sexual Composition and the Effect of Density on Group Size and Cohesiveness." *J. of Personality and Social Psychology* 35 (5): 952–961. doi:10.1037/h0076838.

Maslow, A. A. 1954. *Motivation and Personality*. New York: Harper & Row.

Ould, M. 1999. *Managing Software Quality and Business Risk*. Chichester, UK: John Wiley & Sons.



23 Project planning

Objectives

The objective of this chapter is to introduce project planning, scheduling, and cost estimation. When you have read the chapter, you will:

- understand the fundamentals of software costing and the factors that affect the price of a software system to be developed for external clients;
- know what sections should be included in a project plan that is created within a plan-driven development process;
- understand what is involved in project scheduling and the use of bar charts to present a project schedule;
- have been introduced to agile project planning based on the "planning game";
- understand cost estimation techniques and how the COCOMO II model can be used for software cost estimation.

Contents

23.1 Software pricing
23.2 Plan-driven development
23.3 Project scheduling
23.4 Agile planning
23.5 Estimation techniques
23.6 COCOMO cost modeling

Project planning is one of the most important jobs of a software project manager. As a manager, you have to break down the work into parts and assign them to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems. The project plan, which is created at the start of a project and updated as the project progresses, is used to show how the work will be done and to assess progress on the project.

Project planning takes place at three stages in a project life cycle:

1. At the proposal stage, when you are bidding for a contract to develop or provide a software system. You need a plan at this stage to help you decide if you have the resources to complete the work and to work out the price that you should quote to a customer.
2. During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, and so on. Here, you have more information than at the proposal stage, and you can therefore refine the initial effort estimates that you have prepared.
3. Periodically throughout the project, when you update your plan to reflect new information about the software and its development. You learn more about the system being implemented and the capabilities of your development team. As software requirements change, the work breakdown has to be altered and the schedule extended. This information allows you to make more accurate estimates of how long the work will take.

Planning at the proposal stage is inevitably speculative, as you do not have a complete set of requirements for the software to be developed. You have to respond to a call for proposals based on a high-level description of the software functionality that is required. A plan is often a required part of a proposal, so you have to produce a credible plan for carrying out the work. If you win the contract, you then have to re-plan the project, taking into account changes since the proposal was made and new information about the system, the development process, and the development team.

When you are bidding for a contract, you have to work out the price that you will propose to the customer for developing the software. As a starting point for calculating this price, you need to draw up an estimate of your costs for completing the project work. Estimation involves working out how much effort is required to complete each activity and, from this step, calculating the total cost of activities. You should always calculate software costs objectively, with the aim of accurately predicting the cost of developing the software. Once you have a reasonable estimate of the likely costs, you are then in a position to calculate the price that you will quote to the customer. As I discuss in the next section, many factors influence the pricing of a software project—it is not simply cost plus profit.



Overhead costs

When you estimate the costs of effort on a software project, you don't simply multiply the salaries of the people involved by the time spent on the project. You have to take into account all of the organizational overheads (office space, administration, etc.) that must be covered by the income from a project. You calculate the costs by computing these overheads and adding a proportion to the costs of each engineer working on a project.

http://software-engineering-book.com/web/overhead-costs/

You should use three main parameters when computing the costs of a software development project:

- effort costs (the costs of paying software engineers and managers);
- hardware and software costs, including hardware maintenance and software support; and
- travel and training costs.

For most projects, the biggest cost is the effort cost. You have to estimate the total effort (in person-months) that is likely to be required to complete the work of a project. Obviously, you have limited information to make such an estimate. You therefore make the best possible estimate and then add contingency (extra time and effort) in case your initial estimate is optimistic.

For commercial systems, you normally use commodity hardware, which is relatively cheap. However, software costs can be significant if you have to license middleware and platform software. Extensive travel may be needed when a project is developed at different sites. While travel costs themselves are usually a small fraction of the effort costs, the time spent traveling is often wasted and adds significantly to the effort costs of the project. You can use electronic meeting systems and other collaborative software to reduce travel and so have more time available for productive work.

Once a contract to develop a system has been awarded, the outline project plan for the project has to be refined to create a project startup plan. At this stage, you should know more about the requirements for this system. Your aim should be to create a project plan with enough detail to help make decisions about project staffing and budgeting. You use this plan as a basis for allocating resources to the project from within the organization and to help decide if you need to hire new staff.

The plan should also define project monitoring mechanisms. You must keep track of the progress of the project and compare actual and planned progress and costs. Although most companies have formal procedures for monitoring, a good manager should be able to form a clear picture of what is going on through informal discussions with project staff. Informal monitoring can predict potential project problems by revealing difficulties as they occur. For example, daily discussions with project

staff might reveal that the team is having problems with a software fault in the communications systems. The project manager can then immediately assign a communications expert to the problem to help find and solve the problem.

The project plan always evolves during the development process because of requirements changes, technology issues, and development problems. Development planning is intended to ensure that the project plan remains a useful document for staff to understand what is to be achieved and when it is to be delivered. Therefore, the schedule, cost estimate, and risks all have to be revised as the software is developed.

If an agile method is used, there is still a need for a project startup plan because regardless of the approach used, the company still needs to plan how resources will be allocated to a project. However, this is not a detailed plan, and you only need to include essential information about the work breakdown and project schedule. During development, an informal project plan and effort estimates are drawn up for each release of the software, with the whole team involved in the planning process. Some aspects of agile planning have already been covered in Chapter 3, and I discuss other approaches in Section 23.4.

23.1 Software pricing

In principle, the price of a software system developed for a customer is simply the cost of development plus profit for the developer. In practice, however, the relationship between the project cost and the price quoted to the customer is not usually so simple. When calculating a price, you take broader organizational, economic, political, and business considerations into account (Figure 23.1). You need to think about organizational concerns, the risks associated with the project, and the type of contract that will be used. These issues may cause the price to be adjusted upward or downward.

To illustrate some of the project pricing issues, consider the following scenario:

A small software company, PharmaSoft, employs 10 software engineers. It has just finished a large project but only has contracts in place that require five development staff. However, it is bidding for a very large contract with a major pharmaceutical company that requires 30 person-years of effort over two years. The project will not start for at least 12 months but, if granted, it will transform the finances of the company.

PharmaSoft gets an opportunity to bid on a project that requires six people and has to be completed in 10 months. The costs (including overheads of this project) are estimated at $1.2 million. However, to improve its competitive position, PharmaSoft decides to bid a price to the customer of $0.8 million. This means that, although it loses money on this contract, it can retain specialist staff for the more profitable future projects that are likely to come on stream in a year's time.

Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged might then be reduced to reflect the value of the source code to the developer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Financial health	Companies with financial problems may lower their price to gain a contract. It is better to make a smaller-than-normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.

Figure 23.1 Factors affecting software pricing

This is an example of an approach to software pricing called "pricing to win." Pricing to win means that a company has some *idea* of the price that the customer expects to pay and makes a bid for the contract based on the customer's expected price. This may seem unethical and unbusinesslike, but it does have advantages for both the customer and the system provider.

A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the project costs remain within budget.

For example, say a company (OilSoft) is bidding for a contract to develop a fuel delivery system for an oil company that schedules deliveries of fuel to its service stations. There is no detailed requirements document for this system, so OilSoft estimates that a price of $900,000 is likely to be competitive and within the oil company's budget. After being granted the contract, OilSoft then negotiates the detailed requirements of the system so that basic functionality is delivered. It then estimates the additional costs for other requirements.

This approach has advantages for both the software developer and the customer. The requirements are negotiated to avoid requirements that are difficult to implement and potentially very expensive. Flexible requirements make it easier to reuse software. The oil company has awarded the contract to a known company that it can trust. Furthermore, it may be possible to spread the cost of

the project over several versions of the system. This may reduce the costs of system deployment and allow the client to budget for the project cost over several financial years.

23.2 Plan-driven development

Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail. A project plan is created that records the work to be done, who will do it, the development schedule, and the work products. Managers use the plan to support project decision making and as a way of measuring progress. Plan-driven development is based on engineering project management techniques and can be thought of as the "traditional" way of managing large software development projects. Agile development involves a different planning process, discussed in Section 23.4, where decisions are delayed.

The problem with plan-driven development is that early decisions have to be revised because of changes to the environments in which the software is developed and used. Delaying planning decisions avoids unnecessary rework. However, the arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be taken into account. Potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

In my view, the best approach to project planning involves a sensible mixture of plan-based and agile development. The balance depends on the type of project and skills of the people who are available. At one extreme, large security and safety-critical systems require extensive up-front analysis and may have to be certified before they are put into use. These systems should be mostly plan-driven. At the other extreme, small to medium-size information systems, to be used in a rapidly changing competitive environment, should be mostly agile. Where several companies are involved in a development project, a plan-driven approach is normally used to coordinate the work across each development site.

23.2.1 Project plans

In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work. The plan should identify the approach that is taken to risk management as well as risks to the project and the software under development. The details of project plans vary depending on the type of project and organization but plans normally include the following sections:

1. *Introduction* Briefly describes the objectives of the project and sets out the constraints (e.g., budget, time) that affect the management of the project.
2. *Project organization* Describes the way in which the development team is organized, the people involved, and their roles in the team.

Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.

Figure 23.2 Project plan supplements

3. *Risk analysis* Describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies (discussed in Chapter 22) that are proposed.
4. *Hardware and software resource requirements* Specifies the hardware and support software required to carry out the development. If hardware has to be purchased, estimates of the prices and the delivery schedule may be included.
5. *Work breakdown* Sets out the breakdown of the project into activities and identifies the inputs to and the outputs from each project activity.
6. *Project schedule* Shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities. The ways in which the schedule may be presented are discussed in the next section of the chapter.
7. *Monitoring and reporting mechanisms* Defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

The main project plan should always include a project risk assessment and a schedule for the project. In addition, you may develop a number of supplementary plans for activities such as testing and configuration management. Figure 23.2 shows some supplementary plans that may be developed. These are all usually needed in large projects developing large, complex systems.

23.2.2 The planning process

Project planning is an iterative process that starts when you create an initial project plan during the project startup phase. Figure 23.3 is a UML activity diagram that shows a typical workflow for a project planning process. Plan changes are inevitable. As more information about the system and the project team becomes available

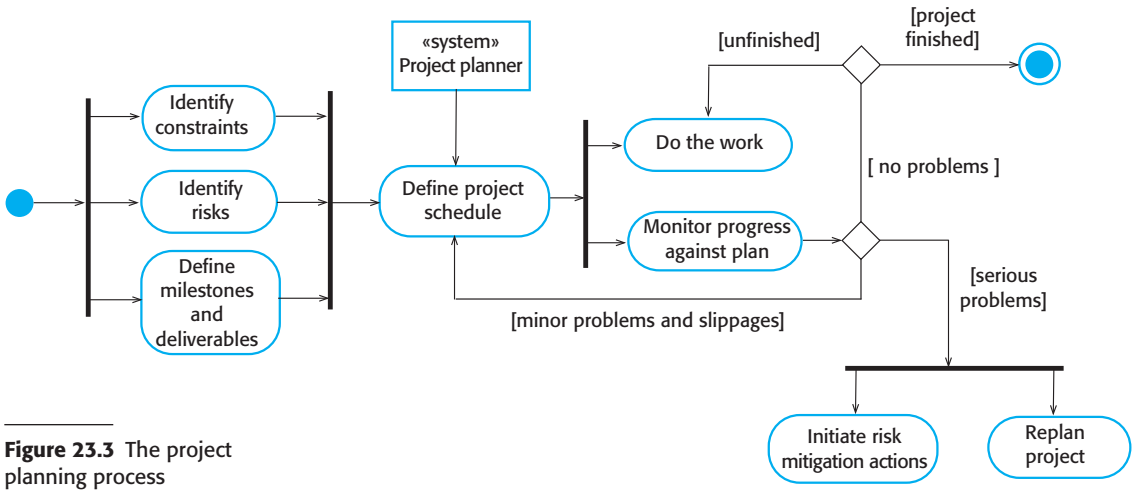


Figure 23.3 The project planning process

during the project, you should regularly revise the plan to reflect requirements, schedule, and risk changes. Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

At the beginning of a planning process, you should assess the constraints affecting the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on. In conjunction with this assessment, you should also identify the project milestones and deliverables. Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing. Deliverables are work products that are delivered to the customer, for example, a requirements document for the system.

The process then enters a loop that terminates when the project is complete. You draw up an estimated schedule for the project, and the activities defined in the schedule are initiated or are approved to continue. After some time (usually about two to three weeks), you should review progress and note discrepancies from the planned schedule. Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and you will have to make modifications to the original plan.

You should make realistic rather than optimistic assumptions when you are defining a project plan. Problems of some description always arise during a project, and these lead to project delays. Your initial assumptions and scheduling should therefore be pessimistic and take unexpected problems into account. You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.

If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of project failure. In conjunction with these actions, you also have to re-plan the project. This may involve renegotiating the project constraints and deliverables with the customer. A new schedule of when work should be completed also has to be established and agreed to with the customer.

If this renegotiation is unsuccessful or the risk mitigation actions are ineffective, then you should arrange for a formal project technical review. The objectives of this review are to find an alternative approach that will allow the project to continue. Reviews should also check that the customer's goals are unchanged and that the project remains aligned with these goals.

The outcome of a review may be a decision to cancel a project. This may be a result of technical or managerial failings but, more often, is a consequence of external changes that affect the project. The development time for a large software project is often several years. During that time, the business objectives and priorities inevitably change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate. Management may then decide to stop software development or to make major changes to the project to reflect the changes in the organizational objectives.

23.3 Project scheduling

Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. You estimate the calendar time needed to complete each task and the effort required, and you suggest who will work on the tasks that have been identified. You also have to estimate the hardware and software resources that are needed to complete each task. For example, if you are developing an embedded system, you have to estimate the time that you need on specialized hardware and the costs of running a system simulator. In terms of the planning stages that I introduced in the introduction of this chapter, an initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development planning.

Both plan-based and agile processes need an initial project schedule, although less detail is included in an agile project plan. This initial schedule is used to plan how people will be allocated to projects and to check the progress of the project against its contractual commitments. In traditional development processes, the complete schedule is initially developed and then modified as the project progresses. In agile processes, there has to be an overall schedule that identifies when the major phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

Scheduling in plan-driven projects (Figure 23.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task. Tasks should normally last at least a week and no longer than 2 months. Finer subdivision means that a disproportionate amount of time must be spent on re-planning and updating the project plan. The maximum amount of time for any task should be 6 to 8 weeks. If a task will take longer than this, it should be split into subtasks for project planning and scheduling.

Some of these tasks are carried out in parallel, with different people working on different components of the system. You have to coordinate these parallel tasks and organize the work so that the workforce is used optimally and you don't introduce

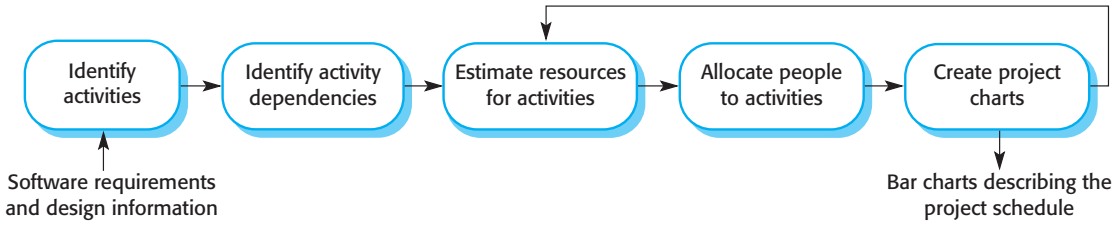


Figure 23.4 The project scheduling process

unnecessary dependencies between the tasks. It is important to avoid a situation where the whole project is delayed because a critical task is unfinished.

If a project is technically advanced, initial estimates will almost certainly be optimistic even when you try to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges, and even new models of cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available. If the project being scheduled is similar to a previous project, previous estimates may be reused. However, projects may use different design methods and implementation languages, so experience from previous projects may not be applicable in the planning of a new project.

When you are estimating schedules, you must take into account the possibility that things will go wrong. People working on a project may fall ill or leave, hardware may fail, and essential support software or hardware may be delivered late. If the project is new and technically advanced, parts of it may turn out to be more difficult and take longer than originally anticipated.

A good rule of thumb is to estimate as if nothing will go wrong and then increase your estimate to cover anticipated problems. A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.), and the quality and experience of the software engineers working on the project. Contingency estimates may add 30 to 50% to the effort and time required for the project.

23.3.1 Schedule presentation

Project schedules may simply be documented in a table or spreadsheet showing the tasks, estimated effort, duration, and task dependencies (Figure 23.5). However, this style of presentation makes it difficult to see the relationships and dependencies between the different activities. For this reason, alternative graphical visualizations of project schedules have been developed that are often easier to read and understand. Two types of visualization are commonly used:

1. Calendar-based bar charts show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are also called Gantt charts, after their inventor, Henry Gantt.

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figure 23.5 Tasks, durations, and dependencies

2. Activity networks show the dependencies between the different activities making up a project. These networks are described in an associated web section.

Project activities are the basic planning element. Each activity has:

- a duration in calendar days or months;
- an effort estimate, which shows the number of person-days or person-months to complete the work;
- a deadline by which the activity should be complete; and
- a defined endpoint, which might be a document, the holding of a review meeting, the successful execution of all tests, or the like.

When planning a project, you may decide to define project milestones. A milestone is a logical end to a stage of the project where the progress of the work can be reviewed. Each milestone should be documented by a brief report (often simply an email) that summarizes the work done and whether or not the work has been completed as planned. Milestones may be associated with a single task or with groups of related activities. For example, in Figure 23.5, milestone M1 is associated with task T1 and marks the end of that activity. Milestone M3 is associated with a pair of tasks T2 and T4; there is no individual milestone at the end of these tasks.



Activity charts

An activity chart is a project schedule representation that presents the project plan as a directed graph. It shows which tasks can be carried out in parallel and those that must be executed in sequence due to their dependencies on earlier activities. If a task is dependent on several other tasks, then all of these tasks must be completed before it can start. The "critical path" through the activity chart is the longest sequence of dependent tasks. This defines the project duration.

http://software-engineering-book.com/web/planning-activities/

Some activities create project deliverables—outputs that are delivered to the software customer. Usually, the deliverables that are required are specified in the project contract, and the customer's view of the project's progress depends on these deliverables. Milestones and deliverables are not the same thing. Milestones are short reports that are used for progress reporting, whereas deliverables are more substantial project outputs such as a requirements document or the initial implementation of a system.

Figure 23.5 shows a hypothetical set of tasks, their estimated effort and duration, and task dependencies. From this table, you can see that task T3 is dependent on task T1. This means that task T1 has to be completed before T3 starts. For example, T1 might be the selection of a system for reuse and T3, the configuration of the selected system. You can't start system configuration until you have chosen and installed the application system to be modified.

Notice that the estimated duration for some tasks is more than the effort required and vice versa. If the effort is less than the duration, the people allocated to that task are not working full time on it. If the effort exceeds the duration, this means that several team members are working on the task at the same time.

Figure 23.6 takes the information in Figure 23.5 and presents the project schedule as a bar chart showing a project calendar and the start and finish dates of tasks. Reading from left to right, the bar chart clearly shows when tasks start and end. The milestones (M1, M2, etc.) are also shown on the bar chart. Notice that tasks that are independent may be carried out in parallel. For example, tasks T1, T2, and T4 all start at the beginning of the project.

As well as planning the delivery schedule for the software, project managers have to allocate resources to tasks. The key resource is, of course, the software engineers who will do the work. They have to be assigned to project activities. The resource allocation can be analyzed by project management tools, and a bar chart can be generated showing when staff are working on the project (Figure 23.7). People may be working on more than one task at the same time, and sometimes they are not working on the project. They may be on holiday, working on other projects, or attending training courses. I show part-time assignments using a diagonal line crossing the bar.

Large organizations usually employ a number of specialists who work on a project when needed. In Figure 23.7, you can see that Mary is a specialist who works on

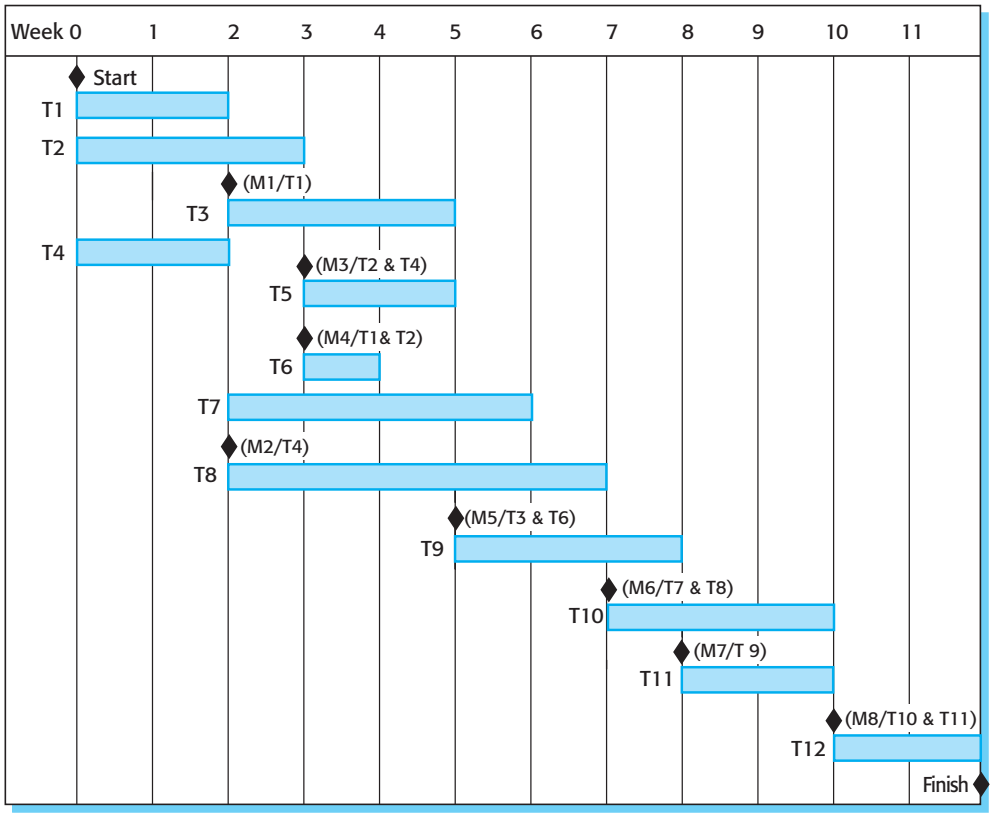


Figure 23.6 Activity bar chart

only a single task (T5) in the project. The use of specialists is unavoidable when complex systems are being developed, but it can lead to scheduling problems. If one project is delayed while a specialist is working on it, this may affect other projects where the specialist is also required. These projects may be delayed because the specialist is not available.

If a task is delayed, later tasks that are dependent on it may be affected. They cannot start until the delayed task is completed. Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time. If a task (T) is delayed, the people allocated to it may be assigned to other work (W). To complete this work may take longer than the delay, but, once assigned, they cannot simply be reassigned back to the original task. This may then lead to further delays in T as they complete W.

Normally, you should use a project planning tool, such as the Basecamp or Microsoft project, to create, update, and analyze project schedule information. Project management tools usually expect you to input project information into a table, and they create a database of project information. Bar charts and activity charts can then be generated automatically from this database.

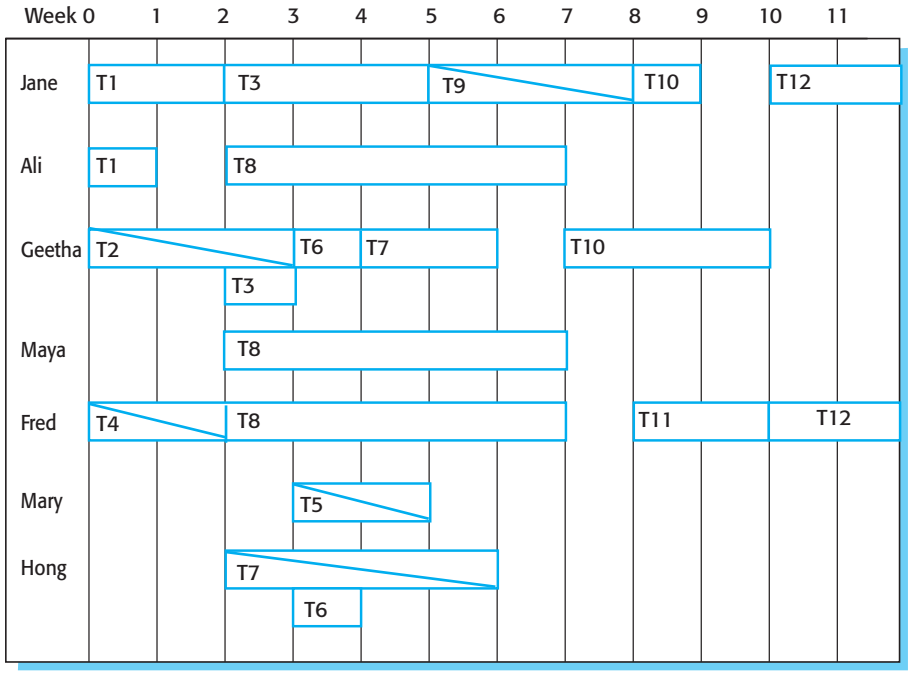


Figure 23.7 Staff allocation chart

23.4 Agile planning

Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments. Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development. The decision on what to include in an increment depends on progress and on the customer's priorities. The argument for this approach is that the customer's priorities and requirements change, so it makes sense to have a flexible plan that can accommodate these changes. Cohn's book (Cohn 2005) is an excellent introduction to agile planning.

Agile development methods such as Scrum (Rubin 2013) and Extreme Programming (Beck and Andres 2004) have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:

1. *Release planning,* which looks ahead for several months and decides on the features that should be included in a release of a system.
2. *Iteration planning,* which has a shorter term outlook and focuses on planning the next increment of a system. This usually represents 2 to 4 weeks of work for the team.

I have already explained the Scrum approach to planning in Chapter 3, which is based on project backlogs and daily reviews of work to be done. It is primarily geared

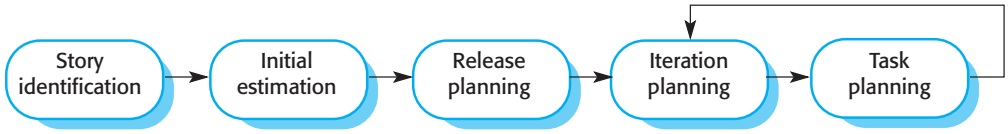


Figure 23.8 The "planning game"

to iteration planning. Another approach to agile planning, which was developed as part of Extreme Programming, is based on user stories. The so-called planning game can be used in both release planning and iteration planning.

The basis of the planning game (Figure 23.8) is a set of user stories (see Chapter 3) that cover all of the functionality to be included in the final system. The development team and the software customer work together to develop these stories. The team members read and discuss the stories and rank them based on the amount of time they think it will take to implement the story. Some stories may be too large to implement in a single iteration, and these are broken down into smaller stories.

The problem with ranking stories is that people often find it difficult to estimate how much effort or time is needed to do something. To make this easier, relative ranking may be used. The team compares stories in pairs and decides which will take the most time and effort, without assessing exactly how much effort will be required. At the end of this process, the list of stories has been ordered, with the stories at the top of the list taking the most effort to implement. The team then allocates notional effort points to all of the stories in the list. A complex story may have 8 points and a simple story 2 points.

Once the stories have been estimated, the relative effort is translated into the first estimate of the total effort required by using the idea of "velocity." Velocity is the number of effort points implemented by the team, per day. This can be estimated either from previous experience or by developing one or two stories to see how much time is required. The velocity estimate is approximate but is refined during the development process. Once you have a velocity estimate, you can calculate the total effort in person-days to implement the system.

Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented. The customer has to be involved in this process. A release date is then chosen, and the stories are examined to see if the effort estimate is consistent with that date. If not, stories are added or removed from the list.

Iteration planning is the first stage in developing a deliverable system increment. Stories to be implemented during that iteration are chosen, with the number of stories reflecting the time to deliver an workable system (usually 2 or 3 weeks) and the team's velocity. When the delivery date is reached, the development iteration is complete, even if all of the stories have not been implemented. The team considers the stories that have been implemented and adds up their effort points. The velocity can then be recalculated, and this measure is used in planning the next version of the system.

At the start of each development iteration, there is a task planning stage where the developers break down stories into development tasks. A development task should take 4–16 hours. All of the tasks that must be completed to implement all of the stories in that iteration are listed. The individual developers then sign up for the specific

tasks that they will implement. Each developer knows their individual velocity and so should not sign up for more tasks than they can implement in the time allotted.

This approach to task allocation has two important benefits:

1. The whole team gets an overview of the tasks to be completed in an iteration. They therefore have an understanding of what other team members are doing and who to talk to if task dependencies are identified.
2. Individual developers choose the tasks to implement; they are not simply allocated tasks by a project manager. They therefore have a sense of ownership in these tasks, and this is likely to motivate them to complete the task.

Halfway through an iteration, progress is reviewed. At this stage, half of the story effort points should have been completed. So, if an iteration involves 24 story points and 36 tasks, 12 story points and 18 tasks should have been completed. If this is not the case, then there has to be discussions with the customer about which stories should be removed from the system increment that is being developed.

This approach to planning has the advantage that a software increment is always delivered at the end of each project iteration. If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced. The delivery schedule is never extended. However, this can cause problems as it means that customer plans may be affected. Reducing the scope may create extra work for customers if they have to use an incomplete system or change the way they work between one release of the system and another.

A major difficulty in agile planning is that it relies on customer involvement and availability. This involvement can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game. Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented. However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management. Consequently, large projects are usually planned using traditional approaches to project management.

23.5 Estimation techniques

Estimating project schedules is difficult. You have to make initial estimates on the basis of an incomplete user requirements definition. The software may have to run on unfamiliar platforms or use new development technology. The people involved in the project and their skills will probably not be known. There are so many uncertainties that it is impossible to estimate system development costs accurately during the early

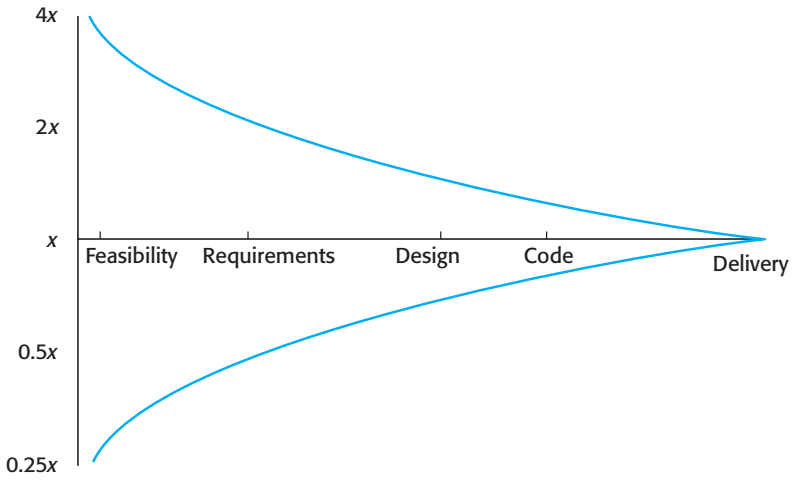


Figure 23.9 Estimate uncertainty

stages of a project. Nevertheless, organizations need to make software effort and cost estimates. Two types of techniques can be used for making estimates:

1. *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
2. *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, process characteristics, and experience of staff involved.

In both cases, you need to use your judgment to estimate either the effort directly or the project and product characteristics. In the startup phase of a project, these estimates have a wide margin of error. Based on data collected from a large number of projects, Boehm et al. (B. Boehm et al. 1995) discovered that startup estimates vary significantly. If the initial estimate of effort required is x months of effort, they found that the range may be from $0.25x$ to $4x$ of the actual effort as measured when the system was delivered. During development planning, estimates become more and more accurate as the project progresses (Figure 23.9).

Experience-based techniques rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development. Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed. You document these in a spreadsheet, estimate them individually, and compute the total effort required. It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate. This often reveals factors that others have not considered, and you then iterate toward an agreed group estimate.

The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects. Software development changes very quickly, and a project will often use unfamiliar techniques such as web services, application system configuration, or HTML5. If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.

It is impossible to say whether experience-based or algorithmic approaches are more accurate. Project estimates are often self-fulfilling. The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized. A project that is within budget may have achieved this at the expense of features in the software being developed.

To make a comparison of the accuracy of these techniques, a number of controlled experiments would be required where several techniques were used independently to estimate the project effort and costs. No changes to the project would be allowed, and the final effort could them be compared. The project manager would not know the effort estimates, so no bias would be introduced. However, this scenario is completely impossible in real projects, so we will never have an objective comparison of these approaches.

23.5.1 Algorithmic cost modeling

Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size, the type of software being developed, and other team, process, and product factors. Algorithmic cost models are developed by analyzing the costs and attributes of completed projects, then finding the closest-fit formula to the actual costs incurred.

Algorithmic cost models are primarily used to make estimates of software development costs. However, Boehm and his collaborators (B. W. Boehm et al. 2000) discuss a range of other uses for these models, such as the preparation of estimates for investors in software companies, alternative strategies to help assess risks and to inform decisions about reuse, redevelopment, or outsourcing.

Most algorithmic models for estimating effort in a software project are based on a simple formula:

$$\text{Effort} = A \times \text{Size}^B \times M$$

A: a constant factor, which depends on local organizational practices and the type of software that is developed.

Size: an assessment of the code size of the software or a functionality estimate expressed in function or application points.

B: represents the complexity of the software and usually lies between 1 and 1.5.

M: is a factor that takes into account process, product and development attributes, such as the dependability requirements for the software and the experience of the development team. These attributes may increase or decrease the overall difficulty of developing the system.

The number of lines of source code (SLOC) in the delivered system is the fundamental size metric that is used in many algorithmic cost models. To estimate the number of lines of code in a system, you may use a combination of approaches:

1. Compare the system to be developed with similar systems and use their code size as the basis for your estimate.
2. Estimate the number of function or application points in the system (see the following section) and formulaically convert these to lines of code in the programming language used.
3. Rank the system components using judgment of their relative sizes and use a known reference component to translate this ranking to code sizes.

Most algorithmic estimation models have an exponential component (B in the above equation) that increases with the size and complexity of the system. This reflects the fact that costs do not usually increase linearly with project size. As the size and complexity of the software increase, extra costs are incurred because of the communication overhead of larger teams, more complex configuration management, more difficult system integration, and so on. The more complex the system, the more these factors affect the cost.

The idea of using a scientific and objective approach to cost estimation is an attractive one, but all algorithmic cost models suffer from two key problems:

1. It is practically impossible to estimate Size accurately at an early stage in a project, when only the specification is available. Function-point and application-point estimates (see later) are easier to produce than estimates of code size but are also usually inaccurate.
2. The estimates of the complexity and process factors contributing to B and M are subjective. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed.

Accurate code size estimation is difficult at an early stage in a project because the size of the final program depends on design decisions that may not have been made when the estimate is required. For example, an application that requires high-performance data management may either implement its own data management system or use a commercial database system. In the initial cost estimation, you are unlikely to know if there is a commercial database system that performs well enough to meet the performance requirements. You therefore don't know how much data management code will be included in the system.

The programming language used for system development also affects the number of lines of code to be developed. A language like Java might mean that more lines of code are necessary than if C (say) was used. However, this extra code allows more compile-time checking, so validation costs are likely to be reduced. It is not clear how this should be taken into account in the estimation process. Code reuse also



Software productivity

Software productivity is an estimate of the average amount of development work that software engineers complete in a week or a month. It is therefore expressed as lines of code/month, function points/month, and so forth.

However, while productivity can be easily measured where there is a tangible outcome (e.g., an administrator processes *N* travel claims/day), software productivity is more difficult to define. Different people may implement the same functionality in different ways, using different numbers of lines of code. The quality of the code is also important but is, to some extent, subjective. Therefore, you can't really compare the productivity of individual engineers. It only makes sense to use productivity measures with large groups.

http://software-engineering-book.com/web/productivity/

makes a difference, and some models explicitly estimate the number of lines of code reused. However, if application systems or external services are reused, it is very difficult to compute the number of lines of source code that these replace.

Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use. There are many attributes and considerable scope for uncertainty in estimating their values. This complexity means that the practical application of algorithmic cost modeling has been limited to a relatively small number of large companies, mostly working in defense and aerospace systems engineering.

Another barrier that discourages the use of algorithmic models is the need for calibration. Model users should calibrate their model and the attribute values using their own historical project data, as this reflects local practice and experience. However, very few organizations have collected enough data from past projects in a form that supports model calibration. Practical use of algorithmic models, therefore, has to start with the published values for the model parameters. It is practically impossible for a modeler to know how closely these relate to his or her organization.

If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected, and best) rather than a single estimate and apply the costing formula to all of them. Estimates are most likely to be accurate when you understand the type of software that is being developed and have calibrated the costing model using local data, or when programming language and hardware choices are predefined.

23.6 COCOMO cost modeling

The best known algorithmic cost modeling technique and tool is the COCOMO II model. This empirical model was derived by collecting data from a large number of software projects of different sizes. These data were analyzed to discover the formulas that were the best fit to the observations. These formulas linked the size of the

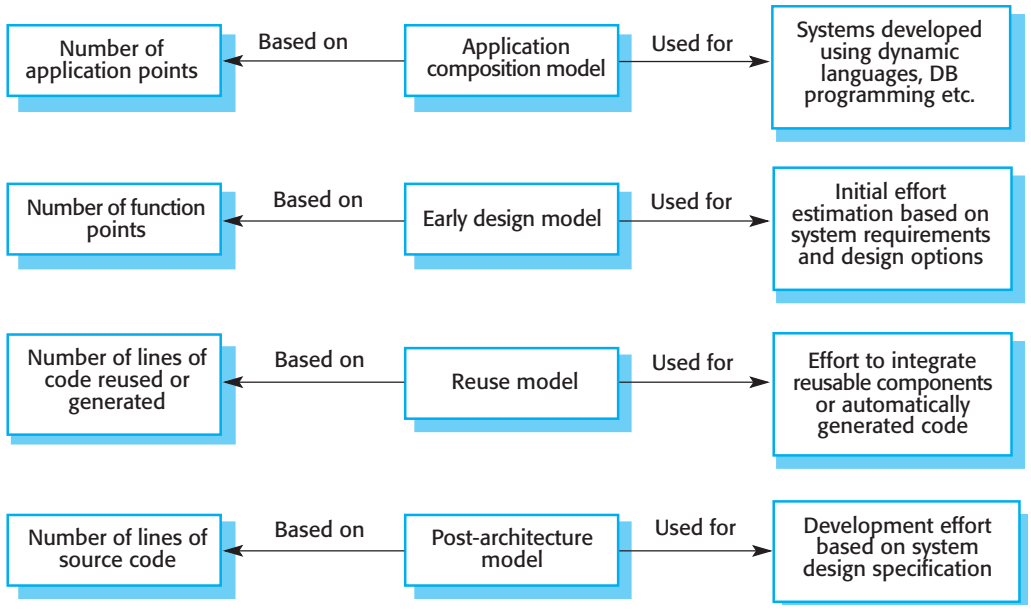


Figure 23.10 COCOMO estimation models

system and product, project, and team factors to the effort to develop the system. COCOMO II is a freely available model that is supported with open-source tools.

COCOMO II was developed from earlier COCOMO (Constructive Cost Modeling) cost estimation models, which were largely based on original code development (B. W. Boehm 1981; B. Boehm and Royce 1989). The COCOMO II model takes into account modern approaches to software development, such as rapid development using dynamic languages, development with reuse, and database programming. COCOMO II embeds several submodels based on these techniques, which produce increasingly detailed estimates.

The submodels (Figure 23.10) that are part of the COCOMO II model are:

1. *An application composition model* This models the effort required to develop systems that are created from reusable components, scripting, or database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.

2. *An early design model* This model is used during early stages of the system design after the requirements have been established. The estimate is based on the standard estimation formula that I discussed in the introduction of this chapter, with a simplified set of seven multipliers. Estimates are based on function points, which are then converted to number of lines of source code.

   Function points are a language-independent way of quantifying program functionality. You compute the total number of function points in a program by measuring or estimating the number of external inputs and outputs, user interactions, external interfaces, and files or database tables used by the system.

3. *A reuse model* This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally used in conjunction with the post-architecture model.
4. *A post-architecture model* Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again, this model uses the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability, product, and project characteristics.

Of course, in large systems, different parts of the system may be developed using different technologies, and you may not have to estimate all parts of the system to the same level of accuracy. In such cases, you can use the appropriate submodel for each part of the system and combine the results to create a composite estimate.

The COCOMO II model is a very complex model and, to make it easier to explain, I have simplified its presentation. You could use the models as I have explained them here for simple cost estimation. However, to use COCOMO properly, you should refer to Boehm's book and the manual for the COCOMO II model (B. W. Boehm et al. 2000; Abts et al. 2000).

23.6.1 The application composition model

The application composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components. It is based on an estimate of weighted application points (sometimes called object points), divided by a standard estimate of application point productivity (B. W. Boehm et al. 2000). The number of application points in a program is derived from four simpler estimates:

- the number of separate screens or web pages that are displayed;
- the number of reports that are produced;
- the number of modules in imperative programming languages (such as Java); and
- the number of lines of scripting language or database programming code.

This estimate is then adjusted according to the difficulty of developing each application point. Productivity depends on the developer's experience and capability as well as the capabilities of the software tools (ICASE) used to support development. Figure 23.11 shows the levels of application-point productivity suggested by the COCOMO model developers.

Application composition usually relies on reusing existing software and configuring application systems. Some of the application points in the system will therefore be implemented using reusable components. Consequently, you have to adjust the

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

Figure 23.11 Application-point productivity

estimate to take into account the percentage of reuse expected. Therefore, the final formula for effort computation for system prototypes is:

$$PM = (NAP \times (1 - \%reuse/100)) / PROD$$

PM: the effort estimate in person-months.

NAP: the total number of application points in the delivered system.

%reuse: an estimate of the amount of reused code in the development.

PROD: the application-point productivity as shown in Figure 23.11.

23.6.2 The early design model

This model may be used during the early stages of a project, before a detailed architectural design for the system is available. The early design model assumes that user requirements have been agreed and initial stages of the system design process are underway. Your goal at this stage should be to make a quick and approximate cost estimate. Therefore, you have to make simplifying assumptions, such as the assumption that there is no effort involved in integrating reusable code.

Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements. The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$Effort = A \times Size^{B} \times M$$

Based on his own large dataset, Boehm proposed that the co-efficient **A** should be 2.94. The size of the system is expressed in KSLOC, which is the number of thousands of lines of source code. You calculate KSLOC by estimating the number of function points in the software. You then use standard tables, which relate software size to function points for different programming languages (QSM 2014) to compute an initial estimate of the system size in KSLOC.

The exponent **B** reflects the increased effort required as the size of the project increases. This can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team, and the process maturity level (see web Chapter 26) of the organization. I discuss how the value of this exponent is calculated using these parameters in the description of the COCOMO II post-architecture model.

This results in an effort computation as follows:

$$PM = 2.94 \times Size^{(1.1 \text{ to } 1.24)} \times M$$

$$M = PERS \times PREX \times RCPX \times RUSE \times PDIF \times SCED \times FSIL$$

PERS: personnel capability

PREX: personnel experience

RCPX: product reliability and complexity

RUSE: reuse required

PDIF: platform difficulty

SCED: schedule

FSIL: support facilities

The multiplier M is based on seven project and process attributes that increase or decrease the estimate. I explain these attributes on the book's web pages. You estimate values for these attributes using a six-point scale, where 1 corresponds to "very low" and 6 corresponds to "very high"; for example, PERS = 6 means that expert staff are available to work on the project.

23.6.3 The reuse model

The COCOMO reuse model is used to estimate the effort required to integrate reusable or generated code. As I have discussed in Chapter 15, software reuse is now the norm in all software development. Most large systems include a significant amount of code that has been reused from previous development projects.

COCOMO II considers two types of reused code. Black-box code is code that can be reused without understanding the code or making changes to it. Examples of black-box code are components that are automatically generated from UML models or application libraries such as graphics libraries. It is assumed that the development effort for black-box code is zero. Its size is not taken into account in the overall effort computation.

White-box code is reusable code that has to be adapted to integrate it with new code or other reused components. Development effort is required for reuse because the code has to be understood and modified before it can work correctly in the system. White-box code could be automatically generated code that needs manual changes or additions. Alternatively, it can be reused components from other systems that have to be modified in the system that is being developed.

Three factors contribute to the effort involved in reusing white-box code components:

1. The effort involved in assessing whether or not a component could be reused in a system that is being developed.
2. The effort required to understand the code that is being reused.
3. The effort required to modify the reused code to adapt it and integrate it with the system being developed.

The development effort in the reuse model is calculated using the COCOMO early design model and is based on the total number of lines of code in the system. The code size includes new code developed for components that are not reused plus an additional factor that allows for the effort involved in reusing and integrating existing code. This additional factor is called ESLOC, the equivalent number of lines of new source code. That is, you express the reuse effort as the effort that would be involved in developing some additional source code.

The formula used to calculate the source code equivalence is:

$$\text{ESLOC} = (\text{ASLOC} \times (1\text{-AT}/100) \times \text{AAM})$$

ESLOC: the equivalent number of lines of new source code.

ASLOC: an estimate of the number of lines of code in the reused components that have to be changed.

AT: the percentage of reused code that can be modified automatically.

AAM: an Adaptation Adjustment Multiplier that reflects the additional effort required to reuse components.

In some cases, the adjustments required to reuse code are syntactic and can be implemented by an automated tool. These do not involve significant effort, so you should estimate what fraction of the changes made to reused code can be automated (AT). This reduces the total number of lines of code that have to be adapted.

The Adaptation Adjustment Multiplier (AAM) adjusts the estimate to reflect the additional effort required to reuse code. The COCOMO model documentation (Abts et al. 2000) discusses in detail how AAM should be calculated. Simplistically, AAM is the sum of three components:

1. An assessment factor (referred to as AA) that represents the effort involved in deciding whether or not to reuse components. AA varies from 0 to 8 depending on the amount of time you need to spend looking for and assessing potential candidates for reuse.
2. An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code that is being reused. SU ranges from 50 for complex, unstructured code to 10 for well-written, object-oriented code.
3. An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. These include design, code, and integration changes.

Once you have calculated a value for ESLOC, you apply the standard estimation formula to calculate the total effort required, where the Size parameter = ESLOC. Therefore, the formula to estimate the reuse effort is:

$$\text{Effort} = A \times \text{ESLOC}^{B} \times M$$

where A, B, and M have the same values as used in the early design model.



COCOMO cost drivers

COCOMO II cost drivers are attributes that reflect some of the product, team, process, and organizational factors that affect the amount of effort needed to develop a software system. For example, if a high level of reliability is required, extra effort will be needed; if there is a need for rapid delivery, extra effort will be required; if the team members change, extra effort will be required.

There are 17 of these attributes in the COCOMO II model, which have been assigned estimated values by the model developers.

http://software-engineering-book.com/web/cost-drivers/

23.6.4 The post-architecture level

The post-architecture model is the most detailed of the COCOMO II models. It is used when you have an initial architectural design for the system. The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

$$PM = A \times Size^{B} \times M$$

By this stage in the process, you should be able to make a more accurate estimate of the project size, as you know how the system will be decomposed into subsystems and components. You make this estimate of the overall code size by adding three code size estimates:

1. An estimate of the total number of lines of new code to be developed (SLOC).
2. An estimate of the reuse costs based on an equivalent number of source lines of code (ESLOC), calculated using the reuse model.
3. An estimate of the number of lines of code that may be changed because of changes to the system requirements.

The final component in the estimate—the number of lines of modified code—reflects the fact that software requirements always change. This leads to rework and development of extra code, which you have to take into account. Of course there will often be even more uncertainty in this figure than in the estimates of new code to be developed.

The exponent term (B) in the effort computation formula is related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. The value of the exponent B is based on five factors, as shown in Figure 23.12. These factors are rated on a six-point scale from 0 to 5, where 0 means "extra high" and 5 means "very low." To calculate B, you add the ratings, divide them by 100, and add the result to 1.01 to get the exponent that should be used.

Scale factor	Explanation
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Team cohesion	Reflects how well the development team knows each other and works together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization as discussed in web chapter 26. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Figure 23.12 Scale factors used in the exponent computation in the post-architecture model

For example, imagine that an organization is taking on a project in a domain in which it has little previous experience. The project client has not defined the process to be used or allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organization has recently put in place a process improvement program and has been rated as a Level 2 organization according to the SEI capability assessment, as discussed in Chapter 26 (web chapter). These characteristics lead to estimates of the ratings used in exponent calculation as follows:

1. *Precedentedness,* rated low (4). This is a new project for the organization.
2. *Development flexibility,* rated very high (1). There is no client involvement in the development process, so there are few externally imposed changes.
3. *Architecture/risk resolution,* rated very low (5). There has been no risk analysis carried out.
4. *Team cohesion,* rated nominal (3). This is a new team, so there is no information available on cohesion.
5. *Process maturity,* rated nominal (3). Some process control is in place.

The sum of these values is 16. You then calculate the final value of the exponent by dividing this sum by 100 and adding 0.01 to the result. The adjusted value of B is therefore 1.17.

The overall effort estimate is refined using an extensive set of 17 product, process, and organizational attributes (see breakout box) rather than the seven attributes used in the early design model. You can estimate values for these attributes because you have more information about the software itself, its non-functional requirements, the development team, and the development process.

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128 KLOC
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Figure 23.13 The effect of cost drivers on effort estimates

Figure 23.13 shows how the cost driver attributes influence effort estimates. Assume that the exponent value is 1.17 as discussed in the above example. Reliability (RELY), complexity (CPLX), storage (STOR), tools (TOOL), and schedule (SCED) are the key cost drivers in the project. All of the other cost drivers have a nominal value of 1, so they do not affect the effort computation.

In Figure 23.13, I have assigned maximum and minimum values to the key cost drivers to show how they influence the effort estimate. The values used are those from the COCOMO II reference manual (Abts et al. 2000). You can see that high values for the cost drivers lead an effort estimate that is more than three times the initial estimate, whereas low values reduce the estimate to about one third of the original. This highlights the significant differences between different types of project and the difficulties of transferring experience from one application domain to another.

23.6.5 Project duration and staffing

As well as estimating the overall costs of a project and the effort that is required to develop a software system, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project. Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor's.

The COCOMO model includes a formula to estimate the calendar time required to complete a project:

$$\text{TDEV} = 3 \times (\text{PM})^{(0.33 + 0.2*(\text{B} - 1.01))}$$

TDEV: the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.

PM: the effort computed by the COCOMO model.

B: a complexity-related exponent, as discussed in section 23.5.2.

If B = 1.17 and PM = 60 then

$$\text{TDEV} = 3 \times (60)^{0.36} = 13 \text{ months}$$

The nominal project schedule predicted by the COCOMO model does not necessarily correspond with the schedule required by the software customer. You may have to deliver the software earlier or (more rarely) later than the date suggested by the nominal schedule. If the schedule is to be compressed (i.e., software is to be developed more quickly), this increases the effort required for the project. This is taken into account by the SCED multiplier in the effort estimation computation.

Assume that a project estimated TDEV as 13 months, as suggested above, but the actual schedule required was 10 months. This represents a schedule compression of approximately 25%. Using the values for the SCED multiplier as derived by Boehm's team, we see that the effort multiplier for this level of schedule compression is 1.43. Therefore, the actual effort that will be required if this accelerated schedule is to be met is almost 50% more than the effort required to deliver the software according to the nominal schedule.

There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project. and the project delivery schedule. If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you could complete the work in 11 months (55 person-months of effort). However, the COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).

The reason for this is that adding people to a project reduces the productivity of existing team members. As the project team increases in size, team members spend more time communicating and defining interfaces between the parts of the system developed by other people. Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.

Consequently, when you add an extra person, the actual increment of effort added is less than one person as others become less productive. If the development team is large, adding more people to a project sometimes increases rather than reduces the development schedule because of the overall effect on productivity.

You cannot simply estimate the number of people required for a project team by dividing the total effort by the required project schedule. Usually, a small number of people are needed at the start of a project to carry out the initial design. The team then

builds up to a peak during the development and testing of the system, and then declines in size as the system is prepared for deployment. A very rapid build-up of project staff has been shown to correlate with project schedule slippage. As a project manager, you should therefore avoid adding too many staff to a project early in its lifetime.

KEY POINTS

- The price charged for a system does not just depend on its estimated development costs and the profit required by the development company. Organizational factors may mean that the price is increased to compensate for increased risk or decreased to gain competitive advantage.
- Software is often priced to gain a contract, and the functionality of the system is then adjusted to meet the estimated price.
- Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule, and who is responsible for each activity.
- Project scheduling involves the creation of various graphical representations of part of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- A project milestone is a predictable outcome of an activity or set of activities. At each milestone, a formal report of progress should be presented to management. A deliverable is a work product that is delivered to the project customer.
- The agile planning game involves the whole team in project planning. The plan is developed incrementally, and, if problems arise, it is adjusted so that software functionality is reduced instead of delaying the delivery of an increment.
- Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- The COCOMO II costing model is a mature algorithmic cost model that takes project, product, hardware, and personnel attributes into account when formulating a cost estimate.

FURTHER READING

Further reading suggested in Chapter 22 is also relevant to this chapter.

"Ten Unmyths of Project Estimation." A pragmatic article that discusses the practical difficulties of project estimation and challenges some fundamental assumptions in this area. (P. Armour, *Comm. ACM*, 45(11), November 2002). http://dx.doi.org/10.1145/581571.581582

Agile Estimating and Planning. This book is a comprehensive description of story-based planning as used in XP, as well as a rationale for using an agile approach to project planning. The book also includes a good, general introduction to project planning issues. (M. Cohn, 2005, Prentice-Hall).

"Achievements and Challenges in COCOMO-based Software Resource Estimation." This article presents a history of the COCOMO models and influences on these models, and discusses the variants of these models that have been developed. It also identifies further possible developments in the COCOMO approach. (B. W. Boehm and R. Valeridi, *IEEE Software*, 25 (5), September/October 2008). http://dx.doi.org/10.1109/MS.2008.133

All About Agile; Agile Planning. This website on agile methods includes an excellent set of articles on agile planning from a number of different authors. (2007–2012). http://www.allaboutagile.com/category/agile-planning/

Project Management Knowhow: Project Planning. This website has a number of useful articles on project management in general. These are aimed at people who don't have previous experience in this area. (P. Stoemmer, 2009–2014). http://www.project-management-knowhow.com/project_planning.html

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

23.1. Describe the factors that affect software pricing. Define the "pricing to win" approach in software pricing.

23.2. Explain why the process of project planning is iterative and why a plan must be continually reviewed during a software project.

23.3. Define project scheduling. What are the things to be considered while estimating schedules?

23.4. What is algorithmic cost modeling? What problems does it suffer from when compared with other approaches to cost estimation?

23.5. Figure 23.14 sets out a number of tasks, their durations, and their dependencies. Draw a bar chart showing the project schedule.

Task	Duration	Dependencies
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

Figure 23.14 Scheduling example

23.6. Figure 23.14 shows the task durations for software project activities. Assume that a serious, unanticipated setback occurs, and instead of taking 10 days, task T5 takes 40 days. Draw up new bar charts showing how the project might be reorganized.

23.7. The planning game is based on the notion of planning to implement the stories that represent the system requirements. Explain the potential problems with this approach when software has high performance or dependability requirements.

23.8. A software manager is in charge of the development of a safety-critical software system, which is designed to control a radiotherapy machine to treat patients suffering from cancer. This system is embedded in the machine and must run on a special-purpose processor with a fixed amount of memory (256 Mbytes). The machine communicates with a patient database system to obtain the details of the patient and, after treatment, automatically records the radiation dose delivered and other treatment details in the database.

The COCOMO method is used to estimate the effort required to develop this system, and an estimate of 26 person-months is computed. All cost driver multipliers were set to 1 when making this estimate.

Explain why this estimate should be adjusted to take project, personnel, product, and organizational factors into account. Suggest four factors that might have significant effects on the initial COCOMO estimate and propose possible values for these factors. Justify why you have included each factor.

23.9. Some very large software projects involve writing millions of lines of code. Explain why the effort estimation models, such as COCOMO, might not work well when applied to very large systems.

23.10. Is it ethical for a company to quote a low price for a software contract knowing that the requirements are ambiguous and that they can charge a high price for subsequent changes requested by the customer?

REFERENCES

Abts, C., B. Clark, S. Devnani-Chulani, and B. W. Boehm. 2000. "COCOMO II Model Definition Manual." Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

Beck, K., and C. Andres. 2004. *Extreme Programming Explained: 2nd ed*. Boston: Addison-Wesley.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. 1995. "Cost Models for Future Software Life Cycle Processes: COCOMO 2." *Annals of Software Engineering*: 1–31. doi:10.1007/BF02249046.

Boehm, B., and W. Royce. 1989. "Ada COCOMO and the Ada Process Model." In *Proc. 5th COCOMO Users' Group Meeting*. Pittsburgh: Software Engineering Institute. http://www.dtic.mil/dtic/tr/fulltext/u2/a243476.pdf

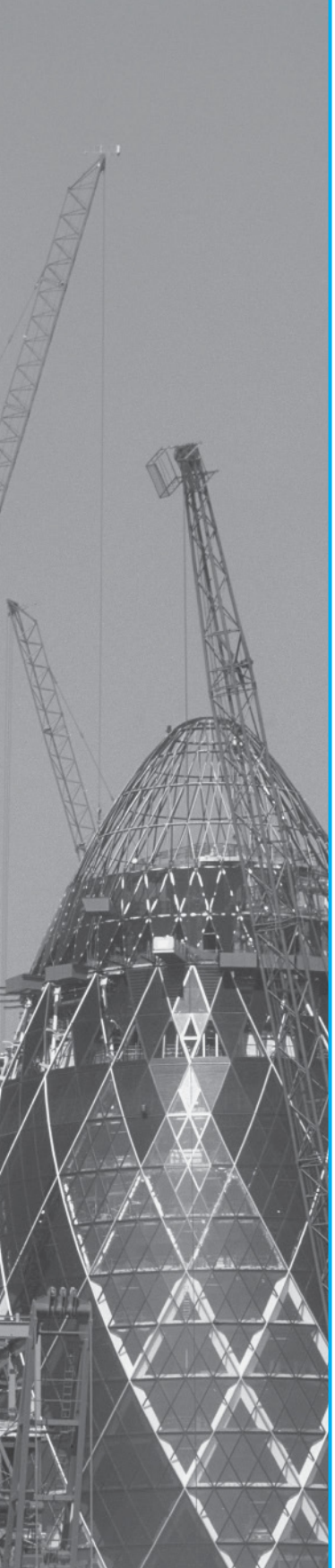
Boehm, B. W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall.

Cohn, M. 2005. *Agile Estimating and Planning*. Englewood-Cliffs, NJ: Prentice Hall.

QSM. 2014. "Function Point Languages Table." http://www.qsm.com/resources/function-point-languages-table

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.



24
Quality management

Objectives

The objectives of this chapter are to introduce software quality management and software measurement. When you have read the chapter, you will:

- have been introduced to the quality management process and know why quality planning is important;
- be aware of the importance of standards in the quality management process and know how standards are used in quality assurance;
- understand how reviews and inspections are used as a mechanism for software quality assurance;
- understand how quality management in agile methods is based on the development of a team quality culture;
- understand how measurement may be helpful in assessing some software quality attributes, the notion of software analytics, and the limitations of software measurement.

Contents

24.1 Software quality
24.2 Software standards
24.3 Reviews and inspections
24.4 Quality management and agile development
24.5 Software measurement

Software quality management is concerned with ensuring that developed software systems are "fit for purpose." That is, systems should meet the needs of their users, should perform efficiently and reliably, and should be delivered on time and within budget. The use of quality management techniques along with new software technologies and testing methods has led to significant improvements in the level of software quality over the past 20 years.

Formalized quality management (QM) is particularly important in teams that are developing large, long-lifetime systems that take several years to develop. These systems are developed for external clients, usually using a plan-based process. For these systems, quality management is both an organizational and an individual project issue:

1. At an organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software. The QM team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.
2. At a project level, quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs meet the defined project standards. Project quality management may also involve defining a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Software quality management techniques have their roots in methods and techniques that were developed in manufacturing industries, where the terms *quality assurance* and *quality control* are widely used. Quality assurance is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process. Quality control is the application of these quality processes to weed out products that are not of the required level of quality. Both quality assurance and quality control are part of quality management.

In the software industry, some companies see quality assurance as the definition of procedures, processes, and standards to ensure that software quality is achieved. In other companies, quality assurance also includes all configuration management, verification, and validation activities that are applied after a product has been handed over by a development team.

Quality management provides an independent check on the software development process. The QM team checks the project deliverables to ensure that they are consistent with organizational standards and goals (Figure 24.1). They also check process documentation, which records the tasks that have been completed by each team working on this project. The QM team uses documentation to check that important tasks have not been forgotten or that one group has not made incorrect assumptions about what other groups have done.

The QM team in large companies is usually responsible for managing the release testing process. As I discussed in Chapter 8, this means that they manage the testing of the software before it is released to customers. In addition, they are responsible

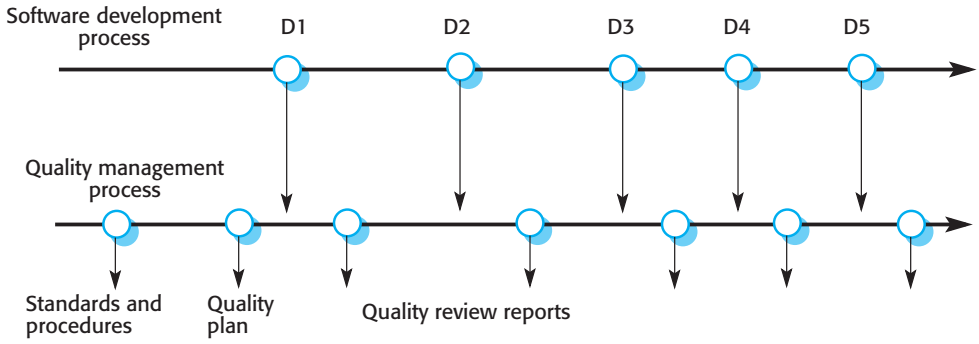


Figure 24.1 Quality management and software development

for checking that the system tests provide coverage of the requirements and that proper records of the testing process are maintained.

The QM team should be independent and not part of the software development group so that they can take an objective view of the quality of the software. They can report on software quality without being influenced by software development issues. Ideally, the QM team should have organization-wide responsibility for quality management. They should report to management above the project manager level.

Because project managers have to maintain the project budget and schedule, they may be tempted to compromise on product quality to meet that schedule. An independent QM team ensures that the organizational goals of quality are not influenced by short-term budget and schedule considerations. In smaller companies, however, this is practically impossible. Quality management and software development are inevitably intertwined with people having both development and quality responsibilities.

Formalized quality planning is an integral part of plan-based development processes. It is the process of developing a quality plan for a project. The quality plan should set out the desired software qualities and describe how these qualities are to be assessed. It defines what "high-quality" software actually means for a particular system. Engineers, therefore, have a shared understanding of the most important software quality attributes.

Humphrey (Humphrey 1989), in his classic book on software management, suggests an outline structure for a quality plan. This outline includes the following:

1. *Product introduction* A description of the product, its intended market, and the quality expectations for the product.
2. *Product plans* The critical release dates and responsibilities for the product, along with plans for distribution and product servicing.
3. *Process descriptions* The development and service processes and standards that should be used for product development and management.
4. *Quality goals* The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. *Risks and risk management* The key risks that might affect product quality and the actions to be taken to address these risks.

Quality plans, which are developed as part of the general project planning process, differ in detail depending on the size and type of system being developed. However, when writing quality plans, you should try to keep them as short as possible. If the document is too long, people will not read it, so defeating the purpose of producing the quality plan.

Traditional quality management is a formal process that relies on maintaining extensive documentation about testing and system validation and on how processes have been followed. In this respect, it is diametrically opposed to agile development, where the aim is to spend as little time as possible in writing documents and formalizing how the development work should be done. QM techniques have therefore had to evolve when agile methods are used. I discuss QM and agile development in Section 24.4.

24.1 Software quality

The manufacturing industry established the fundamentals of quality management in a drive to improve the quality of the products that were being made. As part of this effort, the industry developed a definition of quality that was based on conformance with a detailed product specification. The underlying assumption was that products could be completely specified and procedures could be established that could check a manufactured product against its specification. Of course, products will never exactly meet a specification, so some tolerance was allowed. If the product was "almost right," it was classed as acceptable.

Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is applicable in analog systems but does not apply to software. Furthermore, it is often impossible to come to an objective conclusion about whether or not a software system meets its specification:

1. It is difficult to write complete and unambiguous software requirements. Software developers and customers may interpret the requirements in different ways, and it may be impossible to reach agreement on whether or not software conforms to its specification.
2. Specifications usually integrate requirements from several classes of stakeholder. These requirements are inevitably a compromise and may not include the requirements of all stakeholder groups. The excluded stakeholders may therefore perceive the system as a poor-quality system, even though it implements the agreed requirements.
3. It is impossible to measure certain quality characteristics (e.g., maintainability) directly, and so they cannot be specified in an unambiguous way. I discuss the difficulties of measurement in Section 24.4.

Because of these problems, the assessment of software quality is a subjective process. The quality management team uses their judgment to decide if an acceptable level of quality has been achieved. They decide whether or not the software is fit for

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Figure 24.2 Software quality attributes

its intended purpose. This decision involves answering questions about the system's characteristics. For example:

1. Has the software been properly tested, and has it been shown that all requirements have been implemented?
2. Is the software sufficiently dependable to be put into use?
3. Is the performance of the software acceptable for normal use?
4. Is the software usable?
5. Is the software well structured and understandable?
6. Have programming and documentation standards been followed in the development process?

There is a general assumption in software quality management that the system will be tested against its requirements. The judgment on whether or not it delivers the required functionality should be based on the results of these tests. Therefore, the QM team should review the tests that have been developed and examine the test records to check that testing has been properly carried out. In some companies, the QM group carries out final system testing; in others, a dedicated system testing team reports to the system quality manager.

The subjective quality of a software system is largely based on its non-functional characteristics. This reflects practical user experience—if the software's functionality is not what is expected, then users will often just work around this deficiency and find other ways to do what they want to do. However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Therefore, software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes as shown in Figure 24.2. These attributes reflect the software dependability, usability, efficiency, and maintainability.

It is not possible for any system to be optimized for all of these attributes. For example, improving security may lead to loss of performance. The quality plan should therefore define the most important quality attributes for the software that is being developed. It may be that efficiency is critical and other factors have to be sacrificed to achieve it. If you have emphasized the importance of efficienty in the quality plan, the engineers working on the development can work together to achieve this. The plan should also include a definition of the quality assessment process.

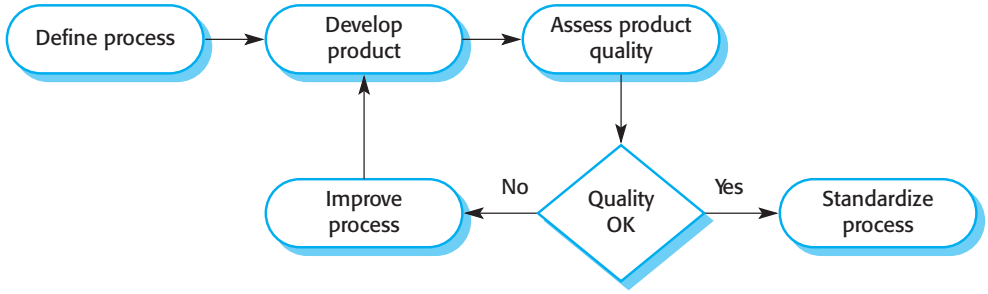


Figure 24.3 Process-based quality

This process should be an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Traditional software quality management is based on the assumption that the quality of software is directly related to the quality of the software development process. This assumption comes from manufacturing systems where product quality is intimately related to the production process. A manufacturing process involves configuring, setting up, and operating the machines involved in the process. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need. Figure 24.3 illustrates this process-based approach to achieving product quality.

There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. However, software is designed rather than manufactured, and the relationship between process quality and product quality is more complex. Software design is a creative process, so the influence of individual skills and experience is significant. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

Without doubt, the development process used has a significant influence on the quality of the software, and good processes are more likely to lead to good quality software. Process quality management and improvement can result in fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as reliability and maintainability, without using the software for a long period. Consequently, it is hard to tell how process characteristics influence these attributes. Furthermore, because of the role of design and creativity in the software process, process standardization can sometimes stifle creativity, which may lead to poorer rather than better quality software.

Defined processes are important, but quality managers should also aim to develop a "quality culture" in which everyone responsible for software development is committed to achieving a high level of product quality. They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement. While standards and procedures are the basis of quality management, good-quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.



Documentation standards

Project documents are a tangible way of describing the different representations of a software system (requirements, UML, code, etc.) and its production process. Documentation standards define the organization of different types of document as well as the document format. They are important because they make it easier to check that important material has not been omitted from documents and ensure that project documents have a common "look and feel." Standards may be developed for the process of writing documents, for the documents themselves and for document exchange.

http://software-engineering-book.com/web/documentation-standards/

24.2 Software standards

Software standards play an important role in plan-based software quality management. As I have discussed, an important part of quality assurance is the definition or selection of standards that should apply to the software development process or software product. As part of this process, tools and methods to support the use of these standards may also be chosen. Once standards have been selected for use, project-specific processes have to be defined to monitor the use of the standards and check that they have been followed.

Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often acquired only after a great deal of trial and error. Building it into a standard helps the company reuse this experience and avoid previous mistakes.
2. Standards provide a framework for defining what quality means in a particular setting. As I have discussed, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved. Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.
3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

Two related types of software engineering standard may be defined and used in software quality management:

1. *Product standards* These apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Figure 24.4 Product and process standards

2. *Process standards* These define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design, and validation processes, process support tools, and a description of the documents that should be written during these processes.

Examples of product and process standards that may be used are shown in Figure 24.4.

Standards have to deliver value, in the form of increased product quality. There is no point in defining standards that are expensive in terms of time and effort to apply that only lead to marginal improvements in quality. Product standards have to be designed so that they can be applied and checked in a cost-effective way, and process standards should include the definition of processes that check if product standards have been followed.

The software engineering standards that are used within a company are usually adapted from broader national or international standards. National and international standards have been developed covering software engineering terminology, programming languages such as Java and C++, notations such as charting symbols, procedures for deriving and writing software requirements, quality assurance procedures, and software verification and validation processes (IEEE 2003). More specialized standards have been developed for safety and security critical systems.

Software engineers sometimes consider standards to be overprescriptive and irrelevant to the technical activity of software development. This is particularly likely when project standards require tedious documentation and work recording. Although they usually agree about the general need for standards, engineers often find good reasons why standards are not necessarily appropriate to their particular project. Quality managers who set the standards should therefore consider possible actions to convince engineers of the value of standards:

1. *Involve software engineers in the selection of product standards* If developers understand why standards have been selected, they are more likely to be committed to these standards. Ideally, the standards document should not just set out the standard to be followed but should also include commentary explaining why standardization decisions have been made.

2. *Review and modify standards regularly to reflect changing technologies* Standards are expensive to develop, and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential, but it should evolve to reflect changing circumstances and technology.

3. *Make sure that tool support is available to support standards-based development* Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, standards can be followed without much extra effort. For example, program layout standards can be defined and implemented by a syntax-directed program editing system.

Different types of software need different development processes, so standards have to be adaptable. There is no point in prescribing a particular way of working if it is inappropriate for a project or project team. Each project manager should have the authority to modify process standards according to individual circumstances. However, when changes are made, it is important to ensure that these changes do not lead to a loss of product quality.

The project manager and the quality manager can avoid the problems of inappropriate standards by careful quality planning early in the project. They should decide which of the organizational standards should be used without change, which should be modified, and which should be ignored. New standards may have to be created in response to customer or project requirements. For example, standards for formal specifications may be required if these standards have not been used in previous projects.

24.2.1 The ISO 9001 standards framework

The international set of standards used in the development of quality management systems in all industries is called ISO 9000. ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries. ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software. The ISO 9001 standard was originally developed in 1987. I explain the 2008 version of the standard here, but the standard may change in 2015 when a new version is scheduled for release.

The ISO 9001 standard is not a standard for software development but rather is a framework for developing software standards. It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

A major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes (Figure 24.5). If an organization is to be ISO 9001 conformant, it must document how its processes relate to these core processes. It must also define and maintain records demonstrating that the defined organizational processes have

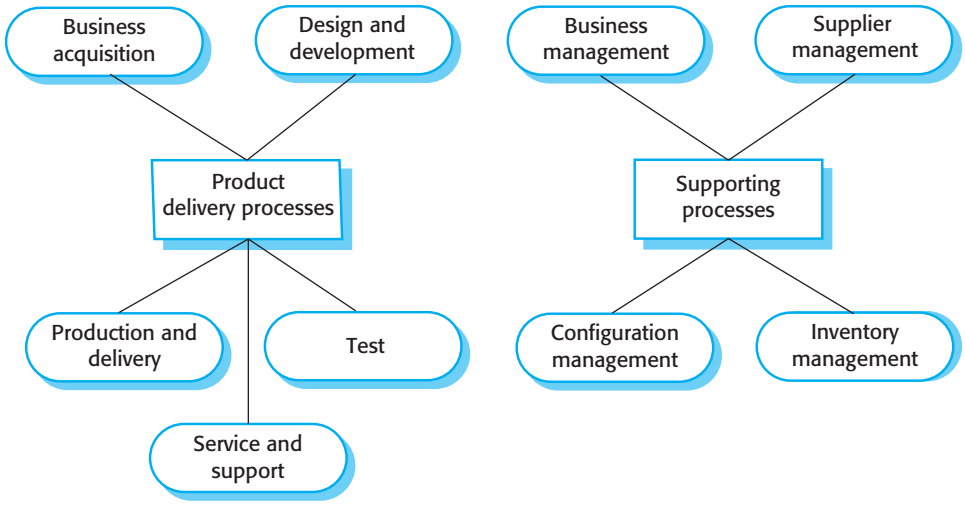


Figure 24.5 ISO 9001 core processes

been followed. The company quality manual should describe the relevant processes and the process data that has to be collected and maintained.

The ISO 9001 standard does not define or prescribe the specific quality processes that a company should use. To be conformant with ISO 9001, a company must define the types of process shown in Figure 24.5 and have procedures in place demonstrating that its quality processes are being followed. This allows flexibility across industrial sectors and company sizes.

Quality standards can be defined that are appropriate for the type of software being developed. Small companies can have simple processes without much documentation and still be ISO 9001 compliant. However, this flexibility means that you cannot make assumptions about the similarities or differences between the processes in different ISO 9001–compliant companies. Some companies may have very rigid quality processes that keep detailed records while others may be much less formal, with minimal additional documentation.

The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 24.6. This diagram has been adapted from a model given by Ince (Ince 1994), who explains how the general ISO 9001 standard can be used as a basis for software quality management processes. Bamford and Deibler (Bamford and Deibler 2003) explain how the later ISO 9001: 2000 standard can be applied in software companies.

Some software customers demand that their suppliers be ISO 9001 certified. The customers can then be confident that the software development company has an approved quality management system in place. Independent accreditation authorities examine the quality management processes and process documentation and decide if these processes cover all of the areas specified in ISO 9001. If so, they certify that a company's quality processes, as defined in the quality manual, conform to the ISO 9001 standard.

Some people mistakenly think that ISO 9001 certification means that the quality of the software produced by certified companies will always be better than that from

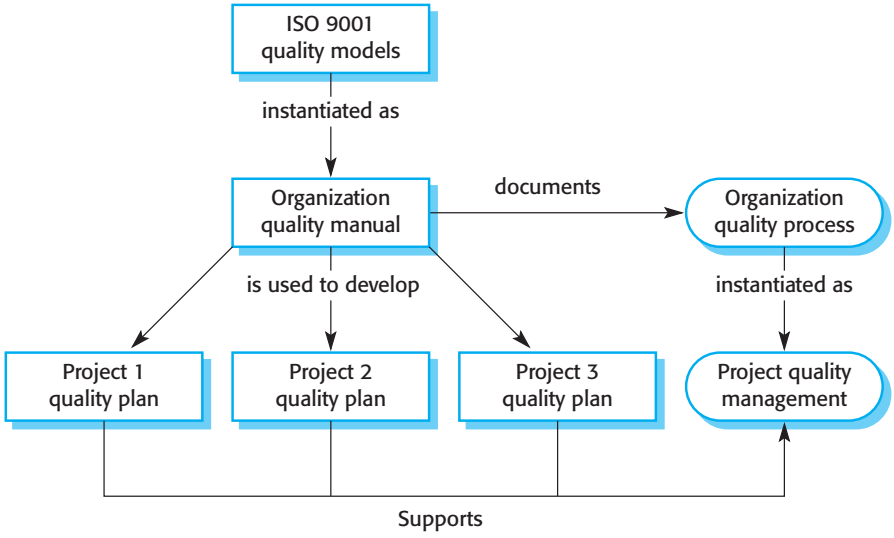


Figure 24.6 ISO 9001 and quality management

uncertified companies. The ISO 9001 standard focuses on ensuring that the organization has quality management procedures in place and that it follows these procedures. There is no guarantee that ISO 9001 certified companies use the best software development practices or that their processes lead to high-quality software.

The ISO 9001 certification is inadequate, in my view, because it defines quality to be the conformance to standards. It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once. Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. As long as the defined testing procedures are followed and test records are maintained, the company could be ISO 9001 certified.

24.3 Reviews and inspections

Reviews and inspections are quality assurance activities that check the quality of project deliverables. This involves checking the software, its documentation, and records of the process to discover errors and omissions as well as standards violations. As I explained in Chapter 8, reviews and inspections are used alongside program testing as part of the general process of software verification and validation.

During a review, several people examine the software and its associated documentation, looking for potential problems and nonconformance with standards. The review team makes informed judgments about the level of quality of the software or project documents. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

Quality reviews are based on documents that have been produced during the software development process. As well as software specifications, designs, code, process models, test plans, configuration management procedures, process standards,

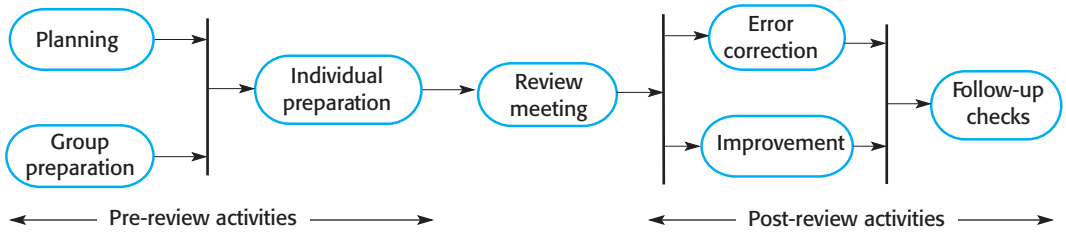


Figure 24.7 The software review process

and user manuals may all be reviewed. The review should check the consistency and completeness of the documents or code under review and, if standards have been defined, make sure that these quality standards have been followed.

Reviews are not just about checking conformance to standards. They are also used to help discover problems and omissions in the software or project documentation. The conclusions of the review should be formally recorded as part of the quality management process. If problems have been discovered, the reviewers' comments should be passed to the author of the software or whoever is responsible for correcting errors or omissions.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team. Reviewing is a public process of error detection, compared with the more private component-testing process. Inevitably, mistakes that are made by individuals are revealed to the whole programming team. To ensure that all developers engage constructively with the review process, project managers have to be sensitive to individual concerns. They must develop a working culture that provides support without blame when errors are discovered.

Quality reviews are not management progress reviews, although information about the software quality may be used in making management decisions. Progress reviews compare the actual progress in a software project against the planned progress. Their prime concern is whether or not the project will deliver useful software on time and on budget. Progress reviews take external factors into account, and changed circumstances may mean that software under development is no longer required or has to be radically changed. Projects that have developed high-quality software may have to be canceled because of changes to the business or its operating environment.

24.3.1 The review process

Although there are many variations in the details of reviews, review processes (Figure 24.7) are structured into three phases:

1. *Pre-review activities* These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards.



Roles in the inspection process

When program inspection was established at IBM (Fagan, 1986), a number of formal roles were defined for members of the inspection team. These included moderator, code reader, and scribe. Other users of inspections have modified these roles, but it is generally accepted that an inspection should involve the code author, an inspector, and a scribe and should be chaired by a moderator.

http://software-engineering-book.com/web/qm-roles

They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. *The review meeting* During the review meeting, an author of the document or program being reviewed should "walk through" the document with the review team. The review itself should be relatively short—two hours at most. One team member should chair the review, and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all submitted comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. *Post-review activities* After a review meeting has ended, the issues and problems raised during the review must be addressed. Actions may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents. Sometimes the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that all the review comments have been taken into account. Sometimes a further review will be required to check that the changes made cover all of the previous review comments.

Review teams should normally have a core of three to four people who are selected as principal reviewers. One member should be an experienced designer who will take the responsibility for making significant technical decisions. The principal reviewers may invite other project members, such as the designers of related subsystems, to contribute to the review. They may not be involved in reviewing the whole document but should concentrate on those sections that affect their work. Alternatively, the review team may circulate the document and ask for written comments from a broad spectrum of project members. The project manager need not be involved in the review, unless problems are anticipated that require changes to the project plan.

The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents they are reviewing. However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face. Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments. Face-to-face meetings may be impossible

because of work schedules or the fact that people work in different time zones. The review chair is responsible for coordinating comments and for discussing changes individually with the review team members.

24.3.2 Program inspections

Program inspections are peer reviews where team members collaborate to find bugs in the program that is being developed. As I discussed in Chapter 8, inspections may be part of the software verification and validation processes. They complement testing as they do not require the program to be executed. Incomplete versions of the system can be verified, and representations such as UML models can be checked. Program tests may be reviewed. Test reviews often find problems with tests and so improve their effectiveness in detecting program bugs.

Program inspections involve team members from different backgrounds who make a careful, line-by-line review of the program source code. They look for defects and problems and describe them at an inspection meeting. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or features that have been omitted from the code. The review team examines the design models or the program code in detail and highlights anomalies and problems for repair.

During an inspection, a checklist of common programming errors is often used to focus the search for bugs. This checklist may be based on examples from books or from knowledge of defects that are common in a particular application domain. You use different checklists for different programming languages because each language has its own characteristic errors. Humphrey (Humphrey, 1989), in a comprehensive discussion of inspections, gives a number of examples of inspection checklists.

Possible checks that might be made during the inspection process are shown in Figure 24.8. Organizations should develop their own inspection checklists based on local standards and practices. These checklists should be regularly updated, as new types of defects are found. The items in the checklist vary according to programming language because of the different levels of checking that are possible at compile-time. For example, a Java compiler checks that functions have the correct number of parameters; a C compiler does not.

Companies that use inspections have found that they are effective in finding bugs. In early work, Fagan (Fagan 1986) reported that more than 60% of the errors in a program were detected using informal program inspections. McConnell (McConnell 2004) compares unit testing, where the defect detection rate is about 25%, with inspections, where the defect detection rate was 60%. These comparisons were made before widespread automated testing. We don't know how inspections compare to this approach.

In spite of their well-publicized cost-effectiveness, many software development companies are reluctant to use inspections or peer reviews. Software engineers with experience in program testing are sometimes unwilling to accept the fact that inspections can be more effective for defect detection than testing. Managers may be suspicious because inspections require additional costs during design and development. They may not want to take the risk that there will be no corresponding savings in program testing costs.

Fault class	Inspection check
Data faults	■ Are all program variables initialized before their values are used? ■ Have all constants been named? ■ Should the upper bound of arrays be equal to the size of the array or Size −1? ■ If character strings are used, is a delimiter explicitly assigned? ■ Is there any possibility of buffer overflow?
Control faults	■ For each conditional statement, is the condition correct? ■ Is each loop certain to terminate? ■ Are compound statements correctly bracketed? ■ In case statements, are all possible cases accounted for? ■ If a break is required after each case in case statements, has it been included?
Input/output faults	■ Are all input variables used? ■ Are all output variables assigned a value before they are output? ■ Can unexpected inputs cause corruption?
Interface faults	■ Do all function and method calls have the correct number of parameters? ■ Do formal and actual parameter types match? ■ Are the parameters in the right order? ■ If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	■ If a linked structure is modified, have all links been correctly reassigned? ■ If dynamic storage is used, has space been allocated correctly? ■ Is space explicitly de-allocated after it is no longer required?
Exception management faults	■ Have all possible error conditions been taken into account?

Figure 24.8 An inspection checklist

24.4 Quality management and agile development

Agile methods of software engineering focus on the development of code. They minimize documentation and processes that are not directly concerned with code development and emphasize the importance of informal communications among team members rather than communications based on project documents. Quality, in agile development, means code quality and practices such as refactoring, and test-driven development are used to ensure that high-quality code is produced.

Quality management in agile development is informal rather than document-based. It relies on establishing a quality culture, where all team members feel responsible for software quality and take actions to ensure that quality is maintained. The agile community is fundamentally opposed to what it sees as the bureaucratic overhead of standards-based approaches and quality processes as embodied in ISO 9001. Companies that use agile development methods are rarely concerned with ISO 9001 certification.

In agile development, quality management is based on shared good practice rather than formal documentation. Some examples of this good practice are:

1. *Check before check-in* Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.

2. *Never break the build* It is not acceptable for team members to check in code that causes the system as a whole to fail. Therefore, individuals have to test their code changes against the whole system and be confident that these codes work as expected. If the build is broken, the person responsible is expected to give top priority to fixing the problem.

3. *Fix problems when you see them* The code of the system belongs to the team rather than to individuals. Therefore, if a programmer discovers problems or obscurities in code developed by someone else, he or she can fix these problems directly rather than referring them back to the original developer.

Agile processes rarely use formal inspection or review processes. In Scrum, the development team meets after each iteration to discuss quality issues and problems. The team may decide on changes to the way they work to avoid any quality problems that have emerged. A collective decision may be made to focus on refactoring and quality improvement during a sprint rather than the addition of new system functionality.

Code reviews may be the responsibility of individuals (check before check-in) or may rely on the use of pair programming. As I discussed in Chapter 3, pair programming is an approach in which two people are responsible for code development and work together to achieve it. Code developed by an individual is therefore constantly being examined and reviewed by another team member. Two people look at every line of code and check it before it is accepted.

Pair programming leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development. This depth of knowledge is sometimes difficult to achieve in other inspection processes, and so pair programming can find bugs that sometimes would not be discovered in formal inspections. However, the two people involved cannot be as objective as an external inspection team inasmuch as they are examining their own work. Potential problems are:

1. *Mutual misunderstandings* Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.

2. *Pair reputation* Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.

3. *Working relationships* The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

The informal approach to quality management adopted in agile methods is particularly effective for software product development where the company developing the software also controls its specification. There is no need to deliver quality reports to an external customer, nor is there any need to integrate with other quality management teams. However, when a large system is being developed for an

external customer, agile approaches to quality management with minimal documentation may be impractical:

1. If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with these processes. Therefore, the development team may have to produce a formal quality plan and quality reports as required by the customer.
2. Where several geographically distributed teams are involved in development, perhaps from different companies, then informal communications may be impractical. Different companies may have different approaches to quality management, and you may have to agree to produce some formal documentation.
3. For long-lifetime systems, the team involved in development will change over time. If there is no documentation, new team members may find it impossible to understand why development decisions have been made.

Consequently, the informal approach to quality management in agile methods may have to be adapted so that some quality documentation and processes are introduced. Generally, this approach is integrated with the iterative development process. Instead of developing software, one of the sprints or iterations should focus on producing essential software documentation.

24.5 Software measurement

Software measurement is concerned with quantifying some attribute of a software system such as its complexity or its reliability. By comparing the measured values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software or assess the effectiveness of software processes, tools, and methods. In an ideal world, quality management could rely on measurements of attributes that affect the software quality. You could then objectively assess process and tool changes that aim to improve software quality.

For example, say you work in a company that plans to introduce a new software-testing tool. Before introducing the tool, you record the number of software defects discovered in a given time. This is a baseline for assessing the effectiveness of the tool. After using the tool for some time, you repeat this process. If more defects have been found in the same amount of time, after the tool has been introduced, then you may decide that it provides useful support for the software validation process.

The long-term goal of software measurement is to use measurement to make judgments about software quality. Ideally, a system could be assessed using a range of metrics to measure its attributes. From the measurements made, a value for the quality of the system could be inferred. If the software had reached a required quality threshold, then it could be approved without review. When appropriate, the measurement tools might also highlight areas of the software that could be improved.

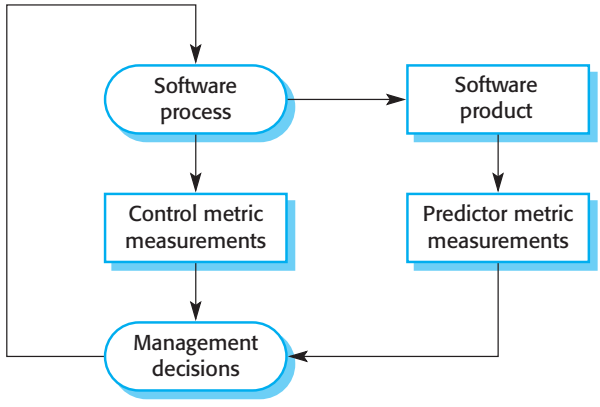


Figure 24.9 Predictor and control measurements

However, we are still a long way from this ideal situation, and automated quality assessment is unlikely to become a reality in the near future.

A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured. Examples of metrics include the size of a product in lines of code, the Fog index, which is a measure of the readability of narrative text, the number of reported faults in a delivered software product, and the number of person-days required to develop a system component.

Software metrics may be either control metrics or predictor metrics. As the names imply, control metrics support process management, and predictor metrics help you predict characteristics of the software. Control metrics are usually associated with software processes. Examples of control or process metrics are the average effort and the time required to repair reported defects. Three kinds of process metrics can be used:

1. *The time taken for a particular process to be completed* This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
2. *The resources required for a particular process* Resources might include total effort in person-days, travel costs, or computer resources.
3. *The number of occurrences of a particular event* Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system, and the average number of lines of code modified in response to a requirements change.

Predictor metrics (sometimes called product metrics) are associated with the software itself. Examples of predictor metrics are the cyclomatic complexity of a module, the average length of identifiers in a program, and the number of attributes and operations associated with object classes in a design. Both control and predictor metrics may influence management decision making as shown in Figure 24.9. Managers use process measurements to decide if process changes should be made and predictor metrics to decide if software changes are necessary and if the software is ready for release.

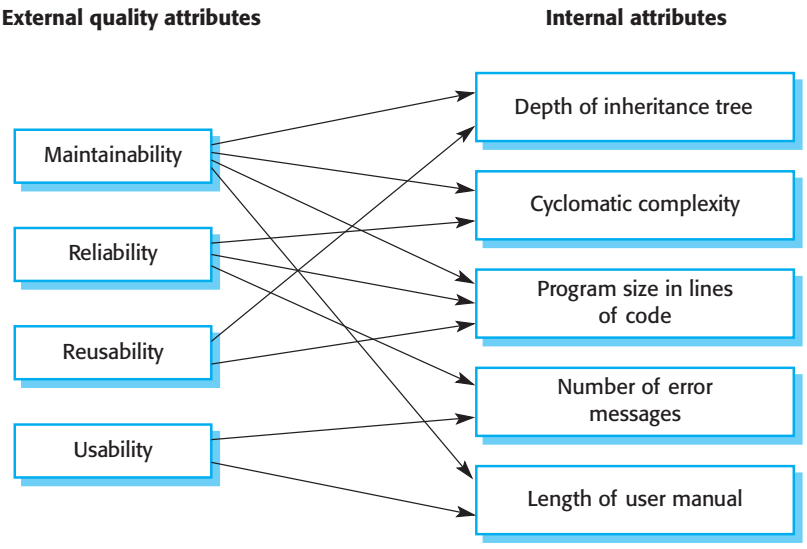


Figure 24.10 Relationships between internal and external software attributes

In this chapter, I focus on predictor metrics, whose values are automatically assessed by analyzing code or documents. I discuss control metrics and how they are used in process improvement in web Chapter 26.

Measurements of a software system may be used in two ways:

1. *To assign a value to system quality attributes* By measuring the characteristics of system components and then aggregating these measurements, you may be able to assess system quality attributes, such as maintainability.
2. *To identify the system components whose quality is substandard* Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These components are most likely to contain bugs because the complexity makes it more likely that the component developer has made mistakes.

It is difficult to make direct measurements of many of the software quality attributes shown in Figure 24.2. Quality attributes such as maintainability, understandability, and usability are external attributes that relate to how developers and users experience the software. They are affected by subjective factors, such as user experience and education, and they cannot therefore be measured objectively. To make a judgment about these attributes, you have to measure some internal attributes of the software (such as its size and complexity) and assume that these are related to the quality characteristics that you are concerned with.

Figure 24.10 shows some external software quality attributes and internal attributes that could, intuitively, be related to them. The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related. Kitchenham (Kitchenham 1990) suggested that if the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold:

1. The internal attribute must be measured accurately. However, measurement is not always straightforward and may require specially developed tools.
2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.
3. This relationship between the internal and external attributes must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.) by analysis of collected data, identifying the parameters that are to be included in the model and calibrating these parameters using existing data.

Recent work in the area of software analytics (Zhang et al. 2013) has used data-mining and machine-learning techniques to analyze repositories of software product and process data. The idea behind software analytics (Menzies and Zimmermann 2013) is that we do not, in fact, need a model that reflects the relationships between software quality and collected data. Rather, if there is enough data, correlations can be discovered and predictions made about software attributes. I discuss software analytics in Section 24.5.4.

We have very little published information about systematic software measurement in industry. Many companies do collect information about their software, such as the number of requirements change requests or the number of defects discovered in testing. However, it is not clear if they then use these measurements systematically to compare software products and processes or assess the impact of changes to software processes and tools. There are several reasons why this is difficult:

1. It is impossible to quantify the return on investment of introducing an organizational metrics or software analytics program. We have seen significant improvements in software quality over the past few years without the use of metrics, so it is difficult to justify the initial costs of introducing systematic software measurement and assessment.
2. There are no standards for software metrics or standardized processes for measurement and analysis. Many companies are reluctant to introduce measurement programs until such standards and supporting tools are available.
3. Measurement may require the development and maintenance of specialized software tools. It is difficult to justify the costs of tool development when the returns from measurement are unknown.
4. In many companies, software processes are not standardized and are poorly defined and controlled. As such, there is too much process variability within the same company for measurements to be used in a meaningful way.
5. Much of the research on software measurement and metrics has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by reusing and configuring existing application

systems, or by using agile methods. We don't know how previous research on metrics applies to these software development techniques.

6. Introducing measurement adds overhead to processes. This contradicts the aims of agile methods, which recommend the elimination of process activities that are not directly related to program development. Companies that have adopted agile methods are therefore not likely to adopt a metrics program.

Software measurement and metrics are the basis of empirical software engineering. In this research area, experiments on software systems and the collection of data about real projects have been used to form and validate hypotheses about software engineering methods and techniques. Researchers working in this area argue that we can be confident of the value of software engineering methods and techniques only if we can provide concrete evidence that they actually provide the benefits their inventors suggest.

However, research on empirical software engineering has not had a significant impact on software engineering practice. It is difficult to relate generic research to an individual project that differs from the research study. Many local factors are likely to be more important than general empirical results. For this reason, researchers in software analytics argue that analysts should not try to draw general conclusions but should provide analyses of the data for specific systems.

24.5.1 Product metrics

Product metrics are predictor metrics used to quantify internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class. Unfortunately, as I have explained earlier in this section, software characteristics that can be easily measured, such as size and cyclomatic complexity, do not have a clear and consistent relationship with quality attributes such as understandability and maintainability. The relationships vary depending on the development processes and technology used and the type of system that is being developed.

Product metrics fall into two classes:

1. *Dynamic metrics*, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.

2. *Static metrics*, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are shown in Figure 24.11.

These types of metrics are related to different quality attributes. Dynamic metrics help to assess the efficiency and reliability of a system. Static metrics help assess the complexity, understandability, and maintainability of a system or its components.

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Figure 24.11 Static software product metrics

A clear relationship usually exists between dynamic metrics and software quality characteristics. It is fairly easy to measure the execution time required for particular functions and to assess the time required to start up a system. These functions relate directly to the system's efficiency. Similarly, the number of system failures and the type of failure can be logged and related directly to the reliability of the software. I have explained how reliability can be measured in Chapter 12.

Static metrics, as shown in Figure 24.11, have an indirect relationship with quality attributes. A large number of different metrics have been proposed, and many experiments have tried to derive and validate the relationships between these metrics and attributes, such as system complexity and maintainability. None of these experiments have been conclusive, but program size and control complexity appear be the most reliable predictors of understandability, system complexity, and maintainability.

The metrics in Figure 24.11 are applicable to any program, but more specific object-oriented metrics have also been proposed. Figure 24.12 summarizes Chidamber and Kemerer's suite (sometimes called the CK suite) of six object-oriented metrics (Chidamber and Kemerer 1994). Although these metrics were originally proposed in the early 1990s, they are still the most widely used object-oriented (OO) metrics. Some UML design tools automatically collect values for these metrics as UML diagrams are created.

El-Amam's review of object-oriented metrics discussed the CK metrics and other OO metrics (El-Amam 2001). It concluded that there was insufficient evidence to understand how these and other object-oriented metrics relate to external software

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so they cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent. Therefore, it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class, and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated, and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Figure 24.12 The CK object-oriented metrics suite

qualities. This situation has not really changed since his analysis in 2001. We still don't know how to use measurements of object-oriented programs to draw reliable conclusions about their quality.

24.5.2 Software component analysis

A measurement process that may be part of a software quality assessment process is shown in Figure 24.13. Each system component can be analyzed separately using a range of metrics. The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements, which deviate significantly from the norm, usually indicate problems with the quality of these components.

The key stages in this component measurement process are:

1. *Choose measurements to be made* The questions that the measurement is intended to answer should be formulated and the measurements required to

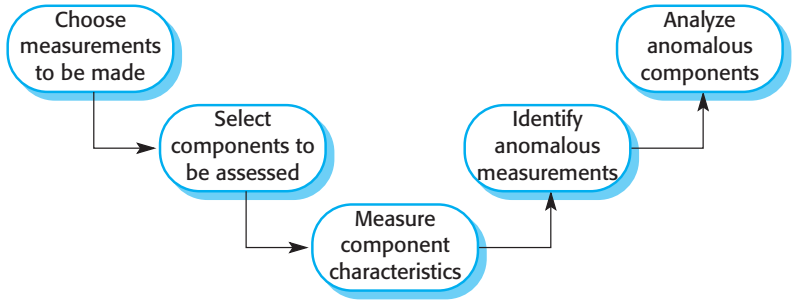


Figure 24.13 The process of product measurement

answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.

2. *Select components to be assessed* You may not need to assess metric values for all of the components in a software system. Sometimes you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are infrequently executed.

3. *Measure component characteristics* The selected components are measured, and the associated metric values are computed. This step normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may be a feature of design tools that are already in use.

4. *Identify anomalous measurements* After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.

5. *Analyze anomalous components* When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor-quality component. There may be some other reason for the high value, so there may not be any component quality problems.

If possible, you should maintain all collected data as an organizational resource and keep historical records of all projects even when data has not been used during a particular project. Once a sufficiently large measurement database has been established, you can then make comparisons of software quality across projects and validate the relations between internal component attributes and quality characteristics.

24.5.3 Measurement ambiguity

When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning. It is easy to misinterpret data and to make incorrect inferences. You cannot simply look at the data on its own. You must also consider the context in which the data is collected.

To illustrate how collected data can be interpreted in different ways, consider the following scenario, which is concerned with the number of change requests made by a system's users:

> *A manager decides to measure the number of change requests submitted by customers based on an assumption that there is a relationship between these change requests and product usability and suitability. She assumes that the higher the number of change requests, the less the software meets the needs of the customer.*
>
> *Handling change requests and changing the software are expensive. The organization therefore decides to modify its process with the aim of improving customer satisfaction and, at the same time, reducing the costs of making changes. The intent is that the process changes will result in better products and fewer change requests. Processes are changed to increase customer involvement in the software design process. Beta testing of all products is introduced, and customer-requested modifications are incorporated in the delivered product.*
>
> *After the process changes have been made, the measurement of change requests continues. New versions of products, developed with the modified process, are delivered. In some cases, the number of change requests is reduced; in others, it is increased. The manager is baffled and finds it impossible to understand the effects of the process changes on the product quality.*

To understand why this kind of ambiguity can occur, you have to understand why users might make change requests:

1. The software is not good enough and does not do what customers want it to do. They therefore request changes to deliver the functionality they require.
2. Alternatively, the software may be very good, and so it is widely and heavily used. Change requests may be generated because many software users creatively think of new things that could be done with the software.

Increasing the customer's involvement in the process may reduce the number of change requests for products where the customers were unhappy. The process changes have been effective and have made the software more usable and suitable. Alternatively, however, the process changes may not have worked, and customers may have decided to look for an alternative system. The number of change requests might decrease because the product has lost market share to a rival product and there are consequently fewer product users.

On the other hand, the process changes might lead to many new, happy customers who wish to participate in the product development process. They therefore generate more change requests. Changes to the process of handling change requests may contribute to this increase. If the company is more responsive to customers, they may generate more change requests because they know that these requests will be taken seriously. They believe that their suggestions will probably be incorporated in later versions of the software. Alternatively, the number of change requests might have increased because the beta-test sites were not typical of most usage of the program.

To analyze the change request data, you do not simply need to know the number of change requests. You need to know who made the request, how the software is used, and why the request was made. You also need information about external factors such as modifications to the change request procedure or market changes that might have an effect. With this information, you are in a better position to find out if the process changes have been effective in increasing product quality.

This illustrates the difficulties of understanding the effects of changes. The "scientific" approach to this problem is to reduce the number of factors that might affect the measurements made. However, processes and products that are being measured are not insulated from their environment. The business environment is constantly changing, and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid. As such, quantitative data about human activities cannot always be taken at face value. The reasons a measured value changes are often ambiguous. These reasons must be investigated in detail before any conclusions can be drawn from any measurements.

24.5.4 Software analytics

Over the past few years, the notion of "big data analysis" has emerged as a means of discovering insights by automatically mining and analyzing very large volumes of automatically collected data. It is possible to discover relationships between data items that could not be found by manual data analysis and modeling. Software analytics is the application of such techniques to data about software and software processes.

Two factors have made software analytics possible:

1. The automated collection of user data by software product companies when their product is used. If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer. As a result, large volumes of data about individual products such as Internet Explorer or Photoshop have become available for analysis.

2. The use of open-source software available on platforms such as Sourceforge and GitHub and open-source repositories of software engineering data (Menzies and Zimmermann 2013). The source code of open-source software is available for automated analysis and can sometimes be linked with data in the open-source repository.

Menzies and Zimmerman (Menzies and Zimmermann 2013) define software analytics as:

> *Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.*

Menzies and Zimmermann emphasize that the point of analytics is not to derive general theories about software but to identify specific issues that are of interest to software developers and managers. Analytics aims to provide information about these issues in real time so that actions can be taken in response to the information provided by the analysis. In a study of managers at Microsoft, Buse and Zimmermann (Buse and Zimmermann 2012) identified information needs such as how to target testing, inspections, and refactoring, when to release software, and how to understand the needs of software customers.

A range of different data mining and analysis tools can be used for software analytics (Witten, Frank, and Hall 2011). In general, it is impossible to know what are the best analysis tools to use in a particular situation. You have to experiment with several tools to discover which are most effective. Buse and Zimmerman suggest a number of guidelines for tool use:

- Tools should be easy to use, as managers are unlikely to have experience with analysis.
- Tools should run quickly and produce concise outputs rather than large volumes of information.
- Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- Tools should be interactive and allow managers and developers to explore the analyses. They should recognize that managers and developers are interested in different things. They should not be predictive but should support decision making based on the analysis of past and current data.

Zhang and her colleagues (Zhang et al. 2013) describe an excellent practical application of software analytics for performance debugging. User software was instrumented to collect data on response times and the associated system state. When the response time was greater than expected, this data was sent for analysis. The automated analysis highlighted performance bottlenecks in the software. The development team could then improve the algorithms to eliminate the bottleneck so that performance was improved in a later software release.

At the time of writing, software analytics is immature, and it is too early to say what effect it will have. Not only are there general problems of "big data" processing (Harford 2013), but it will always be the case that our knowledge depends on collected data from large companies. This data is primarily from software products, and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software. Small companies are unlikely to invest in the data collection systems that are required for automated analysis and so they may not be able to use software analytics.

KEY POINTS

- Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability, and so forth. It includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- Software standards are important for quality assurance as they represent an identification of best practice. When developing software, standards provide a solid foundation for building good-quality software.
- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.
- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are then discussed at a code review meeting.
- Agile quality management does not usually rely on a separate quality management team. Instead, it relies on establishing a quality culture where the development team works together to improve software quality.
- Software measurement can be used to gather quantitative data about software and the software process. You may be able to use the values of the software metrics that are collected to make inferences about product and process quality.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems. These components should then be analyzed in more detail.
- Software analytics is the automated analysis of large volumes of software product and process data to discover relationships that may provide insights for project managers and developers.

FURTHER READING

Software Quality Assurance: From Theory to Implementation. An excellent, still relevant, book on the principles and practice of software quality assurance. It includes a discussion of standards such as ISO 9001. (D. Galin, Addison-Wesley, 2004).

"Misleading Metrics and Unsound Analyses." An excellent article by leading metrics researchers that discusses the difficulties of understanding what measurements really mean. (B. Kitchenham, R. Jeffrey and C. Connaughton, *IEEE Software*, 24 (2), March–April 2007). http://dx.doi.org/10.1109/MS.2007.49

"A Practical Guide to Implementing an Agile QA Process on Scrum Projects." This slide set presents an overview of how to integrate software quality assurance with agile development using Scrum. (S. Rayhan, 2008). https://www.scrumalliance.org/system/resource_files/0000/0459/agileqa.pdf

"Software Analytics: So What?" This is a good introductory article that explains what software analytics is and why it is increasingly important. It is the introduction to a special issue on software analytics, and you may find several other articles in that issue to be helpful in understanding software analytics. (T. Menzies and T. Zimmermann, *IEEE Software,* 30 (4), July–August 2013). http://dx.doi.org/10.1109/MS.2013.86

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

24.1. Define the terms quality assurance and quality control. List out the key points included in Humphrey's outline structure for software management.

24.2. Explain how standards may be used to capture organizational wisdom about effective methods of software development. Suggest four types of knowledge that might be captured in organizational standards.

24.3. Discuss the assessment of software quality according to the quality attributes shown in Figure 24.2. You should consider each attribute in turn and explain how it might be assessed

24.4. Briefly describe possible standards that might be used for:

- the use of control constructs in C, C#, or Java;
- reports that might be submitted for a term project in a university;
- the process of making and approving program changes (web Chapter 26); and
- the process of purchasing and installing a new computer.

24.5. Assume you work for an organization that develops database products for individuals and small businesses. This organization is interested in quantifying its software development. Write a report suggesting appropriate metrics and suggest how these can be collected.

24.6. Briefly explain what happens during the software quality review process and the software quality inspection process.

24.7. What problems are likely to arise if formalized program inspections are introduced in a company where some software is developed using agile methods.

24.8. What is a software metric? Define different types of software metrics with examples.

24.9. You work for a software product company and your manager has read an article on software analytics. She asks you to do some research in this area. Survey the literature on analytics and write a short report that summarizes work in software analytics and issues to be considered if analytics is introduced.

24.10 A colleague who is a very good programmer produces software with a low number of defects but consistently ignores organizational quality standards. How should her managers react to this behavior?

REFERENCES

Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Buse, R. P. L., and T. Zimmermann. 2012. "Information Needs for Software Development Analytics." In *Int. Conf. on Software Engineering*, 987–996. doi:10.1109/ICSE.2012.6227122.

Chidamber, S., and C. Kemerer. 1994. "A Metrics Suite for Object-Oriented Design." *IEEE Trans. on Software Eng.* 20 (6): 476–493. doi:10.1109/32.295895.

El-Amam, K. 2001. "Object-Oriented Metrics: A Review of Theory and Practice." National Research Council of Canada. http://seg.iit.nrc.ca/English/abstracts/NRC44190.html.

Fagan, M. E. 1986. "Advances in Software Inspections." *IEEE Trans. on Software Eng.* SE-12 (7): 744–751. doi:10.1109/TSE.1986.6312976.

Harford, T. 2013. "Big Data: Are We Making a Big Mistake?" *Financial Times*, March 28. http://timharford.com/2014/04/big-data-are-we-making-a-big-mistake/

Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.

IEEE. 2003. *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, CA: IEEE Computer Society Press.

Ince, D. 1994. *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill.

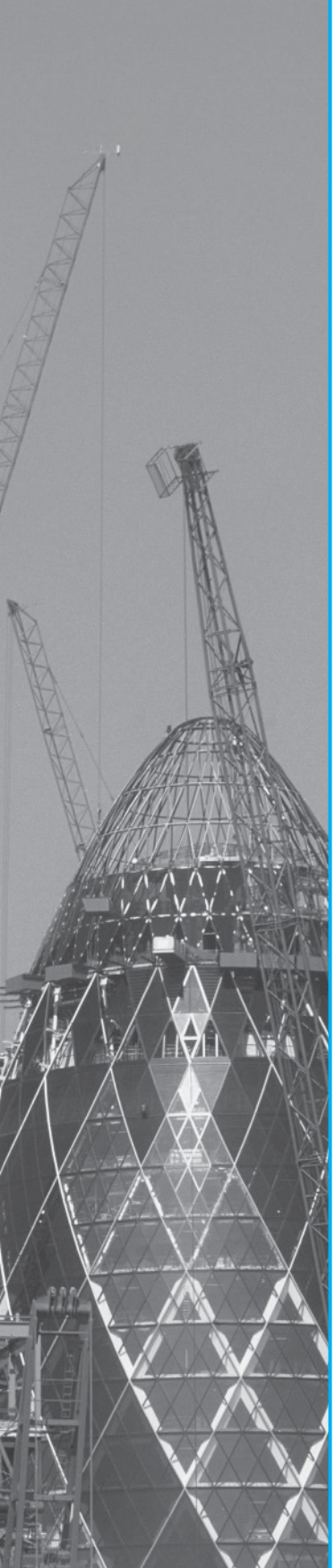
Kitchenham, B. 1990. "Software Development Cost Models." In *Software Reliability Handbook*, edited by P. Rook, 487–517. Amsterdam: Elsevier.

McConnell, S. 2004. *Code Complete: A Practical Handbook of Software Construction, 2nd ed.* Seattle, WA: Microsoft Press.

Menzies, T., and T. Zimmermann. 2013. "Software Analytics: So What?" *IEEE Software* 30 (4): 31–37. doi:10.1109/MS.2013.86.

Witten, I. H., E. Frank, and M. A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Burlington, MA: Morgan Kaufmann.

Zhang, D, S. Han, Y. Dang, J-G. Lou, H. Zhang, and T. Xie. 2013. "Software Analytics in Practice." *IEEE Software* 30 (5): 30–37. doi:10.1109/MS.2013.94.



25 Configuration management

Objectives

The objective of this chapter is to introduce you to software configuration management processes and tools. When you have read the chapter, you will:

- know the essential functionality that should be provided by a version control system, and how this is realized in centralized and distributed systems;
- understand the challenges of system building and the benefits of continuous integration and system building;
- understand why software change management is important and the essential activities in the change management process;
- understand the basics of software release management and how it differs from version management.

Contents

25.1 Version management
25.2 System building
25.3 Change management
25.4 Release management

Software systems are constantly changing during development and use. Bugs are discovered and have to be fixed. System requirements change, and you have to implement these changes in a new version of the system. New versions of hardware and system platforms are released, and you have to adapt your systems to work with them. Competitors introduce new features in their system that you have to match. As changes are made to the software, a new version of a system is created. Most systems, therefore, can be thought of as a set of versions, each of which may have to be maintained and managed.

Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems (Aiello and Sachs 2011). You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version. Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems. Several versions may be under development and in use at the same time. If you don't have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.

Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made. It is essential for team projects where several developers are working at the same time on a software system. Sometimes these developers are all working in the same place, but, increasingly, development teams are distributed with members in different locations across the world. The configuration management system provides team members with access to the system being developed and manages the changes that they make to the code.

The configuration management of a software system product involves four closely related activities (Figure 25.1):

1. *Version control* This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
2. *System building* This is the process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.
3. *Change management* This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
4. *Release management* This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Because of the large volume of information to be managed and the relationships between configuration items, tool support is essential for configuration management. Configuration management tools are used to store versions of system components, build systems from these components, track the releases of system versions to

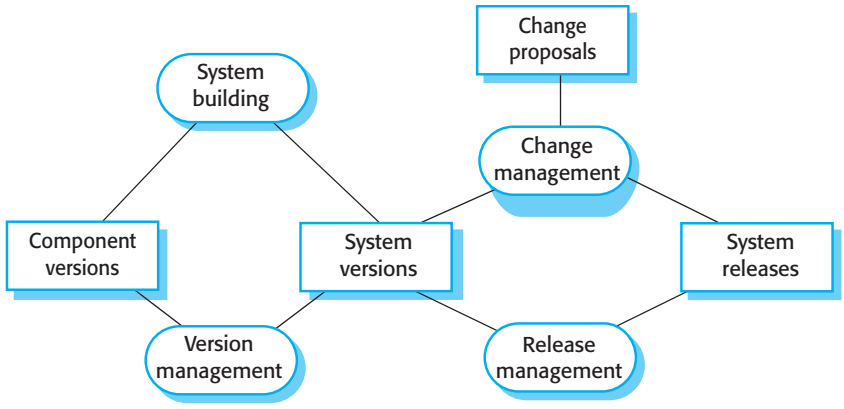


Figure 25.1 Configuration management activities

customers, and keep track of change proposals. CM tools range from simple tools that support a single configuration management task, such as bug tracking, to integrated environments that support all configuration management activities.

Agile development, where components and systems are changed several times a day, is impossible without using CM tools. The definitive versions of components are held in a shared project repository, and developers copy them into their own workspace. They make changes to the code and then use system-building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository. This makes the modified components available to other team members.

The development of a software product or custom software system takes place in three distinct phases:

1. *A development phase* where the development team is responsible for managing the software configuration and new functionality is being added to the software. The development team decides on the changes to be made to the system.
2. *A system testing phase* where a version of the system is released internally for testing. This may be the responsibility of a quality management team or an individual or group within the development team. At this stage, no new functionality is added to the system. The changes made at this stage are bug fixes, performance improvements, and security vulnerability repairs. There may be some customer involvement as beta testers during this phase.
3. *A release phase* where the software is released to customers for use. After the release has been distributed, customers may submit bug reports and change requests. New versions of the released system may be developed to repair bugs and vulnerabilities and to include new features suggested by customers.

For large systems, there is never just one "working" version of a system; there are always several versions of the system at different stages of development. Several

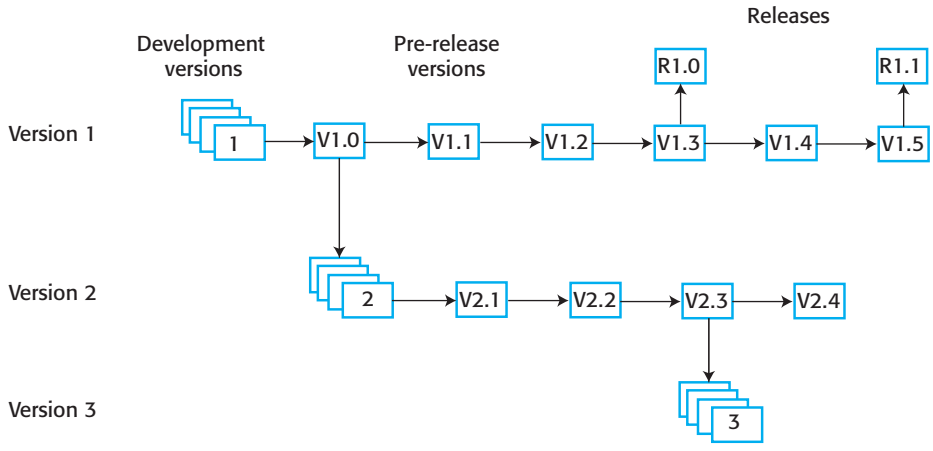


Figure 25.2 Multiversion system development

teams may be involved in the development of different system versions. Figure 25.2 shows situations where three versions of a system are being developed:

1. Version 1.5 of the system has been developed to repair bug fixes and improve the performance of the first release of the system. It is the basis of the second system release (R1.1).
2. Version 2.4 is being tested with a view to it becoming release 2.0 of the system. No new features are being added at this stage.
3. Version 3 is a development system where new features are being added in response to change requests from customers and the development team. This will eventually be released as release 3.0.

These different versions have many common components as well as components or component versions that are unique to that system version. The CM system keeps track of the components that are part of each version and includes them as required in the system build.

In large software projects, configuration management is sometimes part of software quality management (covered in Chapter 24). The quality manager is responsible for both quality management and configuration management. When a pre-release version of the software is ready, the development team hands it over to the quality management team. The QM team checks that the system quality is acceptable. If so, it then becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.

Many specialized terms are used in configuration management. Unfortunately, these are not standardized. Military software systems were the first systems in which software CM was used, so the terminology for these systems reflected the processes and terminology used in hardware configuration management. Commercial systems developers did not know about military procedures or terminology and so often invented their own terms. Agile methods have also devised new terminology in order to distinguish the agile approach from traditional CM methods.

Term	Explanation
Baseline	A collection of component versions that make up a system. Baselines are controlled, which means that the component versions used in the baseline cannot be changed. It is always possible to re-create a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. Configuration items always have a unique identifier.
Mainline	A sequence of baselines representing different versions of a system.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Figure 25.3 CM terminology

The definition and use of configuration management standards are essential for quality certification in both ISO 9000 and the SEI's capability maturity model (Bamford and Deibler 2003; Chrissis, Konrad, and Shrum 2011). CM standards in a company may be based on generic standards such as IEEE 828-2012, an IEEE standard for configuration management. These standards focus on CM processes and the documents produced during the CM process (IEEE 2012). Using the external standards as a starting point, companies may then develop more detailed, company-specific standards that are tailored to their specific needs. However, agile methods rarely use these standards because of the documentation overhead involved.

25.1 Version management

Version management is the process of keeping track of different versions of software components and the systems in which these components are used. It also involves ensuring that changes made by different developers to these versions do not interfere with each other. In other words, version management is the process of managing codelines and baselines.

Figure 25.4 illustrates the differences between codelines and baselines. A codeline is a sequence of versions of source code, with later versions in the sequence derived from earlier versions. Codelines normally apply to components of systems so that there are different versions of each component. A baseline is a definition of a specific system. The baseline specifies the component versions that are included in the system and identifies the libraries used, configuration files, and other system information. In Figure 25.4, you can see that different baselines use different versions of the components from each codeline. In the diagram, I have shaded the boxes representing components in the baseline definition to indicate that these are actually references to components in a codeline. The mainline is a sequence of system versions developed from an original baseline.

Baselines may be specified using a configuration language in which you define what components should be included in a specific version of a system. It is possible to explicitly specify an individual component version (X.1.2, say) or simply to specify the component identifier (X). If you simply include the component identifier in the configuration description, the most recent version of the component should be used.

Baselines are important because you often have to re-create an individual version of a system. For example, a product line may be instantiated so that there are specific system versions for each system customer. You may have to re-create the version delivered to a customer if they report bugs in their system that have to be repaired.

Version control (VC) systems identify, store, and control access to the different versions of components. There are two types of modern version control system:

1. *Centralized systems*, where a single master repository maintains all versions of the software components that are being developed. Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) is a widely used example of a centralized VC system.
2. *Distributed systems*, where multiple versions of the component repository exist at the same time. Git (Loeliger and McCullough 2012), is a widely used example of a distributed VC system.

Centralized and distributed VC systems provide comparable functionality but implement this functionality in different ways. Key features of these systems include:

1. *Version and release identification* Managed versions of a component are assigned unique identifiers when they are submitted to the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.

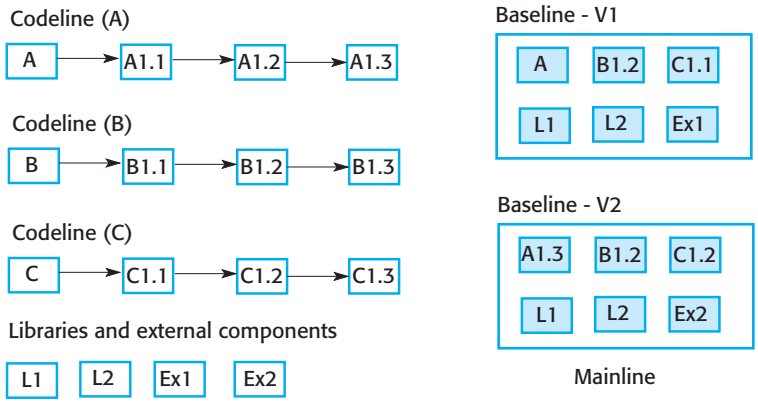


Figure 25.4 Codelines and baselines

2. *Change history recording* The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version. In some systems, these changes may be used to select a particular system version. This involves tagging components with keywords describing the changes made. You then use these tags to select the components to be included in a baseline.

3. *Independent development* Different developers may be working on the same component at the same time. The version control system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

4. *Project support* A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.

5. *Storage management* Rather than maintain separate copies of all versions of a component, the version control system may use efficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version.

Most software development is a team activity, so several team members often work on the same component at the same time. For example, let's say Alice is making some changes to a system, which involves changing components A, B, and C. At the same time, Bob is working on changes that require making changes to components X, Y, and C. Both Alice and Bob are therefore changing C. It's important to avoid situations where changes interfere with each other—Bob's changes to C overwriting Alice's or vice versa.

To support independent development without interference, all version control systems use the concept of a project repository and a private workspace. The project repository maintains the "master" version of all components, which is used to create baselines for system building. When modifying components, developers copy

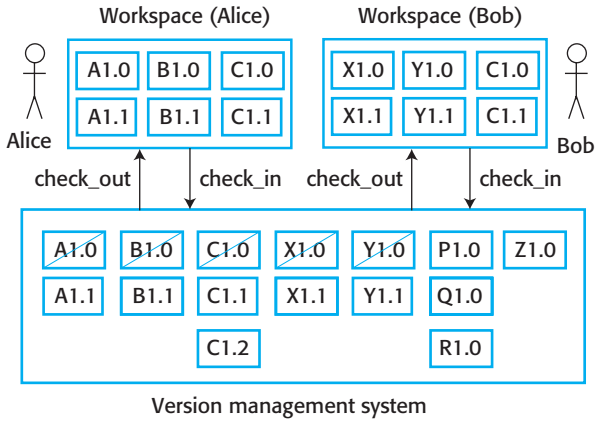


Figure 25.5 Check-in and check-out from a centralized version repository

(check-out) these from the repository into their workspace and work on these copies. When they have completed their changes, the changed components are returned (checked-in) to the repository. However, centralized and distributed VC systems support independent development of shared components in different ways.

In centralized systems, developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace. When their changes are complete, they check-in the components back to the repository. This creates a new component version that may then be shared. For an illustration, see Figure 25.5.

Here, Alice has checked out versions A1.0, B1.0, and C1.0. She has worked on these versions and has created new versions A1.1, B1.1, and C1.1. She checks these new versions into the repository. Bob checks out X1.0, Y1.0, and C1.0. He creates new versions of these components and checks them back in to the repository. However, Alice has already created a new version of C, while Bob has been working on it. His check-in therefore creates another version C1.2, so that Alice's changes are not overwritten.

If two or more people are working on a component at the same time, each must check out the component from the repository. If a component has been checked out, the version control system warns other users wanting to check out that component that it has been checked out by someone else. The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are stored separately.

In a distributed VC system, such as Git, a different approach is used. A "master" repository is created on a server that maintains the code produced by the development team. Instead of simply checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on his or her computer.

Developers work on the files required and maintain the new versions on their private repository on their own computer. When they have finished making changes, they "commit" these changes and update their private server repository. They may then "push" these changes to the project repository or tell the integration manager that changed versions are available. He or she may then "pull" these files to the project repository (see Figure 25.6). In this example, both Bob and Alice have cloned the project repository and have updated files. They have not yet pushed these back to the project repository.

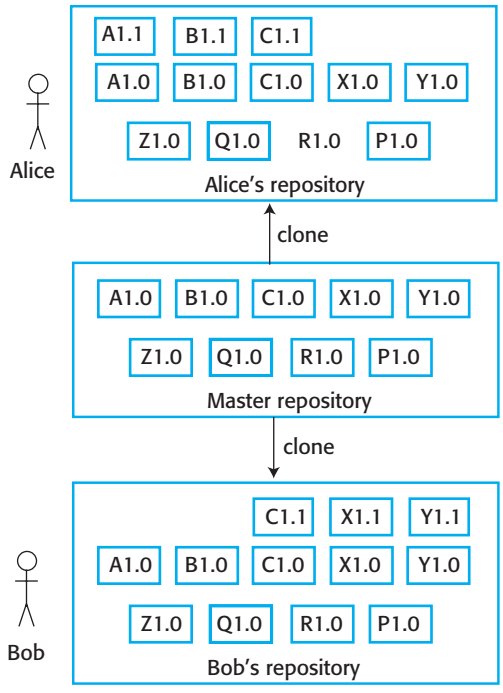


Figure 25.6 Repository cloning

This model of development has a number of advantages:

1. It provides a backup mechanism for the repository. If the repository is corrupted, work can continue and the project repository can be restored from local copies.
2. It allows for offline working so that developers can commit changes if they do not have a network connection.
3. Project support is the default way of working. Developers can compile and test the entire system on their local machines and test the changes they have made.

Distributed version control is essential for open-source development where several people may be working simultaneously on the same system without any central coordination. There is no way for the open-source system "manager" to know when changes will be made. In this case, as well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed. It is then up to the open-source system "manager" to decide when to pull these changes into the definitive system. This organization is shown in Figure 25.7.

In this example, Charlie is the integration manager for the open-source system. Alice and Bob work independently on system development and clone the definitive project repository (1). As well as their private repositories, both Alice and Bob maintain a public repository on a server than can be accessed by Charlie. When they have made and tested changes, they push the changed versions from their private repositories to their personal public repositories and tell Charlie that these repositories are available (2). Charlie pulls these from their repositories into his

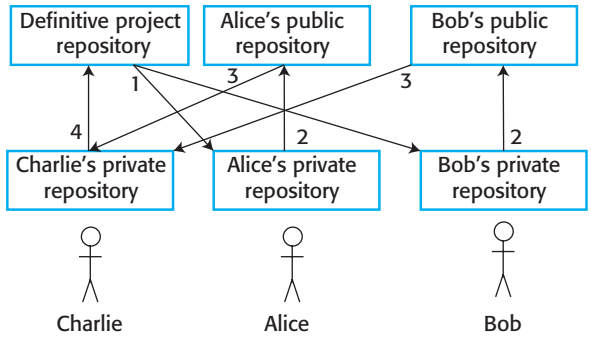


Figure 25.7 Open-source development

own private repository for testing (3). Once he is satisfied that the changes are acceptable, he then updates the definitive project repository (4).

A consequence of the independent development of the same component is that codelines may branch. Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences, as shown in Figure 25.8. This is normal in system development, where different developers work independently on different versions of the source code and change it in different ways. It is generally recommended when working on a system that a new branch should be created so that changes do not accidentally break a working system.

At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made. This is also shown in Figure 25.8, where component versions 2.1.2 and 2.3 are merged to create version 2.4. If the changes made involve completely different parts of the code, the component versions may be merged automatically by the version control system by combining the code changes. This is the normal mode of operation when new features have been added. These code changes are merged into the master copy of the system. However, the changes made by different developers sometimes overlap. The changes may be incompatible and interfere with each other. In this case, a developer has to check for clashes and make changes to the components to resolve the incompatibilities between the different versions.

When version control systems were first developed, storage management was one of their most important functions. Disk space was expensive, and it was important to

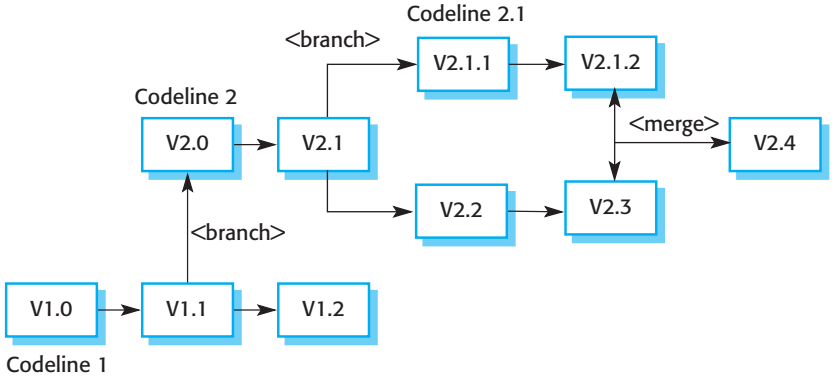


Figure 25.8 Branching and merging

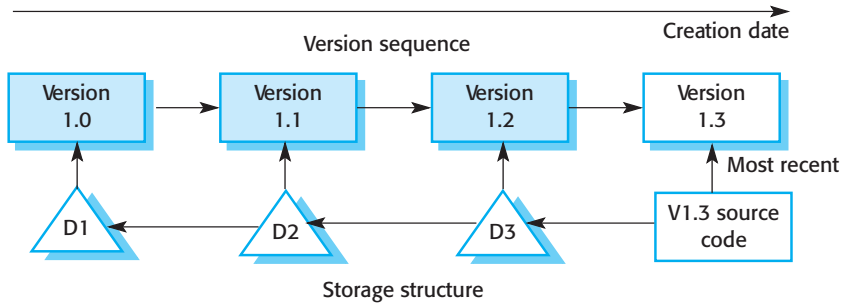


Figure 25.9 Storage management using deltas

minimize the disk space used by the different copies of components. Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another. By applying these to a master version (usually the most recent version), a target version can be re-created. This is illustrated in Figure 25.9.

When a new version is created, the system simply stores a delta, a list of differences, between the new version and the older version used to create that new version. In Figure 25.9, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recent component version. Deltas are usually stored as lists of changed lines, and, by applying these automatically, one version of a component can be created from another. As the most recent version of a component will most likely be the one used, most systems store that version in full. The deltas then define how to re-create earlier system versions.

One of the problems with a delta-based approach to storage management is that it can take a long time to apply all of the deltas. As disk storage is now relatively cheap, Git uses an alternative, faster approach. Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information. It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations. Git also uses the notion of packfiles where several smaller files are combined into an indexed single file. This reduces the overhead associated with lots of small files. Deltas are used within packfiles to further reduce their size.

25.2 System building

System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other information. System-building tools and version control tools must be integrated as the build process takes component versions from the repository managed by the version control system.

System building involves assembling a large amount of information about the software and its operating environment. Therefore, it always makes sense to use an automated build tool to create a system build (Figure 25.10). Notice that you don't just need the source code files that are involved in the build. You may have to link these with externally provided libraries, data files (such as a file of error messages), and configuration files that define the target installation. You may have to specify the versions of

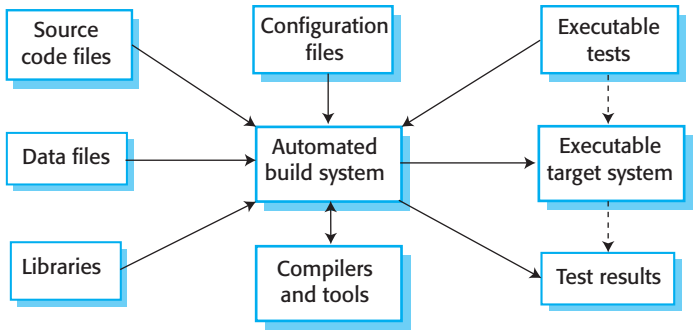


Figure 25.10 System building

the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with a single command or mouse click.

Tools for system integration and building include some or all of the following features:

1. *Build script generation* The build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (configuration file). The system should also support the manual creation and editing of build scripts.
2. *Version control system integration* The build system should check out the required versions of components from the version control system.
3. *Minimal recompilation* The build system should work out what source code needs to be recompiled and set up compilations if required.
4. *Executable system creation* The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.
5. *Test automation* Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been "broken" by changes.
6. *Reporting* The build system should provide reports about the success or failure of the build and the tests that have been run.
7. *Documentation generation* The build system may be able to generate release notes about the build and system help pages.

The build script is a definition of the system to be built. It includes information about components and their dependencies, and the versions of tools used to compile and link the system. The configuration language used to define the build script includes constructs to describe the system components to be included in the build and their dependencies.

Building is a complex process, which is potentially error-prone, as three different system platforms may be involved (Figure 25.11):

1. *The development system*, which includes development tools such as compilers and source code editors. Developers check out code from the version control system into

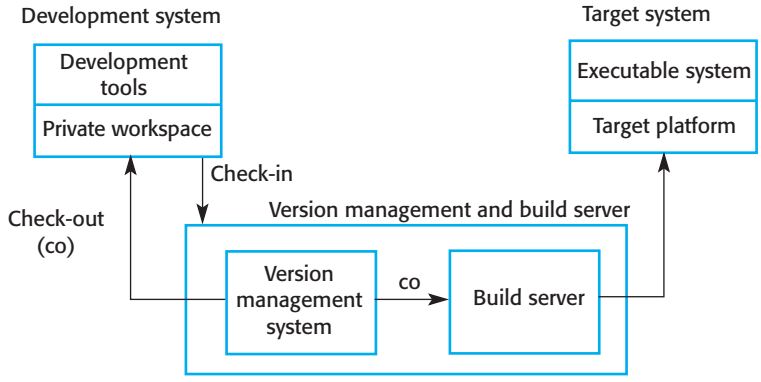


Figure 25.11 Development, build, and target platforms

a private workspace before making changes to the system. They may wish to build a version of a system for testing in their development environment before committing changes that they have made to the version control system. This involves using local build tools that use checked-out versions of components in the private workspace.

2. *The build server*, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system. All of the system developers check in code to the version control system on the build server for system building.

3. *The target environment*, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other application systems that cannot be installed on development machines. In these situations, it is not possible to build and test the system on the development computer or on the build server.

Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration as shown in Figure 25.12. In keeping with the agile methods notion of making many small changes, continuous integration involves rebuilding the mainline frequently, after small source code changes have been made. The steps in continuous integration are:

1. Extract the mainline system from the VC system into the developer's private workspace.

2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken, and you should inform whoever checked in the last baseline system. He or she is responsible for repairing the problem.

3. Make the changes to the system components.

4. Build the system in a private workspace and rerun system tests. If the tests fail, continue editing.

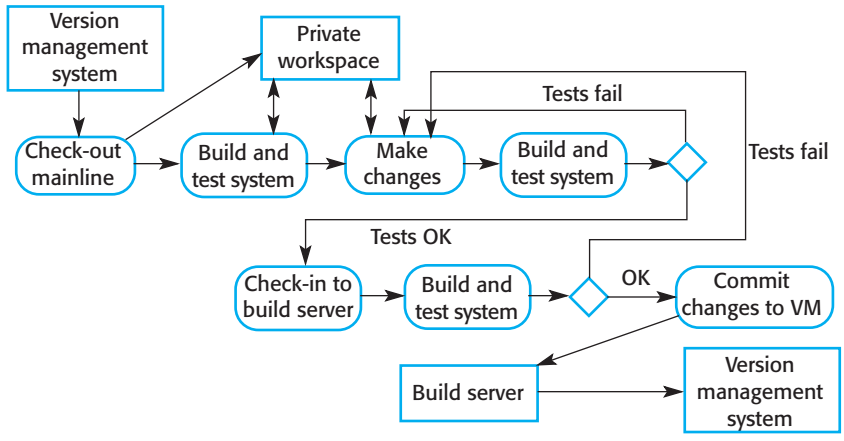


Figure 25.12
Continuous integration

5. Once the system has passed its tests, check it into the build system server but do not commit it as a new system baseline in the VC system.

6. Build the system on the build server and run the tests. Alternatively, if you are using Git, you can pull recent changes from the server to your private workspace. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.

7. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

Tools such as Jenkins (Smart 2011) are used to support continuous integration. These tools can be set up to build a system as soon as a developer has completed a repository update.

The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible. The most recent system in the mainline is the definitive working system. However, although continuous integration is a good idea, it is not always possible to implement this approach to system building:

1. If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved. It may be impractical to build the system being developed several times per day.

2. If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system.

For large systems or for systems where the execution platform is not the same as the development platform, continuous integration is usually impossible. In those circumstances, frequent system building is supported using a daily build system:

1. The development organization sets a delivery time (say 2 p.m.) for system components. If developers have new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testing of components. Psychologically, developers are put under pressure not to "break the build"; that is, they try to avoid checking in versions of components that cause the whole system to fail. They are therefore reluctant to deliver new component versions that have not been properly tested. Consequently, less time is spent during system testing discovering and coping with software faults that could have been found by the developer.

As compilation is a computationally intensive process, tools to support system building may be designed to minimize the amount of compilation that is required. They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component. Therefore, there has to be a way of unambiguously linking the source code of a component with its equivalent object code.

This linking is accomplished by associating a unique signature with each file where a source code component is stored. The corresponding object code, which has been compiled from the source code, has a related signature. The signature identifies each source code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code component was used to generate the object code component.

Two types of signature may be used, as shown in Figure 25.13:

1. *Modification timestamps* The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

   For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2014 and 16:58:43:02:14:2014, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2014 and the compiled version was modified at 58 minutes and 43 seconds past 4 on the 14th of February 2014. In this case, the system would automatically recompile Comp.java because the compiled version has an earlier modification date than the most recent version of the component.

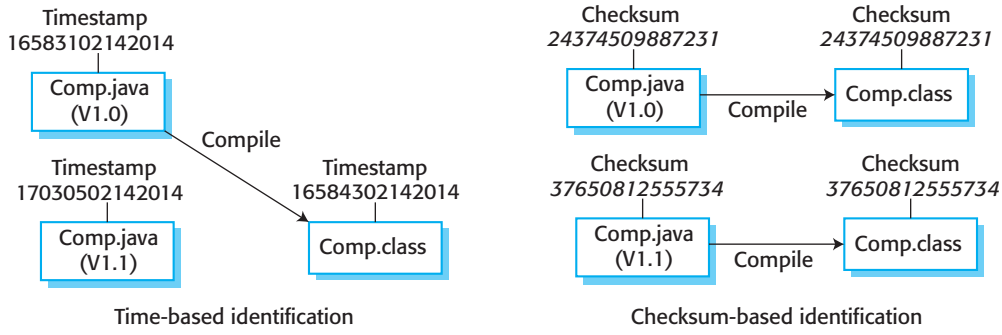


Figure 25.13 Linking source and object code

2. *Source code checksums* The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different. The checksum is assigned to the source code just before compilation and uniquely identifies the source file. The build system then tags the generated object code file with the checksum signature. If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code is necessary.

As object code files are not normally versioned, the first approach means that only the most recently compiled object code file is maintained in the system. This is normally related to the source code file by name; that is, it has the same name as the source code file but with a different suffix. Therefore, the source file Comp.Java may generate the object file Comp.class. Because source and object files are linked by name, it is not usually possible to build different versions of a source code component into the same directory at the same time. The compiler would generate object files with the same name, so only the most recently compiled version would be available.

The checksum approach has the advantage of allowing many different versions of the object code of a component to be maintained at the same time. The signature rather than the filename is the link between source and object code. The source code and object code files have the same signature. Therefore, when you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible, and different versions of a component may be compiled at the same time.

25.3 Change management

Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired, and systems have to adapt to changes in their environment. To ensure that the changes are applied

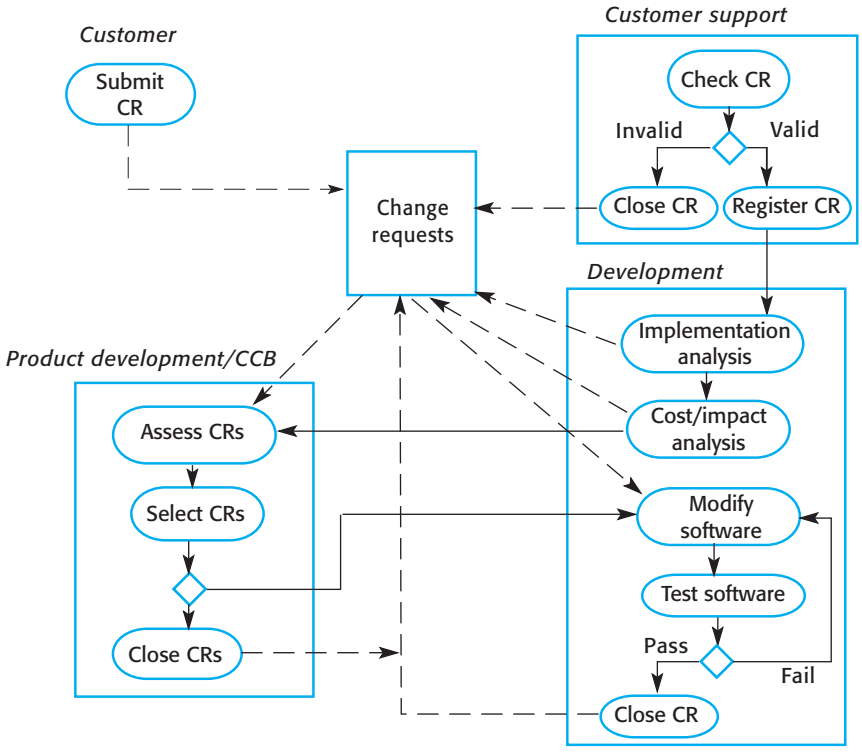


Figure 25.14 The change management process

to the system in a controlled way, you need a set of tool-supported, change management processes. Change management is intended to ensure that the evolution of the system is controlled and that the most urgent and cost-effective changes are prioritized.

Change management is the process of analyzing the costs and benefits of proposed changes, approving those changes that are cost-effective, and tracking which components in the system have been changed. Figure 25.14 is a model of a change management process that shows the main change management activities. This process should come into effect when the software is handed over for release to customers or for deployment within an organization.

Many variants of this process are in use depending on whether the software is a custom system, a product line, or an off-the-shelf product. The size of the company also makes a difference—small companies use a less formal process than large companies that are working with corporate or government customers. However, all change management processes should include some way of checking, costing, and approving changes.

Tools to support change management may be relatively simple issue or bug tracking systems or software that is integrated with a configuration management package for large-scale systems, such as Rational Clearcase. Issue tracking systems allow anyone to report a bug or make a suggestion for a system change, and they keep track of how the development team has responded to the issues. These systems do not impose a process on the users and so can be used in many different settings. More complex systems are built around a process model of the change management process. They

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QM: **QM decision:**
Date submitted to CM:
Comments:

Figure 25.15 A partially completed change request form

automate the entire process of handling change requests from the initial customer proposal to final change approval and change submission to the development team.

The change management process is initiated when a system stakeholder completes and submits a change request describing the change required to the system. This could be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system. Some companies handle bug reports and new requirements separately, but, in principle, both are simply change requests. Change requests may be submitted using a change request form (CRF). Stakeholders may be system owners and users, beta testers, developers, or the marketing department of a company.

Electronic change request forms record information that is shared between all groups involved in change management. As the change request is processed, information is added to the CRF to record decisions made at each stage of the process. At any time, it therefore represents a snapshot of the state of the change request. In addition to recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change, and the dates when the change was requested, approved, implemented, and validated. The CRF may also include a section where a developer outlines how the change may be implemented. Again, the degree of formality in the CRF varies depending on the size and type of organization that is developing the system.

Figure 25.15 is an example of a type of CRF that might be used in a large complex systems engineering project. For smaller projects, I recommend that change requests should be formally recorded; the CRF should focus on describing the



Customers and changes

Agile methods emphasize the importance of involving customers in the change prioritization process. The customer representative helps the team decide on the changes that should be implemented in the next development iteration. While this can be effective for systems that are in development for a single customer, it can be a problem in product development where no real customer is working with the team. In those cases, the team has to make its own decisions on change prioritization.

http://software-engineering-book.com/web/agile-changes/

change required, with less emphasis on implementation issues. System developers decide how to implement the change and estimate the time required to complete the change implementation.

After a change request has been submitted, it is checked to ensure that it is valid. The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. The change request may be rejected at this stage. If the change request is a bug report, the bug may have already been reported and repaired. Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do. On occasions, people request features that have already been implemented but that they don't know about. If any of these features are true, the issue is closed and the form is updated with the reason for closure. If it is a valid change request, it is then logged as an outstanding request for subsequent analysis.

For valid change requests, the next stage of the process is change assessment and costing. This function is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change. The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change. If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation. Next, the required changes to the system modules are assessed. Finally, the cost of making the change is estimated, taking into account the costs of changing related components.

Following this analysis, a separate group decides if it is cost-effective for the business to make the change to the software. For military and government systems, this group is often called the change control board (CCB). In industry, it may be called something like a "product development group" responsible for making decisions about how a software system should evolve. This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, web pages, or documents. These small requests should be passed to the development team for immediate implementation.

The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides whether the change in question is economically justified, and it prioritizes accepted changes for implementation. Accepted changes are passed back to the development group;

rejected change requests are closed and no further action is taken. The factors that influence the decision on whether or not to implement a change include:

1. *The consequences of not making the change* When assessing a change request, you have to consider what will happen if the change is not implemented. If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious, and failure to make the change may disrupt the operational use of the system. On the other hand, if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly. The change should therefore have a low priority.
2. *The benefits of the change* Will the change benefit many users of the system, or will it only benefit the change proposer?
3. *The number of users affected by the change* If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it means that the majority of system users have to adapt to it.
4. *The costs of making the change* If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected.
5. *The product release cycle* If a new version of the software has just been released to customers, it may make sense to delay implementation of the change until the next planned release (see Section 25.4).

Change management for software products (e.g., a CAD system product), rather than custom systems specifically developed for a certain customer, are handled in a different way. In software products, the customer is not directly involved in decisions about system evolution, so the relevance of the change to the customer's business is not an issue. Change requests for these products come from the customer support team, the company marketing team, and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.

The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released. Customers may use a web page or email to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests. Marketing staff may meet with customers and investigate competitive products. They may suggest changes that should be included to make it easier to sell a new version of a system to new and existing customers. The system developers themselves may have some good ideas about new features that can be added to the system.

The change request process shown in Figure 25.14 is initiated after a system has been released to customers. During development, when new versions of the system are created through daily (or more frequent) system builds, there is no need for a formal change management process. Problems and requested changes are recorded in an issue tracking system and discussed in daily meetings. Changes that only affect individual components are passed directly to the system developer, who either accepts them or makes a case for why they are not required. However, an independent

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version   Modifier   Date         Change       Reason
// 1.0       J. Jones   11/11/2009   Add header   Submitted to CM
// 1.1       R. Looek   13/11/2009   New field    Change req. R07/02
```

Figure 25.16
Derivation history

authority, such as the system architect, should assess and prioritize changes that cut across system modules that have been produced by different development teams.

In some agile methods, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system. However, changes that involve software improvement are left to the discretion of the programmers working on the system. Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component. A good way to keep the derivation history is in a standardized comment at the beginning of the component source code (Figure 25.16). This comment should reference the change request that triggered the software change. These comments can be processed by scripts that scan all components for the derivation histories and then generate component change reports. For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document. I discuss this in the web chapter on documentation (Chapter 30).

25.4 Release management

A system release is a version of a software system that is distributed to customers. For mass-market software, it is usually possible to identify two types of release: major releases, which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported. For example, this book is being written on an Apple Mac computer where the operating system is OS 10.9.2. This means minor release 2 of major release 9 of OS 10. Major releases are very important economically to the software vendor, as customers usually have to pay for them. Minor releases are usually distributed free of charge.

A software product release is not just the executable code of the system. The release may also include:

- configuration files defining how the release should be configured for particular installations;
- data files, such as files of error messages in different languages, that are needed for successful system operation;
- an installation program that is used to help install the system on target hardware;
- electronic and paper documentation describing the system;
- packaging and associated publicity that have been designed for that release.

Preparing and distributing a system release for mass-market products is an expensive process. In addition to the technical work involved in creating a release distribution, advertising and publicity material have to be prepared. Marketing strategies may have to be designed to convince customers to buy the new release of the system. Careful thought must be given to release timing. If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it. If system releases are infrequent, market share may be lost as customers move to alternative systems.

The various technical and organizational factors that you should take into account when deciding on when to release a new version of a software product are shown in Figure 25.17.

Release creation is the process of creating the collection of files and documentation that include all components of the system release. This process involves several steps:

1. The executable code of the programs and all associated data files must be identified in the version control system and tagged with the release identifier.
2. Configuration descriptions may have to be written for different hardware and operating systems.
3. Updated instructions may have to be written for customers who need to configure their own systems.
4. Scripts for the installation program may have to be written.
5. Web pages have to be created describing the release, with links to system documentation.
6. Finally, when all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

For custom software or software product lines, the complexity of the system release management process depends on the number of system customers. Special releases of the system may have to be produced for each customer. Individual customers may be running several different releases of the system at the same time on different hardware. Where the software is part of a complex system of systems, several

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date. For marketing reasons, it may be necessary to include new features in a system so that users can be persuaded to upgrade from a previous release.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported that affect the way in which many customers use the system, it may be necessary to correct them in a new system release. Minor system faults may be repaired by issuing patches, distributed over the Internet, which can be applied to the current release of the system.

Figure 25.17 Factors influencing system release planning

different variants of the individual systems may have to be created. For example, in specialized fire-fighting vehicles, each type of vehicle may have its own version of a software system that is adapted to the equipment in that vehicle.

A software company may have to manage tens or even hundreds of different releases of their software. Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions. In the event of a problem with a delivered system, you have to be able to recover all of the component versions used in that specific system.

Therefore, when a system release is produced, it must be documented to ensure that it can be re-created exactly in the future. This is particularly important for customized, long-lifetime embedded systems, such as military systems and those that control complex machines. These systems may have a long lifetime—30 years in some cases. Customers may use a single release of these systems for many years and may require specific changes to that release long after it has been superseded.

To document a release, you have to record the specific versions of the source code components that were used to create the executable code. You must keep copies of the source code files, corresponding executables, and all data and configuration files. It may be necessary to keep copies of older operating systems and other support software because they may still be in operational use. Fortunately, this no longer means that old hardware always has to be maintained. The older operating systems can run in a virtual machine.

You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software. These tools may be required in order to build exactly the same system at some later date. Accordingly, you may have to store copies of the platform software and the tools used to create the system in the version control system, along with the source code of the target system.

When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy with

an existing system and may not consider it worthwhile to absorb the cost of changing to a new release. New releases of the system cannot, therefore, rely on the installation of previous releases. To illustrate this problem, consider the following scenario:

1. Release 1 of a system is distributed and put into use.
2. Release 2 requires the installation of new data files, but some customers do not need the facilities of release 2 and so remain with release 1.
3. Release 3 requires the data files installed in release 2 and has no new data files of its own.

The software distributor cannot assume that the files required for release 3 have already been installed in all sites. Some sites may go directly from release 1 to release 3, skipping release 2. Some sites may have modified the data files associated with release 2 to reflect local circumstances. Therefore, the data files must be distributed and installed with release 3 of the system.

One benefit of delivering software as a service (SaaS) is that it avoids all of these problems. It simplifies both release management and system installation for customers. The software developer is responsible for replacing the existing release of a system with a new release, which is made available to all customers at the same time. However, this approach requires that all servers running the services be updated at the same time. To support server updates, specialized distribution management tools such as Puppet (Loope 2011) have been developed for "pushing" new software to servers.

KEY POINTS

- Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with version control, system building, change management, and release management. Software tools are available to support all of these processes.
- Version control involves keeping track of the different versions of software components that are created as changes are made to them.
- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these changes in a new release of a system.

- System releases include executable code, data files, configuration files, and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.

FURTHER READING

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. A relatively short, easy-to-read book that gives good practical advice on configuration management practice, especially for agile methods of development. (S. P. Berczuk with B. Appleton, Addison-Wesley, 2003).

"Agile Configuration Management for Large Organizations." This web article describes configuration management practices that can be used in agile development processes, with a particular emphasis on how these can scale to large projects and companies. (P. Schuh, 2007). http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html

Configuration Management Best Practices This is a nicely written book that presents a broader view of configuration management than I have discussed here, including hardware configuration management. It's geared to large systems projects and does not really cover agile development issues. (Bob Aiello and Leslie Sachs, Addison-Wesley, 2011).

"A Behind the Scenes Look at Facebook Release Engineering." This is an interesting article that covers the problems of releasing new versions of large systems in the cloud, something that I haven't discussed in this chapter. The challenge here is to make sure that all of the servers are updated at the same time so that users don't see different versions of the system. (P. Ryan, arstechnica.com, 2012). http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/

"Git SVn Comparison." This wiki compares the Git and Subversion version control systems. (2013, https://git.wiki.kernel.org/index.php/GitSvnComparsion).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

25.1. Suggest five possible problems that could arise if a company does not develop effective configuration management policies and processes.

25.2. In version management, what do codeline and baseline terminologies stand for? List the features included in a version control system.

25.3. Imagine a situation where two developers are simultaneously modifying three different software components. What difficulties might arise when they try to merge the changes they have made?

25.4. Software is now often developed by distributed teams, with team members working at different locations and in different time zones. Suggest features in a version control system that could be included to support distributed software development.

25.5. Describe the difficulties that may arise when building a system from its components. What particular problems might occur when a system is built on a host computer for some target machine?

25.6. With reference to system building, explain why you may sometimes have to maintain obsolete computers on which large software systems were developed.

25.7. A common problem with system building occurs when physical filenames are incorporated in system code and the file structure implied in these names differs from that of the target machine. Write a set of programmer's guidelines that helps avoid this and any other system-building problems that you can think of.

25.8. What are the factors that influence the decision on whether or not a change should be implemented?

25.9. Describe six essential features that should be included in a tool to support change management processes.

25.10. Explain why preparing and distributing a system release for mass-market products is an expensive process.

REFERENCES

Aiello, B., and L. Sachs. 2011. *Configuration Management Best Practices*. Boston: Addison-Wesley.

Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd ed.* Boston: Addison-Wesley.

IEEE. 2012. "IEEE Standard for Configuration Management in Systems and Software Engineering" (IEEE Std 828-2012)." doi:10.1109/IEEESTD.2012.6170935.

Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly and Associates.

Loope, J. 2011. *Managing Infrastructure with Puppet*. Sebastopol, CA: O'Reilly and Associates.

Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly and Associates.

Smart, J. F. 2011. *Jenkins: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

This page intentionally left blank



Glossary

abstract data type

A type that is defined by its operations rather than its representation. The representation is private and may only be accessed by the defined operations.

acceptance testing

Customer tests of a system to decide if it is adequate to meet their needs and so should be accepted from a supplier.

activity chart

A chart used by project managers to show the dependencies between tasks that have to be completed. The chart shows the tasks, the time expected to complete these tasks and the task dependencies. The critical path is the longest path (in terms of the time required to complete the tasks) through the activity chart. The critical path defines the minimum time required to complete the project. Sometimes called a PERT chart.

Ada

A programming language that was developed for the US Department of Defense in the 1980s as a standard language for developing military software. It is based on programming language research from the 1970s and includes constructs such as abstract data types and support for concurrency. It is still used for large, complex military and aerospace systems.

agile manifesto

A set of principles encapsulating the ideas underlying agile methods of software development.

agile methods

Methods of software development that are geared to rapid software delivery. The software is developed and delivered in increments, and process documentation and

bureaucracy are minimized. The focus of development is on the code itself, rather than supporting documents.

algorithmic cost modeling

An approach to software cost estimation where a formula is used to estimate the project cost. The parameters in the formula are attributes of the project and the software itself.

application family

A set of software application programs that have a common architecture and generic functionality. These can be tailored to the needs of specific customers by modifying components and program parameters.

application framework

A set of reusable concrete and abstract classes that implement features common to many applications in a domain (e.g. user interfaces). The classes in the application framework are specialized and instantiated to create an application.

application program interface (API)

An interface, generally specified as a set of operations, that allows access to an application program's functionality. This means that this functionality can be called on directly by other programs and not just accessed through the user interface.

architectural pattern (style)

An abstract description of a software architecture that has been tried and tested in a number of different software systems. The pattern description includes information about where it is appropriate to use the pattern and the organization of the components of the architecture.

architectural view

A description of a software architecture from a particular perspective.

availability

The readiness of a system to deliver services when requested. Availability is usually expressed as a decimal number, so an availability of 0.999 means that the system can deliver services for 999 out of 1000 time units.

B

A formal method of software development that is based on implementing a system by systematic transformation of a formal system specification.

bar chart (Gantt chart)

A chart used by project managers to show the project tasks, the schedule associated with these tasks and the people who will work on them. It shows the tasks' start and end dates and the staff allocations against a timeline.

black-box testing

An approach to testing where the testers have no access to the source code of a system or its components. The tests are derived from the system specification.

BPMN

Business Process Modeling Notation. A notation for defining workflows that describe business processes and service composition.

brownfield software development

The development of software for an environment where there are several existing systems that the software being developed must integrate with.

C

A programming language that was originally developed to implement the Unix system. C is a relatively low-level system implementation language that allows access to the system hardware and which can be compiled to efficient code. It is widely used for low-level systems programming and embedded systems development.

C++

An object-oriented programming language that is a superset of C.

C#

An object-oriented programming language, developed by Microsoft, that has much in common with C++, but which includes features that allow more compile-time type checking.

Capability Maturity Model (CMM)

The Software Engineering Institute's Capability Maturity Model, which is used to assess the level of software development maturity in an organization. It has now been superseded by CMMI, but is still widely used.

Computer-Aided Software Engineering (CASE)

The term that was invented in the 1980s to describe process of developing software using automated tool support. Virtually all software development is now reliant on tool support so the term 'CASE is no longer widely used.

CASE tool

A software tool, such as a design editor or a program debugger, used to support an activity in the software development process.

CASE workbench

An integrated set of CASE tools that work together to support a major process activity such as software design or configuration management. Now often called a programming environment.

change management

A process to record, check, analyze, estimate and implement proposed changes to a software system.

class diagram

A UML diagram types that shows the object classes in a system and their relationships.

client–server architecture

An architectural model for distributed systems where the system functionality is offered as a set of services provided by a server. These are accessed by client computers that make use of the services. Variants of this approach, such as three-tier client–server architectures, use multiple servers.

cloud computing

The provision of computing and/or application services over the Internet using a 'cloud' of servers from an external provider. The 'cloud' is implemented using a large number of commodity computers and virtualization technology to make effective use of these systems.

CMMI

An integrated approach to process capability maturity modeling based on the adoption of good software engineering practice and integrated quality management. It supports discrete and continuous maturity modeling and integrates systems and software engineering process maturity models. Developed from the original Capability Maturity Model.

COCOMO II

See Constructive Cost Modeling.

code of ethics and professional practice

A set of guidelines that set out expected ethical and professional behavior for software engineers. This was defined by the major US professional societies (the ACM and the IEEE) and defines ethical behavior under eight headings: public, client and employer, product, judgment, management, colleagues, profession and self.

Common Request Broker Architecture (CORBA)

A set of standards proposed by the Object Management Group (OMG) that defines distributed component models and communications. Influential in the development of distributed systems but no longer widely used.

component

A deployable, independent unit of software that is completely defined and accessed through a set of interfaces.

component model

A set of standards for component implementation, documentation and deployment. These cover the specific interfaces that may be provided by a component, component naming, component interoperation and component composition. Component models provide the basis for middleware to support executing components.

component-based software engineering (CBSE)

The development of software by composing independent, deployable software components that are consistent with a component model.

conceptual design

The development of a high-level vision of a complex system and a description of its essential capabilities. Designed to be understood by people who are not systems engineers.

configurable application system

An application system product, developed by a system vendor, that offers functionality that may be configured for use in different companies and environments.

configuration item

A machine-readable unit, such as a document or a source code file, that is subject to change and where the change has to be controlled by a configuration management system.

configuration management

The process of managing the changes to an evolving software product. Configuration management involves version management, system building, change management and release management.

Constructive Cost Modeling (COCOMO)

A family of algorithmic cost estimation models. COCOMO was first proposed in the early-1980s and has been modified and updated since then to reflect new technology and changing software engineering practice. COCOMO II is its latest instantiation and is a freely available algorithmic cost estimation model that is supported by open source software tools.

CORBA

See Common Request Broker Architecture.

control metric

A software metric that allows managers to make planning decisions based on information about the software process or the software product that is being developed. Most control metrics are process metrics.

critical system

A computer system whose failure can result in significant economic, human or environmental losses.

COTS system

A Commercial Off-the-Shelf system. The term COTS is now mostly used in military systems. See configurable application system.

CVS

A widely used, open-source software tool used for version management.

data processing system

A system that aims to process large amounts of structured data. These systems usually process the data in batches and follow an input-process-output model. Examples of data processing systems are billing and invoicing systems, and payment systems.

denial of service attack

An attack on a web-based software system that attempts to overload the system so that it cannot provide its normal service to users.

dependability

The dependability of a system is an aggregate property that takes into account the system's safety, reliability, availability, security, resilience and other attributes. The dependability of a system reflects the extent to which it can be trusted by its users.

dependability requirement

A system requirement that is included to help achieve the required dependability for a system. Non-functional dependability requirements specify dependability attribute values; functional dependability requirements are functional requirements that specify how to avoid, detect, tolerate or recover from system faults and failures.

dependability case

A structured document that is used to back up claims made by a system developer about the dependability of a system. Specific types of dependability case are safety cases and security cases.

design pattern

A well-tried solution to a common problem that captures experience and good practice in a form that can be reused. It is an abstract representation than can be instantiated in a number of ways.

digital learning environment

An integrated set of software tools, educational applications and content that is geared to support learning.

distributed system

A software system where the software sub-systems or components execute on different processors.

domain

A specific problem or business area where software systems are used. Examples of domains include real-time control, business data processing and telecommunications switching.

domain model

A definition of domain abstractions, such as policies, procedures, objects, relationships and events. It serves as a base of knowledge about some problem area.

DSDM

Dynamic System Development Method. Claimed to be one of the first agile development methods.

embedded system

A software system that is embedded in a hardware device e.g. the software system in a cell phone. Embedded systems are usually real-time systems and so have to respond in a timely way to events occurring in their environment.

emergent property

A property that only becomes apparent once all of the components of the system have been integrated to create the system.

Enterprise Java Beans (EJB)

A Java-based component model.

enterprise resource planning (ERP) system

A large-scale software system that includes a range of capabilities to support the operation of business enterprises and which provides a means of sharing information across these capabilities. For example, an ERP system may include support for supply chain management, manufacturing and distribution. ERP systems are configured to the requirements of each company using the system.

ethnography

An observational technique that may be used in requirements elicitation and analysis. The ethnographer immerses him or herself in the users' environment and observes their day-to-day work habits. Requirements for software support can be inferred from these observations.

event-based systems

Systems where the control of operation is determined by events that are generated in the system's environment. Most real-time systems are event-based systems.

extreme programming (XP)

A widely-used agile method of software development that includes practices such as scenario-based requirements, test-first development and pair programming.

fault avoidance

Developing software in such a way that faults are not introduced into that software.

fault detection

The use of processes and run-time checking to detect and remove faults in a program before these result in a system failure.

fault tolerance

The ability of a system to continue in execution even after faults have occurred.

fault-tolerant architectures

System architectures that are designed to allow recovery from software faults. These are based on redundant and diverse software components.

formal methods

Methods of software development where the software is modeled using formal mathematical constructs such as predicates and sets. Formal transformation converts this model to code. Mostly used in the specification and development of critical systems.

Gantt chart

See bar chart.

Git

A distributed version management and system building tool where developers take complete copies of the project repository to allow concurrent working.

GitHub

A server that maintains a large number of Git repositories. Repositories may be private or public. The repositories for many open-source projects are maintained on GitHub.

hazard

A condition or state in a system that has the potential to cause or contribute to an accident.

host-target development

A mode of software development where the software is developed on a separate computer from where it is executed. The normal approach to development for embedded and mobile systems.

iLearn system

A digital learning environment to support learning in schools. Used as a case study in this book.

incremental development

An approach to software development where the software is delivered and deployed in increments.

information hiding

Using programming language constructs to conceal the representation of data structures and to control external access to these structures.

inspection

See program inspection.

insulin pump

A software-controlled medical device that can deliver controlled doses of insulin to people suffering from diabetes. Used as a case study in this book.

integrated application system

An application system that is created by integrating two or more configurable application systems or legacy systems.

interface

A specification of the attributes and operations associated with a software component. The interface is used as the means of accessing the component's functionality.

ISO 9000/9001

A set of standards for quality management processes that is defined by the International Standards Organization (ISO). ISO 9001 is the ISO standard that is most applicable to software development. These may be used to certify the quality management processes in an organization.

iterative development

An approach to software development where the processes of specification, design, programming and testing are interleaved.

J2EE

Java 2 Platform Enterprise Edition. A complex middleware system that supports the development of component-based web applications in Java. It includes a component model for Java components, APIs, services, etc.

Java

A widely used object-oriented programming language that was designed by Sun (now Oracle) with the aim of platform independence.

language processing system

A system that translates one language into another. For example, a compiler is a language-processing system that translates program source code to object code.

legacy system

A socio-technical system that is useful or essential to an organization but which has been developed using obsolete technology or methods. Because legacy systems often perform critical business functions, they have to be maintained.

Lehman's Laws

A set of hypotheses about the factors that influence the evolution of complex software systems.

maintenance

The process of making changes to a system after it has been put into operation.

mean time to failure (MTTF)

The average time between observed system failures. Used in reliability specification.

Mentcare system

Mental Health Care Patient Management System. This is a system used to record information about consultations and treatments prescribed for people suffering from mental health problems. Used as a case study in this book.

middleware

The infrastructure software in a distributed system. It helps manage interactions between the distributed entities in the system and the system databases. Examples of middleware are an object request broker and a transaction management system.

misuse case

A description of a possible attack on a system that is associated with a system use case.

model-driven architecture (MDA)

An approach to software development based on the construction of a set of system models, which can be automatically or semi-automatically processed to generate an executable system.

model checking

A method of static verification where a state model of a system is exhaustively analyzed in an attempt to discover unreachable states.

model-driven development (MDD)

An approach to software engineering centered around system models that are expressed in the UML, rather than programming language code. This extends MDA to consider activities other than development such as requirements engineering and testing.

multi-tenant databases

Databases where information from several different organizations is stored in the same database. Used in the implementation of software as a service.

mutual exclusion

A mechanism to ensure that a concurrent process maintains control of memory until updates or accesses have been completed.

.NET

A very extensive framework used to develop applications for Microsoft Windows systems. Includes a component model that defines standards for components in Windows systems and associated middleware to support component execution.

object class

An object class defines the attributes and operations of objects. Objects are created at run-time by instantiating the class definition. The object class name can be used as a type name in some object-oriented languages.

object model

A model of a software system that is structured and organized as a set of object classes and the relationships between these classes. Various different perspectives on the model may exist such as a state perspective and a sequence perspective.

object-oriented (OO) development

An approach to software development where the fundamental abstractions in the system are independent objects. The same type of abstraction is used during specification, design and development.

object constraint language (OCL)

A language that is part of the UML, used to define predicates that apply to object classes and interactions in a UML model. The use of the OCL to specify components is a fundamental part of model-driven development.

Object Management Group (OMG)

A group of companies formed to develop standards for object-oriented development. Examples of standards promoted by the OMG are CORBA, UML and MDA.

open source

An approach to software development where the source code for a system is made public and external users are encouraged to participate in the development of the system.

operational profile

A set of artificial system inputs that reflect the pattern of inputs that are processed in an operational system. Used in reliability testing.

pair programming

A development situation where programmers work in pairs, rather than individually, to develop code. A fundamental part of extreme programming.

peer-to-peer system

A distributed system where there is no distinction between clients and servers. Computers in the system can act as both clients and servers. Peer-to-peer applications include file sharing, instant messaging and cooperation support systems.

People Capability Maturity Model (P-CMM)

A process maturity model that reflects how effective an organization is at managing the skills, training and experience of the people in that organization.

plan-driven process

A software process where all of the process activities are planned before the software is developed.

planning game

An approach to project planning based on estimating the time required to implement user stories. Used in some agile methods.

predictor metric

A software metric that is used as a basis for making predictions about the characteristics of a software system, such as its reliability or maintainability.

probability of failure on demand (POFOD)

A reliability metric that is based on the likelihood of a software system failing when a demand for its services is made.

process improvement

Changing a software development process with the aim of making that process more efficient or improving the quality of its outputs. For example, if your aim is to reduce the number of defects in the delivered software, you might improve a process by adding new validation activities.

process model

An abstract representation of a process. Process models may be developed from various perspectives and can show the activities involved in a process, the artifacts used in the process, constraints that apply to the process, and the roles of the people enacting the process.

process maturity model

A model of the extent to which a process includes good practice and reflective and measurement capabilities that are geared to process improvement.

program evolution dynamics

The study of the ways in which an evolving software system changes. It is claimed that Lehman's Laws govern the dynamics of program evolution.

program generator

A program that generates another program from a high-level, abstract specification. The generator embeds knowledge that is reused in each generation activity.

program inspection

A review where a group of inspectors examine a program, line by line, with the aim of detecting program errors. A checklist of common programming errors often drives inspections.

Python

A programming language with dynamic types, which is particularly well-suited to the development of web-based systems.

quality management (QM)

The set of processes concerned with defining how software quality can be achieved and how the organization developing the software knows that the software has met the required level of quality.

quality plan

A plan that defines the quality processes and procedures that should be used. This involves selecting and instantiating standards for products and processes and defining the system quality attributes that are most important.

rapid application development (RAD)

An approach to software development aimed at rapid delivery of the software. It often involves the use of database programming and development support tools such as screen and report generators.

rate of occurrence of failure (ROCOF)

A reliability metric that is based on the number of observed failures of a system in a given time period.

Rational Unified Process (RUP)

A generic software process model that presents software development as a four-phase iterative activity, where the phases are inception, elaboration, construction and transition. Inception establishes a business case for the system, elaboration defines the architecture, construction implements the system, and transition deploys the system in the customer's environment.

real-time system

A system that has to recognize and process external events in 'real-time'. The correctness of the system does not just depend on what it does but also on how quickly it does it. Real-time systems are usually organized as a set of concurrent processes.

reductionism

An engineering approach that relies on breaking down a problem to sub-problems, solving these sub-problems independently then integrating these solutions to create the solution to the larger problem.

reengineering

The modification of a software system to make it easier to understand and change. Reengineering often involves software and data restructuring and organization, program simplification and redocumentation.

reengineering, business process

Changing a business process to meet a new organizational objective such as reduced cost and faster execution.

refactoring

Modifying a program to improve its structure and readability without changing its functionality.

reference architecture

A generic, idealized architecture that includes all the features that systems might incorporate. It is a way of informing designers about the general structure of that class of system rather than a basis for creating a specific system architecture.

release

A version of a software system that is made available to system customers.

reliability

The ability of a system to deliver services as specified. Reliability can be specified quantitatively as a probability of failure on demand or as the rate of occurrence of failure.

reliability growth modeling

The development of a model of how the reliability of a system changes (improves) as it is tested and program defects are removed.

requirement, functional

A statement of some function or feature that should be implemented in a system.

requirement, non-functional

A statement of a constraint or expected behavior that applies to a system. This constraint may refer to the emergent properties of the software that is being developed or to the development process.

requirements management

The process of managing changes to requirements to ensure that the changes made are properly analyzed and tracked through the system.

resilience

A judgement of how well a system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.

REST

REST (Representational State Transfer) is a style of development based around simple client/server interaction which uses the HTTP protocol for communications. REST is based around the idea of an identifiable resource, which has a URI. All interaction with resources is based on HTTP POST, GET, PUT and DELETE. Widely used for implementing low overhead web services (RESTful services).

revision control systems

See version control systems.

risk

An undesirable outcome that poses a threat to the achievement of some objective. A process risk threatens the schedule or cost of a process; a product risk is a risk that may mean that some of the system requirements may not be achieved. A safety risk is a measure of the probability that a hazard will lead to an accident.

risk management

The process of identifying risks, assessing their severity, planning measures to put in place if the risks arise and monitoring the software and the software process for risks.

Ruby

A programming language with dynamic types that is particularly well-suited to web application programming.

SaaS

See software as a service.

safety

The ability of a system to operate without behavior that may injure or kill people or damage the system's environment.

safety case

A body of evidence and structured argument from that evidence that a system is safe and/or secure. Many critical systems must have associated safety cases that are assessed and approved by external regulators before the system is certified for use.

SAP

A German company that has developed a well-known and widely-used ERP system. It also refers to the name given to the ERP system itself.

scenario

A description of one typical way in which a system is used or a user carries out some activity.

scenario testing

An approach to software testing where test cases are derived from a scenario of system use.

Scrum

An agile method of development, which is based on sprints – short development, cycles. Scrum may be used as a basis for agile project management alongside other agile methods such as XP.

security

The ability of a system to protect itself against accidental or deliberate intrusion. Security includes confidentiality, integrity and availability.

SEI

Software Engineering Institute. A software engineering research and technology transfer center, founded with the aim of improving the standard of software engineering in US companies.

sequence diagram

A diagram that shows the sequence of interactions required to complete some operation. In the UML, sequence diagrams may be associated with use cases.

server

A program that provides a service to other (client) programs.

service

See web service.

socio-technical system

A system, including hardware and software components, that has defined operational processes followed by human operators and which operates within an organization. It is therefore influenced by organizational policies, procedures and structures.

software analytics

Automated analysis of static and dynamic data about software systems to discover relationships between these data. These relationships may provide insights about possible ways to improve the quality of the software.

software architecture

A model of the fundamental structure and organization of a software system.

software as a service (SaaS)

Software applications that are accessed remotely through a web browser rather than installed on local computers. Increasingly used to deliver application services to end-users.

software development life cycle

Often used as another name for the software process. Originally coined to refer to the waterfall model of the software process.

software metric

An attribute of a software system or process that can be expressed numerically and measured. Process metrics are attributes of the process such as the time taken to complete a task; product metrics are attributes of the software itself such as size or complexity.

software process

The activities and processes that are involved in developing and evolving a software system.

software product line

See application family.

spiral model

A model of a development process where the process is represented as a spiral, with each round of the spiral incorporating the different stages in the process. As you move from one round of the spiral to another, you repeat all of the stages of the process.

state diagram

A UML diagram type that shows the states of a system and the events that trigger a transition from one state to another.

static analysis

Tool-based analysis of a program's source code to discover errors and anomalies. Anomalies, such as successive assignments to a variable with no intermediate use may be indicators of programming errors.

structured method

A method of software design that defines the system models that should be developed, the rules and guidelines that should apply to these models and a process to be followed in developing the design.

Structured Query Language (SQL)

A standard language used for relational database programming.

Subversion

A widely-used, open source version control and system building tool that is available on a range of platforms.

Swiss cheese model

A model of system defenses against operator failure or cyberattack that takes vulnerabilities in these defenses into account.

system

A system is a purposeful collection of interrelated components, of different kinds, which work together to deliver a set of services to the system owner and users.

system building

The process of compiling the components or units that make up a system and linking these with other components to create an executable program. System building is normally automated so that recompilation is minimized. This automation may be built in to the language processing system (as in Java) or may involve software tools to support system building.

systems engineering

A process that is concerned with specifying a system, integrating its components and testing that the system meets its requirements. System engineering is concerned

with the whole socio-technical system—software, hardware and operational processes—not just the system software.

system of systems

A system that is created by integrating two or more existing systems.

system testing

The testing of a completed system before it is delivered to customers.

test coverage

The effectiveness of system tests in testing the code of an entire system. Some companies have standards for test coverage e.g. the system tests shall ensure that all program statements are executed at least once.

test-driven development

An approach to software development where executable tests are written before the program code. The set of tests are run automatically after every change to the program.

TOGAF

An architectural framework, supported by the Object Management Group, that is intended to support the development of enterprise architectures for systems of systems.

transaction

A unit of interaction with a computer system. Transactions are independent and atomic (they are not broken down into smaller units) and are a fundamental unit of recovery, consistency and concurrency.

transaction processing system

A system that ensures that transactions are processed in such a way so that they do not interfere with each other and so that individual transaction failure does not affect other transactions or the system's data.

Unified Modeling Language (UML)

A graphical language used in object-oriented development that includes several types of system model that provide different views of a system. The UML has become a *de facto* standard for object-oriented modeling.

unit testing

The testing of individual program units by the software developer or development team.

use case

A specification of one type of interaction with a system.

use-case diagram

A UML diagram type that is used to identify use-cases and graphically depict the users involved. It must be supplemented with additional information to completely describe use-cases.

user interface design

The process of designing the way in which system users can access system functionality, and the way that information produced by the system is displayed.

user story

A natural language description of a situation that explains how a system or systems might be used and the interactions with the systems that might take place.

validation

The process of checking that a system meets the needs and expectations of the customer.

verification

The process of checking that a system meets its specification.

version control

The process of managing changes to a software system and its components so that it is possible to know which changes have been implemented in each version of the component/system, and also to recover/recreate previous versions of the component/system.

version control (VC) systems

Software tools that have been developed to support the processes of version control. These may be based on either centralized or distributed repositories.

waterfall model

A software process model that involves discrete development stages: specification, design, implementation, testing and maintenance. In principle, one stage must be complete before progress to the next stage is possible. In practice, there is significant iteration between stages.

web service

An independent software component that can be accessed through the Internet using standard protocols. It is completely self-contained without external dependencies. XML-based standards such as SOAP (Standard Object Access Protocol), for web service information exchange, and WSDL (Web Service Definition Language), for the definition of web service interfaces, have been developed. However, the REST approach may also be used for web service implementation.

white-box testing

An approach to program testing where the tests are based on knowledge of the structure of the program and its components. Access to source code is essential for white-box testing.

wicked problem

A problem that cannot be completely specified or understood because of the complexity of the interactions between the elements that contribute to the problem.

wilderness weather system

A system to collect data about the weather conditions in remote areas. Used as a case study in this book.

workflow

A detailed definition of a business process that is intended to accomplish a certain task. The workflow is usually expressed graphically and shows the individual process activities and the information that is produced and consumed by each activity.

WSDL

An XML-based notation for defining the interface of web services.

XML

Extended Markup Language. XML is a text markup language that supports the interchange of structured data. Each data field is delimited by tags that give information about that field. XML is now very widely used and has become the basis of protocols for web services.

XP

See Extreme Programming.

Z

A model-based, formal specification language developed at the University of Oxford in England.

This page intentionally left blank



Subject Index

A

abstraction level (reuse), 213
acceptability, 22, 347–48
acceptance testing, 77, 82, 249, 250–51, 252
accidents (mishaps), 343–44, 347
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, 29–30
acquisition (procurement), 473, 553–54, 566–70
activities (software engineering activities), 20, 23, 44, 47–48, 54–61, 142, 298, 643–44. *See also* development; evolution; specification; validation
activity charts (planning), 678–80
activity diagrams (UML), 33–34, 47, 50, 56, 141, 143–44, 163
actuators, 218, 502, 613–14, 615
Ada programming language, 359
adaptors, 469, 482–83
additive composition, 481
Adobe Creative Suite, 27
aggregation, 153
agile methods, 45, 66, 72–100
  architectural design and, 168, 175
  change and, 76, 78, 91, 131–32
  change management and, 97, 748, 750
  configuration management (CM) for, 732, 742–43, 748, 750
  critical systems and, 75, 92, 96
  continuous integration, 742–43
  custom systems and, 90, 732
  customer involvement and, 76, 77, 91, 748, 750
  development team, 85, 90, 92–93
  documentation and, 73–75, 86, 89–90, 92–93, 175
  evolution and, 90, 261
  extreme programming (XP), 73, 77–84
  incremental development and, 45, 50, 73–74, 77
  large system complexity and, 93–96
  manifesto, 75–76, 77–78
  model-driven architecture (MDA) and, 162
  organizations and, 91, 97
  pair programming, 78, 83–84
  'people, not process' and, 76, 77, 91
  plan-driven approach *v.*, 45, 74–75, 91–93, 98
  principles of, 76
  process improvement and, 66
  project management and, 84–88, 643, 647, 661
  project planning, 91–93, 670, 680–83, 696
  quality management (QM), 714–16, 727
  refactoring, 51, 80–81
  risk management and, 647
  scaling, 88–97, 98
  simplicity of, 76, 78, 91
  Scrum approach and, 73, 78, 85–88, 96
  test first development, 59, 78, 81–83
  user stories for, 681–82
  user testing, 251
agile modeling, 50
Agile Scaling Model (ASM), 95
air traffic management (ATC) systems, 554–55, 569

Airbus 340 flight control system, 321–22, 340
AJAX programming, 28, 445, 512
algorithm error, 351–52
algorithmic cost modeling, 683, 684–86
alpha testing, 249
analysis systems, 25
Android, 219
Apache web server, 219
aperiodic stimuli, 613
Apollo 13 mission resilience, 409, 411, 416
application assessment (legacy systems), 269
application data, 262
application frameworks, 442, 443–46, 460
application layer, 292
application-level protection, 393–394
application programming interfaces (APIs), 39, 595–96
application security, 374–375
application software, 262
application system, 53, 438, 453–60
  COTS systems, 453
  ERP systems, 454–457
  reuse, 438, 442, 453–60
architectural description languages (ADLs), 175
architectural design, 57, 149, 167–195, 570–71, 595, 599–606
  block diagrams for, 170
  Booch's architecture catalog and, 170
  decisions, 171–73, 192
  4+1 view model, 173–74
  levels of abstraction, 169
  maintenance and, 172–73, 178
  model-driven architecture (MDA), 159–62
  non-functional requirements for, 169, 172–73
  object-oriented systems, 201–02
  patterns, 175–84, 192
  refactoring and, 168
  security and, 172, 388, 392–395
  structural models for, 149
  system development and, 570–71
  systems of systems (SoS), 595, 599–606
  views, 173–75, 192
architectural frameworks, 600–02
architectural patterns (styles), 172
  client-server architecture, 180–82, 501, 503–06, 517
  container systems, 603–05
  data-feed systems, 602–03
  distributed component systems, 501, 506–09, 517
  distributed systems, 175–84, 192, 501–12, 517
  embedded software and, 620–26, 634
  environmental control, 620, 623–25
  layered architecture, 177–79
  master-slave architecture, 501–02
  model-view-controller (MVC), 176–77
  multi-tier client-server architecture, 501, 505–06
  observe and react, 620, 621–23
  peer-to-peer (p2p) architecture, 501, 509–12, 517
  pipe and filter architecture, 182–84
  process pipeline, 620, 625–26
  real-time software, 620–26, 634
  repository architecture, 179–80
  security and, 172, 388, 392–95
  systems of systems (SoS), 602–606, 607
  trading systems, 605–06
  two-tier client-server architecture, 501, 503–05
Architecture Development Method (ADM), 601
architectures (software architectures)
  application, 184–91, 192
  architecture in the large, 169
  architecture in the small, 169
  defined, 192
  distributed, 171, 182
  fault-tolerant, 318–25
  industrial practice *v*., 170
  pipe and filter compiler, 190–91
  reference, 191
  self-monitoring, 320–22
Ariane 5 explosion, 296, 479, 480
arithmetic error, 351
as low as reasonably practical (ALARP) risks, 347
aspect-oriented software development, 442
Assertion checking, 360
assessment
  hazards for safety requirements, 345, 346–349
  security risk, 381–82
assets, 377, 378, 413, 414–415
assurance
  safety processes, 353–56
  security testing and, 402–04
ATMs (automated teller machines), 186–87, 315–16
attacks, 377, 378–79, 389, 413, 414–15, 494–95
attributes of software, 20, 22, 40
authentication, 413, 414, 416
automated management, 423–24
automated testing, 78, 81–83, 233–34, 242, 252
automatic static analysis, 359–60

availability
  security and, 374, 375, 413
  system availability, 172, 288, 309–12
availability metric (AVAIL), 313–314
avoidance
  error discovery and, 300–01
  fault, 308
  hazard, 342, 351
  strategies (risk management), 650
  vulnerability, 378

B

B method, 49, 300, 301, 357
banking system, Internet, 505
baselines, 734, 735, 736
batch processing systems, 25
behavioral models, 154–59, 163
beta testing, 58, 60, 249–250
bidding (projects), 669, 671–72
bindings, 527–28
blackboard model, 180
block diagrams, 170, 199
Boehm's spiral process model, 48
Booch's software architecture catalog, 170
boundaries (system models), 141–42, 163, 199, 556–57
branching, 734, 739
broadcast (listener) models, 202
Brownfield systems, 94, 256
BSD (Berkeley Standard Distribution) license, 220
Bugzilla, 216
build system, 741–42
burglar alarm system, 614, 622, 629–31
business-critical system, 287
business process layer, 292
Business Process Modeling Notation (BPMN), 544–46
business process models, 544–46
businesses
  activity diagrams (UML) for processes, 143–44
  interrelated 4 R's approach, 426–27
  legacy system evolution, 261–68
  maintenance costs, 274–76, 279
  modeling workflow, 67–68
  open-source software and, 221
  policies (rules), 262
  process maturity models, 67–68
  process reengineering, 276–78
  processes, 262
  rapid software development and, 73–74
  requirements changes, 131
  resilience and, 426–27
  security and, 380–382
  services, 534, 541–47, 548
  social change and, 24
  software systems, 24, 27, 45, 68, 267–68
  system construction by composition, 543–44
  system values, 267–68, 280
  web-based applications, 27
  workflow, 542, 543, 544–46

C

C and C++ programming languages, 197, 327, 330, 359, 360, 401, 444, 619
callbacks, 445
catalog interface design, 537–538
centralized systems, version management of, 735, 737
certification (software dependability), 294, 299, 302, 354, 355–56, 474, 477, 709–10
change, 61–65. *See also* process change
  agile methods and, 73–74, 78, 90–91, 97
  business and social needs, 24
  cost effectiveness of, 133
  cultural (social), 24, 97
  customers and, 748–49
  effects on software engineering, 27–28
  extreme programming (XP) and, 78
  implementation, 134, 259–60, 280
  incremental delivery, 62, 64–65
  plan-driven process and, 73
  problem analysis and, 133
  prototyping, 62–63
  rapid software development for, 73–74
  requirements management for, 111, 130–34
  reuse, 27–28
  rework for, 61, 73

change anticipation, 61
change control board (CCB), 748–49
change management, 97, 731, 745–50, 753
  agile methods and, 97, 748, 750
  change requests, 747–50
  dependability and, 299
  development environments and, 217
  requirements and, 111, 130–34
change proposals, 90, 258–59
change request form (CRF), 747–48
change tolerance, 61
characteristic error checking, 359–60
check array bounds, 330
checking requirements, 317
checklists, 403, 713–714
checksums, 745
circular buffer, 616–17
class diagrams, 141, 149–51, 163
class identification, 202–04
Cleanroom process, 230, 332
client-server architecture, 180–82, 428, 501, 503–06, 517
client-server systems, 499–501, 517
clouds, 25, 27, 532
COBOL code, 263
COCOMO II modeling, 276, 476, 686–96
  application composition model, 688–89
  cost drivers, 692
  early design model, 689–90
  post-architectural level, 692–94
  project duration and staffing, 694–96
  reuse model, 690–92
code coverage, 243–44, 252
code inspection and review, 83, 715
Code of Ethics and Professional Practice (software engineering), 29–30
codelines, 734, 735, 736, 739
collaborative systems, 588
collective ownership, 78
COM platform, 466
Common Intermediate Language (CIL), 470–71
communication
  data management layer and, 292
  message exchange, 496–97, 526–29, 537
  stakeholder, 169
communication latency, 218
compartmentalization, 399
competence, 28
completeness, 107, 129
complexity, 18, 93–96, 274–75, 278, 584–87, 606
  governance, 586–87, 588–90, 606
  large systems, 93–96
  maintenance prediction and, 274–75
  management, 585, 586–87, 587–90, 606
  reductionism for systems, 590–93, 606
  refactoring, 278
  scaling agile methods and, 93–96
  system releases, 751–52
  systems of systems (SoS), 584–87, 606
  technical, 585, 586–87, 590
compliance to software regulation, 294–95
component-based software engineering (CBSE), 442, 464–489
  component certification, 474, 477
  component management, 474, 476
  development for reuse, 473, 474–77
  development with reuse, 473, 477–80
  middleware and, 465, 472–73
  service-oriented software *v*., 466–67
component level (reuse), 214
components (software), 52–53, 188, 190, 295, 424, 465–73, 487, 526–29
  architectural design and, 172
  communications, 172, 218, 526–29
  composition, 480–86, 487
  defined, 465, 467, 487
  deployment, 471, 472–73
  design and selection of, 57, 424, 452
  external, 330–31
  implementation, 465, 466, 471–72, 475, 487
  incompatibility, 481–83
  interfaces, 208–209, 237–239, 465, 468–69
  measurement (analysis), 722–23
  models, 470–73, 487
  open-source, 220–21
  platforms for, 466–67
  remote procedure calls (RPCs) for, 470, 471
  reuse, 52–53, 212, 214, 221, 438–439, 452, 468, 487
  services *v*., 521
  service-oriented architectures (SOA), 526–29
  testing, 59, 232, 237–239
  timeouts, 330–31
  timing errors, 238–239
components (system), procurement (acquisition) of, 567–68

composition
of components, 480–86, 487
service systems and, 541–47
computation independent model (CIM), 159–61
computer science, software engineering *v.*, 20, 23
concept reuse, 439
conceptual system design, 553, 563–66, 577, 594
conceptual views, 174, 192
concurrency, 491
confidence levels (verification), 228–29
confidentiality, 28, 374, 413
configurable application systems, 442, 454–457
configuration management (CM), 213, 215–216, 222, 730–55. *See also* change management
activities of, 215–16
agile methods and, 732, 742–43, 748, 750
architectural patterns, 175
change management, 731, 745–50, 753
design implementation and, 213, 215–16, 222
problem tracking, 216
release management, 216, 731, 750–53, 754
system building, 731, 740–45, 753
system integration and, 215–16
terminology for, 734
version management (VM), 215, 216, 731, 735–40, 753
configuration, software product lines, 451–52
ConOps document standard, 563
consistency, 107, 129, 652
constants, naming of, 331
construction phase (RUP), 46
consumer/producer processes (circular buffer), 616–17
container systems, 603–05
context models, 141–44, 163, 199–200
contingency plans, 650–51
continuous integration, 78, 742–43
control
application frameworks and, 445
cybersecurity, 413–414
inversion of, 445
safety-critical systems, 341–42
security, 377, 378–79
visibility of information, 325–26
control metrics, 717
controlled systems, 319
cooperative interaction patterns, 175
coordination services, 534, 548
CORBA (Common Object Request Broker Architecture), 466, 493, 507
cost/dependability curve, 290–91
cost drivers, 692
costs. *See also* estimation techniques
change analysis and, 133
COCOMO II modeling, 686–96
dependability and, 290–91
distributed systems, 495
effort, 669
fault removal, 308–09
formal verification, 357
maintenance/development, 274–76, 279, 280
overhead, 669
project planning, 669
safety engineering and, 357, 362–63
software engineering, 20
software reuse and, 214, 439
system failure, 286
COTS (commercial-off-the-shelf) systems, 453. *See also* application system reuse
critical systems, 287. *See also* safety-critical systems
agile methods and, 75
dependable processes for, 297
documentation for, 92, 96
failure of, 287, 303
formal methods for dependability of, 302
redundancy and, 295
types of, 287, 424
verification and validation costs, 290
cultural change, 97
customer involvement (agile methods), 76, 77, 91, 748, 750
customer testing, 59
customization, 471, 732–33
cybersecurity, 376, 412–416, 432

D

damage limitation, 342, 351
data clumping, 279
data collection systems, 25, 202
data flow diagrams (DFD), 154–55
data reengineering, 277
database design, 57

data-driven modeling, 154–55
data-feed systems, 602–03
data-mining system, 508
deadlines (real-time systems), 627
debugging, 58, 216, 232, 244
decentralized systems, 510–11, 517
Decorator pattern, 212
defect testing, 58, 227–28, 232
  debugging *v*., 58, 232
  performance, 248
  release testing, 248
deltas (storage management), 740
denial-of-service attacks, 289–90, 389, 423
Department of Defense Architecture Framework (DODAF), 601
dependability (software dependability), 26, 285–305
  activities for, 298
  assurance, 353–56, 402–04
  costs of, 290–91
  critical systems, 287, 290, 297, 302
  design considerations, 287, 295
  formal methods and, 299–302, 303
  functionality *v*., 286
  properties, 288–91
  redundancy and diversity, 295–97, 303
  reliability and, 288–90, 297, 303
  safety and, 288, 299
  security and, 22, 26, 288, 376–79
  sociotechnical systems, 291–95, 303
  specification and, 300–02
  system, 268, 286–91, 303
dependable programming guidelines, 325–31
deployment
  component model, 471, 472–73
  design for, 399–400
  service implementation and, 540–41
  system development and, 570
  systems of systems (SoS), 595, 597–99
  UML diagrams, 149, 218
deployment-time configuration, 451–52
derivation history, 750
design (software design), 44, 56–58, 69, 78, 196–225. *See also* architectural design; system design
  activity model (diagram), 56
  configuration management, 212, 215–16, 222
  for deployment, 399–400
  engineering programming and, 23, 44, 58
  guidelines, 396–401, 405
  implementation and, 47, 56–58, 69, 196–225
  interface, 57, 208–09, 222
  life-cycle phase, 47
  models, 123–208, 222
  object-oriented, 198–209, 222
  open-source development, 219–21, 222
  patterns, 209–12
  for recovery, 400–01
  for resilience, 424–32
  reuse and, 57, 212, 213–15
  service interfaces, 533, 536–40
  test-case, 234–37
  UML documentation, 197, 198–209
  user interface, 62
design-time configuration, 451–52
'desk' testing, 428
development
  agile techniques, 77–84, 88, 732
  customization stages, 732–33
  configuration management (CM) phases, 732–33
  engineering design and programming, 23, 44
  evolution and, 23, 60–61, 256–57, 280
  implementation stage, 56–58
  maintenance costs, 274–76, 279
  maintenance *v*., 60–61
  pair programming, 83–84
  plan-driven process, 59–60, 570
  professional software, 19–28
  refactoring, 51, 62, 80–81
  regulators for safety, 353
  reuse and, 52–54
  reuse for (CBSE process), 473, 474–77
  reuse with (CBSE process), 473, 477–80
  safety cases, 362–63
  safety-critical systems, 352–53
  services and, 541–47, 548
  sociotechnical systems, 291–295, 303
  software dependability and, 290
  spiral model for, 256–57
  system processes, 554, 570–74
  testing, 58–60, 81–83, 230–32
development team, 85, 90, 92–93
development testing, 231–42, 252
development view, 174, 192
digital art, 566
digital learning environment (iLearn), 38–39
  application programming interface (API), 39
  architecture (diagram), 38–39

elicitation of requirements, 118–20
layered architecture of, 179
photo sharing story, 118–20
services, 38–39
Virtual Learning Environment (VLE), 38
directed systems, 588
distributed architectures, 171
distributed component systems, 501, 506–09, 517
distributed development (Scrum), 88
distributed systems (software engineering), 490–519
advantages of, 491, 517
architectural design of, 171–72, 182
architectural patterns for, 175–84, 501–12, 517
attack defense, 494–95
client-server architecture, 180–82, 501, 503–06, 517
client-server systems, 499–501, 517
CORBA (Common Object Request Broker Architecture), 493, 507
design issues, 492–96, 517
interaction models, 496–97
middleware, 498–99, 517
openness, 491, 492, 493
quality of service (QoS), 492, 495
scalability, 491, 492, 494, 514, 515–16
software as service (SaS), 512–16, 517
version management of, 735, 737–39
diversity (software diversity)
application types, 24–25
dependability and, 295–97, 303
fault-tolerant architecture, 318, 322, 323–25
redundancy and, 318, 398
reliability and, 318, 322, 323–25, 336
risk reduction and, 398
software engineering, 24–27
documentation, 19, 40, 49, 56, 73–75, 92–93, 273
agile methods and, 73–75, 86, 89–90, 92–93, 126
architectural design and, 175
certification and, 294, 299, 302
change implementation, 260
maintenance and, 92, 273
organization of, 127–28
reader requirements, 103–04
safety cases, 361–67
software requirements (SRS), 126–29, 135
standards, 129, 706
system release, 741, 752–53
TDD and, 244
user requirements, 73, 126–27
domain-specific application systems, 438, 441, 446
duplicate code, 279
dynamic metrics, 720–21
dynamic model, 199, 205, 206, 222
dynamic perspective (RUP), 46
dynamic systems development method (DSDM), 73

E

e-commerce systems, 188–89
early design model, 689–90
Eclipse environment, 32, 216, 218, 219
efficiency, 22, 422–23
effort cost, 669
effort distribution, 272
egoless programming, 83
elaboration phase (RUP), 46
elicit stakeholder requirements, 450
elicitation/analysis for requirements, 55, 112–20, 134
embedded software systems, 25, 32, 634. *See also* real-time systems
architectural patterns and, 620–26, 634
design of, 217–18, 613–20
host-target development and, 217
real-time software engineering, 218, 610–37
simulators for, 217
stimulus/response model, 613–14, 634
timing analysis, 626–31
user testing, 251
emergency call log, 422–23
emergency repair process, 260–61
emergent properties, 558, 559–61, 577
encryption, 413
enduring requirements, 132
engineering, *see* software engineering; systems engineering
Enterprise Java Beans (EJB), 446, 466, 470, 507
Enterprise systems, 422, 552. *See also* ERP systems
entertainment systems, 25
environment assessment (legacy systems), 269
environmental adaptation, 271
environmental control pattern, 620, 623–25
environmental specialization (software product lines), 450

environments. *See also* IDEs
  architectural patterns and, 176
  business requirements changes, 131
  context model for, 142–43
  marketing, 229
  software interaction and system failure, 293–94
  work, 663
equipment layer, 292
equity trading system, 394–95
equivalence partitioning, 235–236
ERP (Enterprise Resource Planning) systems, 21, 184, 438, 442, 454–457
  application frameworks, 446
  architecture of, 455–456
  configurable application reuse, 454–457
  customer adaptation of, 438
  system procurement and adaptation, 569
error-prone constructs, 308, 328–29
error tolerance, 289
errors
  algorithmic, 351–52
  arithmetic, 351
  avoidance and discovery, 300–01
  checking, 359–61
  correction, 48
  failure and fault *v*., 308
  human, 307, 351–52, 418–21
  safety engineering and, 359–61
  specification, 324–25
  static analysis for, 359–61
  system, 307–09
  timing, 238–39
estimation techniques (project planning), 682–86, 696
  algorithmic cost modeling, 683, 684–86
  COCOMO II model, 686–96
  experience-based techniques, 683–84
  software productivity and, 686
ethical/professional responsibility, 28–31, 40
ethnography technique, 116–18
evaluation, prototype phase of, 63
event-driven modeling, 156–58
evolution (software evolution), 69, 255–82
  activity model (diagram), 61
  agile technique and, 261
  business costs and, 274–76, 279
  development *v*., 60, 256–57, 280
  engineer activities for, 20, 23, 44
  legacy systems, 261–70, 280
  life cycle, 257–58, 266
  maintenance, 22, 60–61, 270–79, 280
  processes, 258–61
  program evolution dynamics, 271
  refactoring and, 61, 78, 273, 278–79, 280
  requirements changes, 131
  servicing *v*., 257–58
  software lifetime and, 256–57
  software reengineering, 273, 276–78
  spiral model of, 256–57
  system evolution *v*., 575–76
exceptions
  CBSE for reuse, 476–77
  handlers for, 327–28
Executable UML (xUML), 162
execution time (real-time systems), 627
experience-based estimation, 683–84
experience-based testing, 403
explicitly defined process, 297
exposure, 377, 378, 379
external components, 330–31
external requirements, 109
extreme programming (XP), 73, 77–84
  acceptance testing and, 77, 82
  agile methods and, 73, 77–79
  continuous integration and, 78, 96
  pair programming, 78, 83–84
  release cycle in, 77
  story cards, 79–80
  test first development, 78, 81–83, 242
  user requirements, 73, 99

F

façade pattern, 211
failure propagation, 560–61
failures, *see also* system failure
  definition *v*. judgment, 310
  error and fault *v*., 308
  hardware, 287
  human errors and, 307, 351–52, 418–21
  information loss, 286
  operational, 287
  safe state solutions to, 351–52
  server safety *v*. privacy, 36

software, 18, 22, 26, 287, 308, 310, 340–41, 351–52
system failure costs, 286
fault (system faults), 307–09
avoidance, 308
costs of removal, 308–09
detection and correction, 308
error and failure *v.*, 308
repair, 271
tolerance, 308
fault-tolerant architectures, 318–25, 491
distributed systems, 491
diversity of software, 323–25
N-version programming, 322–23
protection systems, 319–20
self-monitoring, 320–22
fault tree analysis, 349–51
feasibility studies, 54, 104
Federal Aviation Administration, 92, 290
federated systems, 589
film library, client-server architecture for, 182
firewalls, 413–14
flight control software, 296, 321–22, 340, 341
floating-point numbers, 329
formal (mathematical) models, 139
formal methods (software development), 49, 139, 299–302, 303, 356–58
B method, 49
dependability and, 299–302, 303
error avoidance and discovery from, 300–01
mathematical approach, 300, 301
model-checking, 300, 358–59
safety engineering, 356–59
security testing, 404
system models and, 139, 299–301
verification and, 300, 356–58
formal specifications, 109, 300–02
Fortify tool, 404
4 Rs model, 410–11, 414–15, 432
4+1 view model, 173–74
frameworks, 443–46, 600–02, 708–10
Free Software Foundation, 219
frequency (real-time systems), 627
fuel delivery system, 618–19
functional requirements, 105–07, 134, 312, 317–18, 335, 344
functional specialization (software product lines), 450
functionality, 286

G

'Gang of Four,' 209–12
General Public License (GPL), 220
generalization of structural models, 152–53, 205
generator-based reuse, 443
Git system, 216, 737, 740
GitHub, 476, 478
'glue code,' 466, 481, 487
GNU build system, 216
GNU General Public License (GPL), 220
Google Apps, 27
Google Code, 478
governance complexity, SoS, 586–87, 588–90, 606
graphical models, 140
graphical notations, 121
groups, *see* teamwork
growth modeling, 334
guideline-based testing, 234
guidelines
hiring, 661
dependable programming, 325–31
system security, 401–02, 405

H

handlers, exceptions, 327–28
hardware (system), 262
hardware failure, 287, 560–61
hazard-driven approaches, 342, 349–51, 368
hazards, 342, 343, 345–51
analysis of, 345, 349–51
assessment, 345, 346–49
avoidance, 342, 351
damage limitation, 342, 351
detection and removal, 342, 351
fault tree analysis, 349–51
identification of, 345–46
probability, 343
safety-critical system development, 342, 368
severity, 343
heterogeneity, software development and, 24
hierarchical composition, 480
hierarchical groups, 661–62

high-availability systems, 172, 218
honesty (people management), 653
host-target development, 213, 216–18, 222
HTML5 programming, 28, 445
http and https protocols, 530–31
human error, 307, 351–52, 418–21
human needs hierarchy, 653–54

I

IDEs (Interactive Development Environments), 53, 217
  ECLIPSE environment and, 218
  general-purpose, 218
  host-target development and, 216, 217–18, 222
  repository architecture for, 180
iLearn, 38–39, 567. *See also* digital learning environment
implementation (system implementation), 28, 47, 56–58, 69, 196–225
  components, 465, 466, 471–72, 475, 487
  configuration management, 212, 215–16
  design and, 56–58, 69, 196–225
  interface specification, 208–09
  life-cycle phase, 47
  host-target development, 213, 216–18
  open-source development, 219–21
  reuse and, 212, 213–215
  service deployment and, 540–41
  service-oriented software for, 28
  UML documentation, 197, 198–209
  unit testing and, 47
in-car information system, 522–24
inception phase (RUP), 46
inclusion (people management), 653, 657
incompatibility, component composition and, 481–83
incremental delivery, 46, 51, 62, 64–65, 76, 91
incremental development, 46, 50–51, 73–74, 77
incremental testing, 59, 242
incremental integration, 242
incremental planning, 78
information loss, 286
information systems, 32, 185–86, 187–89, 522–24
infrastructure security, 374, 375–76
inheritance, 152, 204, 209, 233, 722. *See also* generalization
input/output mapping, 310–11
inputs, validity checks of, 326–27, 399
inspections, 229–30, 239, 710–714. *See also* reviews
insulin pump control system, 32–34
  activity model of, 33, 155
  data-flow model (DMD) for, 155
  dependability properties for, 288–89
  failure in, 316–17
  functional reliability requirements, 317
  hardware components (diagram), 33
  hazards in, 346
  natural language specification for, 122
  non-functional reliability requirements, 316–17
  permanent software failure, 316
  risk classification for, 347–49
  risk reduction for, 351–52
  safety-critical system control, 341
  safety requirements for, 346–349, 351–52
  safe state, 351
  sequence diagrams for, 155
  software control of, 341
  software failure solutions, 351–52
  structured language specification for, 123–24
  tabular specification for, 124
  transient software failure, 316
issue-tracking systems, 746–47
integrated application systems, 442, 454
integration
  configuration and, 46, 52–54
  continuous, 78, 742–43
  system development and, 570
  system testing and, 48
  systems of systems (SoS), 595, 597–99
integrity, security and, 374, 413
intellectual property rights, 28
interacting workflows, 545–46
interaction models, 144–49, 163, 199–200, 496–97
  distributed systems, 496–97
  object-oriented design and, 199–200
  sequence diagrams, 146–49, 163
  use cases, 144–46, 163, 200
interactive applications, 25
interface design, 57, 208–09
interface misunderstanding, 238
interface misuse, 238

interfaces
application programming interfaces (APIs), 595–96
component, 208–09, 222, 237–239, 465, 468–69, 470–71
model specifications, 470–71
service design for, 533, 536–40, 596
specification, 208–09
systems of systems (SoS), 595–97
unified user interface (UI), 596–97
Internet banking system, 505
interviewing techniques, 115–16
intolerable risks, 347
inversion of control, 445
ISO 9001 standards framework, 708–10, 734
iteration planning, 680
iterative development/delivery, 65, 77, 98. *See also* agile methods
Iterator pattern, 212

J

Java programming language, 82, 152, 161, 197, 208, 218, 219, 327, 330, 359, 444
embedded systems development and, 619–20
interfaces, 208
program testing, 243
real-time systems development and, 619
Java Virtual Machine, 217
JavaMail library, 214
Jenkins, 743
JSON (Javascript Object Notation), 531
J2EE platform, 161, 466
JUnit, 59, 82, 217, 233, 243

L

language processing systems, 186, 189–91, 192
large-scale systems, 556
layered architecture, 177–79, 187–88, 192
layers
legacy systems, 262–64
sociotechnical systems, 292–93, 557–58
legacy systems, 261–70, 280, 540, 576
assessments, 269
business value of, 267–68, 280
component integration, 567
elements of, 262–63
management, 266–70
maintenance of, 263–64, 280
reengineering and, 276, 278
refactoring and, 279
replacement problems, 264–65
system evolution of, 546
wrapping, 278, 442, 540
Lehman's laws, 271
Lesser General Public License, GNU, 220
licensing, 220–21, 356
life cycles
application system reuse problems, 459–60
project planning stages, 668
software evolution, 257–58, 266
software model process, 45, 47–49
lifetimes, system evolution and, 575–76
Linux, 219, 398
logging user actions, 398
logical view, 174, 192
long methods, 279

M

maintainability, 22, 104, 169, 173, 198, 230, 266, 274, 275, 289, 494
maintenance (software maintenance), 22, 270
agile methods and, 90, 92
architectural design and, 172–73, 178
costs, 274–76, 279
development *v.*, 60–61
documentation and, 92, 273
legacy systems, 263–64
life-cycle phase, 48
prediction, 274–76
reengineering, 273, 276–78
refactoring, 278–79
software evolution and, 22, 263–64, 270–79
types of, 271, 280

management (software management), 26, 66–68, 84–88. *See also* configuration management; process improvement; project management; project planning; quality management; version management
  agile methods, 84–88
  automated, 423–24
  CBSE process, 474, 476
  coping with change, 63
  planning, 132–33
  process maturity method and, 66–68
  real-time system processes, 632–34
  requirements change, 130–34
  resilience and, 421–24, 432
management complexity (SoS), 585, 586–87, 587–90, 606
manifesto, agile, 75–76, 77–78
marketing environment, 229
Mars exploration, 358
mathematical specifications, 121. *See also* formal methods
mean time to failures (MTTF), 313, 314
measurement. *See also* metrics
  ambiguity in, 724–25
  component analysis, 722–23
  controller/predictor metrics, 717
  quality management (QM) and, 716–26, 727
  software analysis, 725–26, 727
  software quality, 716–26, 727
mental health care system (Mentcare), 34–36
  administrative reporting, 36
  aggregation association in, 153
  authentication procedures, 416
  class diagrams for, 149–151
  client-server architecture of, 428
  context model of, 141–42
  design risk assessment, 390–91
  dose checking test case, 80
  fail-secure approach, 397
  functional requirements in, 106–07
  generalization hierarchy and, 153
  goals of, 35
  individual care management, 35
  key features of, 35–36
  layered architecture pattern in, 179, 188
  non-functional requirements in, 109–10
  organization (diagram) of, 34
  passwords, 400–101, 416
  patient monitoring, 35
  privacy and, 36
  process model of involuntary detention, 143
  release testing, 246, 247
  requirements-based testing and, 246
  resilience of, 289, 428–30
  safety and, 36
  safety-critical system control, 342
  scenario in, 124–25
  scenario testing and, 247
  security of, 289, 377, 400–01
  sequence diagrams for, 146–49
  sociotechnical system for, 562–63
  story cards and, 79–80
  success criteria for, 562–63
  system boundaries, 141–42
  task cards and, 79–80
  use case modeling and, 145–46
  use cases for, 125–26
merging, 734, 739
message exchange, 496–97, 526–29, 537
message passing interfaces, 238
metrics
  AVAIL, 243–314
  control/predictor, 717
  dynamic, 720–21
  events, 717
  non-functional requirements, 110
  process measurement, 717–20
  probability of failure on demand (POFOD), 313–14, 316
  product, 720–22, 727
  rate of occurrence of failures (ROCOF), 313–314
  reliability, 313–14, 316
  resource utilization, 717
  software measurement and, 716–26, 727
  static, 720–21
  time, 717
Microsoft Office 360, 27
microwave oven scenario, 156–58
middleware, 217, 218, 446, 465, 472–73, 498–99
milestones (projects), 673, 674, 677–78, 696
minimization strategies (risk management), 650–51
mission-critical system, 287
MODAF, 600, 601
model checking, 300, 358–59, 368
model-driven architecture (MDA), 159–62
model-driven engineering (MDE), 158–59, 442

modeling systems, 25, 138–66
models, 45–54, 138–66. *See also* spiral models; UML (Unified modeling Language)
  activity diagrams (UML) for, 33–34, 141, 143–44, 163
  activity stages, 47–48, 142
  agile approach and, 50, 162
  algorithmic cost modeling, 683, 684–86
  application architecture, 185
  behavioral, 154–59, 163
  class diagrams for, 149–50
  COCOMO II, 276, 476, 686–96
  component, 470–73, 487
  context, 141–44, 163, 199–200
  data-driven, 154–55
  dynamic, 199, 205, 206, 222
  event-driven, 156–57
  formal (mathematical), 139, 300
  generalization, 152–53, 205
  incremental development, 46, 49–51
  integration and configuration, 46, 52–54
  interaction, 144–49, 163, 199–200, 496–97
  ISO 9000 standards framework, 708–10, 734
  object-oriented design, 199–200, 204–08
  open-source licensing, 220–21
  processes, 45–54, 68
  project estimation, 682–96, 696
  quality management (QM) and, 709–10, 719
  real-time system design, 617–19
  reliability growth, 334
  RUP (Rational Unified Process), 46–47
  reuse-based development, 52–54
  sequence, 144, 146–49, 155, 163, 205, 206–07
  spiral, 63, 256–57
  state machine, 205, 207–08, 222, 617–18, 634
  state-based, 156–158, 163
  static, 205, 222
  stimulus/response, 613–14, 634
  structural, 149–54, 163, 199, 205
  subsystem, 205–06
  'Swiss cheese,' 420–21
  of testing process, 230–31
  UML (Unified Modeling Language), 33–34, 139, 140–41, 144–49, 713
  use case, 125–26, 141, 144–46, 163, 200–01
model-view-controller (MVC) pattern, 176–77, 179, 444
monitoring projects, 651–52, 673
motivation (people management), 653–56
multi-tenancy, 514, 515, 516
multi-tier client-server architecture, 501, 505–06
MySQL, 219, 445

N

N-version programming, 322–23
namespaces, 528–29
natural language requirements, 121–22
nested technical and sociotechnical systems, 416–17
.NET framework, 161, 443, 446, 466, 470–71, 478, 507
non-deterministic properties, 561–62
non-functional requirements, 105, 107–11, 134, 169, 172–73, 312, 314–18, 547

O

object and function reuse, 438
object classes, 149–50, 202–04, 470
object constraint language (OCL), 208, 484–85
object level (reuse), 214
Object Management Group (OMG), 159
object-oriented metrics, 721–22
object-oriented systems
  architectural design and, 201–02
  class diagrams for, 149–50
  class identification, 202–04
  design, 198–209, 222
  frameworks in, 444
  interface specification, 208–09
  system (design) models, 204–08
  Unified Modeling Language (UML) and, 140, 198–209
  use case model, 200–01
Objectory method, 125
observe and react pattern, 620, 621–23
Observer pattern, 210–11
on-site customer, 78
openness, distributed software, 491, 492, 493
open-source development, 219–21, 222, 738–39

operating system layer, 292
operating systems (real-time), 631–34, 635
operation and maintenance, 48
operation incompatibility, 481
operation incompleteness, 481
operation stage (systems), 554
operational failure, 287
operational processes, 421–24, 432
operational profiles, 334–35
operational security, 374, 376
operator reliability, 287, 560–61
Oracle, 21, 219
organizational design patterns, 175
organizational layers, 292, 557
organizational requirements, 108–09
organizational systems, 589
organizations and security, 380–82
overhead costs, 669
overspecification of reliability, 315

P

packing robot control system, 168
pair programming, 78, 83–84, 715
parameter definition, 452
parameter incompatibility, 481
parameter interfaces, 237
partition testing, 234–36
partner company software systems, 49
password checker, 392
passwords, 400–01, 413, 414, 416
path testing, 237
patient records system (PRS), 148–49
patterns, 175–84, 209–12, 442, 444
  application frameworks and, 444
  architectural, 175–84
  design, 209–12, 442
payment models, 547
peer-to-peer (p2p) architecture, 501, 509–12, 517
penetration testing, 403–04
People Capability Maturity Model (P-CMM), 656
people management, 652–56, 664
performance, 172, 248
periodic stimuli, 613
photo library, 483–85
physical view, 174, 192
pipe and filter architecture, 182–84, 191
plan-driven process, 45, 47, 50, 73, 570
  agile methods *v.*, 45, 74–75, 91–93, 98
  changing environment and, 73
  incremental development and, 50
  model processes, 47, 50
  project planning and, 672–75, 696
  scheduling and, 675–76
  system development and, 570
  testing (validation) phases, 59–60
  waterfall model, 47–48
planning game, 681–82
planning. *See also* project planning
  incremental, 78
  requirements management, 132–33
  risk, 650–51
  Scrum product backlog, 85, 86, 98
  test, 231
platform-independent model (PIM), 159–61
platform-level protection, 393–394
platform services, 472
platform specialization (software product lines), 450
platform-specific models (PSM), 160–61
plug-in architecture, 218
pointers, 308, 329
post-architectural level, 692–94
power supply failure, 627–28
practice perspective (RUP), 46
prediction, maintenance and, 274–76
predictor metrics, 717
PRISM model checker, 358
probability of failure on demand (POFOD), 313–14, 316
probability values, hazards, 343
problem tracking, 216
procedural interfaces, 238
process (software processes), 23, 26, 43–71
  activities, 40, 54–61
  agile approach, 45, 66
  analysis, 67, 112–20, 626–31
  assurance, 353–56
  design and implementation, 56–58
  emergency repair, 260–61
  engineer activities for, 20, 23, 44, 54–61
  evolution, 44, 60–61, 258–61
  improvement of, 65–68
  life cycles, 45, 47–49

management, 421–24, 432
maturity approach, 66–68
measurement, 66–67, 717–20
models, 45–54, 68
operational, 421–24, 432
plan-driven, 47–48
professional, 19–28, 45
prototype development, 62–63
quality (process-based), 65–68, 705
quality metrics, 717–20
review phases, 711–13
RUP (Rational Unified Process), 46–47
specification, 44, 54–56
standards, 45, 707, 708
validation, 44, 58–60
process change, 45, 69
agile manifesto and, 75–76
CBSE, 473–480
coping with, 61–65
evolution, 258–61
implementation, 259–60
for safety assurance, 353–56
software processes, 61–65, 67
urgent changes, 260
process improvement, 69
agile approach, 66
business values, 267–68
legacy system management, 266–70
process maturity approach, 66–68
reengineering, 276–78
refactoring, 278–79
software quality and, 65–68
software evolution and, 266–70, 276–79
process management, real-time systems, 632–34
process maturity approach, 66–68
process pipeline pattern, 620, 625–26
process requirements, 317
process specialization (software product lines), 450
process view, 174, 192
procurement (acquisition), 473, 553–54, 566–70, 577
producer/consumer pattern, 202
producer/consumer processes (circular buffer), 616–17
product
instance development, 450
quality metrics, 720–22, 727
requirements, 108–09
software types, 20–21, 24–26
standards, 706, 707
product architects (Scrum), 96
product backlog (Scrum), 85, 86
product owner (Scrum), 85
product risk management, 644–75, 646
professional software development, *see* development
program evolution dynamics, 271
program generators, 442
program inspections, 229–30, 239, 713–14. *See also* reviews
program libraries, 442
program modularization, 277
program structure improvement, 277
programmer/tester pairs, 231–32
programming. *See also* extreme programming
dependable guidelines, 325–31
egoless, 83
engineering design and, 23, 44, 58
real-time systems, 619–20
secure system guidelines, 401–02
techniques/activities, 26, 54–56
project management, 84–88, 641–66
activities, 643–44
agile methods and, 84–88, 643, 647, 661
differences from engineering, 642–43, 664
motivation and, 653–56
relationships with people, 652–56, 664
risk management, 644–52, 664
teamwork, 656–64
project planning, 92–93, 667–99
agile methods and, 670, 680–83, 696
bidding, 669, 671–72
COCOMO II cost modeling, 686–96
development team effectiveness, 92–93
duration and staffing, 694–96
estimation techniques, 682–86, 696
life cycle stages of, 668
milestones, 673, 674, 677–78, 696
plan-driven development and, 672–75, 696
process, 673–75
project costs, 669, 696
scaling agile methods for, 91–93
scheduling and, 675–80, 696
software pricing for, 670–72, 696
supplements, 673
user stories for, 681–82
project risk management, 644–45
Promela, 358

protection, 383
  assets, 380, 384, 390
  cybersecurity, 376, 414
  fault-tolerant architecture, 319–20
  layered architecture design, 393–95
  systems, 319–20, 414
prototyping (system prototyping), 62–63, 69, 117, 130
Python, 190, 197, 198, 327, 444

Q

quality management (QM), 299, 700–29
  agile development and, 714–16, 727
  configuration management (CM) and, 733
  documentation standards, 706
  reviews and inspections, 710–14, 727
  software development and, 701–02
  software measurement/metrics and, 716–26, 727
  software quality and, 703–05, 727
  software standards and, 706–10, 727
quality of service (QoS), 492, 495
quantitative reliability specifications, 314–15

R

range checks, 326
rapid software development, 73–74
rate of occurrence of failure (ROCOF), 313–14
reactive systems, 612
realism checks, 129
real-time systems, 205, 218, 610–37
  architectural patterns for, 620–26, 634
  design, 205, 613–20
  embedded systems, 218, 610–637
  modeling, 617–19, 634
  operating systems, 631–34, 635
  process management, 632–34
  programming, 619–20
  responsiveness, 611–12
  software engineering for, 610–37
  stimulus/response model, 613–14, 634
  timing analysis, 626–31, 635
reasonableness checks, 327
recognition, 410, 411, 414–15, 432
record-level protection, 393–394
recovery
  database integrity checking and, 430
  design for, 400–01
  requirements, 317
  resilience and, 411, 414–15, 430, 432
reductionism of complex systems, 590–93, 606
redundancy
  dependability and, 295–97, 303
  diversity and, 318, 398
  requirements, 317
reengineering (software reengineering), 273, 276–78, 280
refactoring, 51, 62, 78, 80–81, 83–84, 168, 278–79
  agile methods, 51, 80–81
  architectural design and, 168
  extreme programming (XP) methods, 78
  maintenance and, 278–79
  pair programming, 83–84
  software evolution, 273, 278–79, 280
reference architectures, 191
refinement-based development, 300
regression testing, 244
regulation and compliance (software), 294–95, 353
regulators, 294–95, 361, 362, 368
reinstatement, 411, 414–15, 432
release alignment (Scrum), 96
release management, 216, 731, 750–53, 754
release testing, 245–48
reliability, 309
  availability and, 309–12
  dependability and, 288–90, 297, 303, 336
  diversity and, 318, 322, 323–25, 336
  emergent properties, 560–61
  failure and, 18, 307–12, 560–61
  fault-tolerant architectures, 318–25
  functional requirements, 312, 317–18, 335
  growth modeling, 334
  human error, 307
  measurement of, 331–35
  metrics, 312–13, 332, 335
  non-functional requirements, 312, 314–18
  operational profiles, 334–35
  overspecification of, 315
  programming guidelines, 325–31
  requirements, 312–18, 335
  safety and, 340–41
  security and, 379

sociotechnical systems, 560–61
software, 18, 560–61
specification, 314–18
system error, 307–09
system fault, 307–09
systems, 18, 19, 22, 288–90, 297, 303, 306–38
statistical testing, 332–33, 336
remote method invocations (RMIs), 497
remote procedure calls (RPCs), 470, 471, 497
repairability, 289
repeatable process, 297, 303
replicated servers, 318
repository architectural pattern, 179–80, 190
repository cloning, 737–38
representation checks, 327
requirements, 102, 134
agile methods and, 55, 131–32
analysis and definition (life-cycle phase), 47
availability, 218
business changes, 131, 135
classification and organization of, 113
components, 218
discovery and understanding, 113, 115–18
documents (software specification), 103–04, 111, 114, 126–29, 135
elicitation and analysis of, 55, 112–20, 134
enduring, 132
engineering understanding of, 20, 23, 26
evolution, 131
functional, 105–07, 134, 317–18
hazard-based, 345
identification, 132
management, 132–134, 135
non-functional, 105, 107–11, 134, 314–17
notations for writing, 121
prioritization and negotiation of, 113
refinement, 53
reliability and, 312–18
reviews, 130
risk-based, 344, 345
safety, 344–52
specification, 55, 69, 102–03, 106–07, 110, 120–29, 135, 314–18, 344, 345
spiral model for, 572
software process, 44, 54–56
storage, 132
system, 102–03, 120–21
testing (requirements-based), 245–46
traceability, 132, 133
user, 102–03
validation, 55, 129–30, 135
volatile, 132
requirements engineering (RE), 69, 101–37
change management, 111, 130–34
documents for, 103–05
elicitation/analysis process, 112–20, 134
ethnography technique for, 116–18
feasibility studies, 54, 104
interviewing techniques for, 115–16
processes, 111–12, 134
software process activities, 44, 54–56
software documentation (SRS) for, 126–27
spiral model for, 112
system development and, 570
requirements partitioning, 571
research management systems, 448–49
resilience (system resilience), 288, 409, 408–34
activities, 410–11
automated management, 423–24
cybersecurity, 412–16, 432
dependability and, 288, 289
design for, 424–32
efficiency and, 422–23
engineering, 408–34
4 Rs model, 410–11, 414–15, 432
human error and, 418–21
interrelated business approach, 426–27
management, 421–24, 432
operational processes, 421–24, 432
security and, 288, 379
sociotechnical systems, 416–24
survivable systems analysis, 425–26
system failure and, 410–12
testing, 427–428
resistance, 410–11, 414–15, 432
resource management systems, 188–89, 192
resource sharing, 491
respect (people management), 652
restart capabilities, 329–330
restaurant interactions, 496–97
RESTful services, 524, 529–33, 544
reuse (software reuse), 26, 28, 46, 52–54, 169, 209–10, 212, 213–15, 222, 437–63, 474–480
application frameworks, 442, 443–46, 460
application system, 438, 453–60
approaches supporting, 441–43

reuse (*continued*)
architectural design and, 169
CBSE for, 473, 474–77
CBSE with, 473, 477–80
component selection and design, 57
components, 52–53, 212, 214, 221, 438–439, 452, 468, 487
costs of, 214, 439
design patterns, 209–10, 212, 442, 444
engineering applications of, 26, 28
generator-based, 443
implementation and, 212, 213–15
integration and configuration of, 52–54
integration problems, 459–60
landscape, 340–443
levels of, 213–14
object and function, 438
process model for, 52–53
software development tools, 53
software product lines, 442, 446–52
system features and, 46
reuse-based software engineering, 53–54, 438
reuse model, 690–92
reverse engineering, 277
reverse planning, 680
reviews, 130, 229, 239, 710–14
checklists, 713–14
code, 83, 715
hazard register for, 355
inspections and, 229, 710–14, 727
program inspections, 713–14
quality management (QM), 710–14, 727
requirements validation, 130
review process, 711–13
safety, 354, 355
verification and validation using, 229
rework, 49, 56, 61, 73, 75, 84, 129
risk
acceptable, 347–48
accidents (mishaps) and, 343–44, 347
analysis, 362, 648–49
as low as reasonably practical (ALARP), 347
defined, 343
redundancy and diversion for, 398
indicators, 652
intolerable, 347
ranking types of, 649
reduction, 351–52, 398
security assessment, 381–82, 405
triangle, 347–48
risk management, 644–52, 664
identification of risk, 647–48
planning process, 650–51
processes, 645–47
product risks, 644–45
project risks, 644–45
risk analysis and, 648–49
risk monitoring, 651–52
strategies for, 650–81
risk-based requirements specification, 344, 345
robot control system, 168
role replication (Scrum), 96
Ruby, 190, 444
RUP (Rational Unified Process), 46–47

S

safety, 339–72, 379
architectural design and, 172
assurance processes, 353–56
costs and, 357, 362–63
dependability and, 288, 299
engineering processes, 352–61
ethics and, 30–31
formal verification, 356–58
functional requirements, 344
hazard-driven requirements, 345, 368
hazards and, 342, 343, 345–351
model checking, 358–59, 368
regulation and compliance for, 294–95
regulators, 294–95, 361, 362
reliability and, 340–41
requirements, 344–52, 362
risks and, 343, 343–44, 347–48, 351–52
software certification, 355–56
static program analysis, 359–61, 368
terminology, 343
safety cases, 361–67, 368
development of, 362–63
organization of, 361–62
regulators for, 361, 362, 368
software safety arguments, 364–67
structured arguments, 363–364

safety-critical systems, 287, 340–44, 368
certification of, 294, 302, 355–56
control systems, 341–42
dependability and, 294, 302
development process and, 352–53
error-prone constructs and, 329
hazard-driven techniques, 342
primary safety-critical software, 341
process assurance and, 355–56
regulation and compliance for, 294, 353
risk triangle for, 347–48
secondary safety-critical software, 341–42
system failure and, 340–41
safety reviews, 355
SAP, 21
Sarbanes Oxley accounting regulations, 51
scalability, 491, 492, 494, 514, 515–16
scale, software development and, 24
scaling agile methods, 88–97, 98
scenarios
elicitation of requirements from, 118–20
testing, 246–47, 252
use cases, 125–26
scheduling, 675–80, 696
activity charts for, 678–80
project planning and, 675–80, 696
plan-driven projects, 675–76
presentation (visualizing), 676–80
Scrum, 73, 78, 85–88, 96, 98
secure systems, 561
security, 24, 26, 373–407
application, 374–375
architectural design and, 172, 388, 392–95
assurance, 402–04
availability, 374, 375, 379
checklist, 403
confidentiality, 374
controls, 377, 378–79
dependability and, 22, 26, 288, 376–79
design for, 374, 388–402, 405
engineering, 373–407
failure, 397
guidelines, 396–401, 404
infrastructure, 374, 375–76
logging user actions, 398
operational, 374, 376
organizations and, 380–82
policies, 396–97
programming guidelines, 401–02
protection, 380, 384, 390, 393–94, 395
regulation and compliance for, 294–95
reliability and, 379
requirements, 382–88
resilience and, 288, 379
risk assessment, 381–82, 405
safety and, 379
system layers, 374–75
terminology, 377–378
testing, 402–04
threats, 377, 378, 404
trust and, 22, 24
usability guideline, 397–98
validation, 405
vulnerability and, 377, 378, 391, 401
self-monitoring architecture, 320–22
SEMAT (software engineering methods and tools) initiative, 24
semicentralized P2P architecture, 511, 512
sensor-based data collection systems, 32
separation of concerns, 486
sequence diagrams, 141, 144, 146–49, 155, 163, 205, 206–07, 241
sequential composition, 480
server overload, 512–13
service engineering, 533–41
candidate identification, 533–36
implementation and deployment, 540–41
interface design, 533, 536–40
legacy systems and, 540
service information exchange (SOAP), 525–26, 531, 544
service-oriented architectures (SOAs), 513–14, 520–50
approach, 522, 524
components, 526–29
message exchange, 526–29
service interface, 528
service protocols, 525
software as service (SaS) *v.*, 513–14, 522
standards, 525–26
web applications, 524–29
WSDL and, 526, 527–29
service-oriented software engineering, *see* service engineering; service-oriented architectures (SOAs); services
service-oriented systems, 442, 466–67, 526–33

service-to-service communication, *see* integrated services
services, 521
  business, 534, 541–47, 548
  classification of, 534, 548
  communication and, 524–29
  components, 521, 526–29
  composition (construction) of, 541–47
  coordination, 534, 548
  incremental delivery and, 64–65
  operation and maintenance for, 48
  process models for, 544–46
  reusable Web components, 52, 526–29
  reuse of, 542
  software development and, 541–47, 548
  testing, 543, 546–47
  utility, 534, 548
  web-based, 27–28, 521
  RESTful approach, 524, 529–33, 544
  service information exchange (SOAP), 525–26, 531, 544
  workflow, 542, 543, 544–46, 548
servicing, evolution *v*., 257–58
shared memory interfaces, 238
signatures, 744–45
simple design, 78
simplicity (agile methods), 76, 78, 91
simulation systems, 25
simulators, 217
size checks, 327
SLAM model checker, 358
small releases, 78
social change, business and, 24
social layer, 292
sociotechnical systems, 552, 577
  complexity of, 556, 558–59
  defensive layers, 419–20
  emergent properties 544, 559–61, 577
  environment and software interaction, 293–94
  failure propagation, 560–61
  human error and, 418–21
  layers of, 292–93, 557
  management, 421–24, 432
  nested technical systems, 416–17
  non-deterministic properties, 561–62
  operational processes, 421–24, 432
  organizational elements, 557–58
  regulation and compliance, 294–95
  resilience and, 416–24
  success criteria, 562–63
  systems engineering for, 556–59
software, 19, 20, 228
  attributes, 20, 22
  customized (bespoke), 21
  efficiency, 22
  engineering ethics, 28–31
  failures, 18
  generic products, 20–21
  issues affecting, 24
  lifetime, 256–57
  product types, 20–21, 24–26
  professional development, 19–28
  regulation and compliance of, 294–95
  system boundaries and characteristics, 26
software architecture catalog, Booch's, 170
software as service (SaS), 512–16, 517
  configuration of, 514–15
  multi-tenancy, 514, 515, 516
  scalability, 514, 515–16
  server overload and, 512–13
  service-oriented architectures (SOAs) *v*., 513–14, 522
'software crisis', 19
Software Development Life Cycle (SDLC) model, 45
software development tools, 53
software diversity, 318, 322, 323–25, 336
software engineering, 19–23, 40, 92
  activities for software process, 20, 23, 44
  computer science *v*., 20, 23
  diversity, 24–27
  engineering discipline, 21–22
  ethical responsibility and, 28–31, 40
  formal verification, 356–58
  fundamental notions in, 26, 40
  Internet effect on, 20, 27–28
  licensing for, 356
  model checking, 358–59, 368
  model-driven engineering (MDE), 158–59
  product development and, 20–21
  reuse-based, 53–54, 438
  safety processes, 352–61
  static program analysis, 359–61, 368
  systems engineering *v*., 20, 23, 40, 554
  web-based systems, 27–28
Software Engineering Institute (SEI), 67

software measurement/metrics, 716–26, 727
software platform, 57
software pricing, 670–72, 696
software product lines, 442, 446–52
software quality attributes, 704
software requirements specification (SRS), 126–29
software safety arguments, 364–67
source code translation, 277
SourceForge, 476, 478
space shuttle (U.S.) system, 319
specialization, software product lines, 450
specifications (software specifications), 20, 54–56, 208–09, 300–02
  availability, 313
  engineering definition and constraints, 23
  functional requirements, 106–07
  graphical notations, 121
  dependability and, 300–02
  design interface, 208–09
  errors, 324–25
  formal techniques, 300–02
  hazard-driven safety requirements, 345
  management of, 26
  natural language requirements, 121–22
  non-functional requirements, 110
  problem analysis and, 133
  reliability metrics, 313–14
  risk-based requirements, 344, 345
  safety requirements and, 344–45
  software process, 44, 54–56
  SRS document, 126–29
  structured natural language requirements, 121, 122–24
  system failure and, 310
  system requirements, 102–03, 120–29, 135
  use cases, 125–26
  user requirements, 102–03, 120, 135
speculative generosity, 279
SPIN model checker, 358
spiral models, 48, 112, 256–57, 572
sprint (Scrum), 85, 86–87
SQL (Structured Query Language), 218, 399, 401, 445, 505
stable domain abstractions, 475
staff allocation charts, 678, 680
stakeholders, 103–04, 107, 112–16
stand-alone applications, 25
standards
  documentation, 706
  ISO 9000 standards framework, 708–10, 734
  process, 45, 707, 708, 734
  product, 706, 707
  quality management (QM) and, 706–10, 727
  software, 706–10, 727
  service-oriented architectures (SOAs), 524, 525–26
  value of, 707–08
  web service, 525–26
state diagrams (UML), 141, 163, 205, 207–08
state machine models, 205, 207–08, 222, 617–18
state-based modeling, 156–58
static analyzers, 217
static metrics, 720–21
static models, 143, 205, 222
static perspective (RUP), 46
static program analysis, 359–61, 368
statistical testing, 332–33, 336
stimulus/response (embedded systems) model, 613–14, 634
storage management, 132, 740
stories, elicitation of requirements from, 118–20
story cards, 79–80, 99. *See also* user stories
stress testing, 248
structural models, 149–54, 163, 199, 205
structured arguments, 363–64
structured natural language requirements, 121, 122–24
subsystem engineering, 571, 573
subsystem faults, 573
subsystem model, 205–06
Subversion system, 216, 735
support environment, 32
support services, 472
support software, 262
survivable systems analysis, 425–26
sustainable pace, 78
'Swiss cheese' model, 420–21
switch (case) statements, 279
system availability, *see* availability
system boundaries, 141–42, 163, 199, 556–57
system building, 731, 740–45, 753
system construction by composition, 543–44
system design
  actuator control processes, 613–14, 615
  embedded systems, 217–18, 613–20

system design (*continued*)
host-target development, 213, 216–18, 222
modeling, 617–19
producer/consumer processes, 616–17
programming, 619–20
real-time systems, 205, 613–20
risk assessment, 389–92
security systems, 388–402, 405
stimulus response model, 613–14
system error, 307–09
system failure, 307
acceptance of, 410
availability and, 309–12
costs of, 286
critical systems, 287, 290, 297, 302, 340–41
dependability and, 22, 268, 286–91, 303
error and fault *v*., 308
hardware failure and, 287
human errors and, 287, 351–52
nondeterminism and, 560–61
reliability and, 307–12, 560–61
reparability and, 289
resilience and, 410–12, 420–21
safety-critical systems, 340–41
security and, 22, 268, 397
sociotechnical, 560–61
software failures and, 287, 340–41
specifications and, 310
'Swiss cheese' model of, 420–21
types of, 287
system fault, 307–09
system infrastructure frameworks, 446
system integration, 215–16
system level (reuse), 214
system modeling, *see* models
system of system coalitions, 589
system output, 268
system requirements, 52, 102–03
system reuse, 438
system selection, 594–95
system testing, 48, 59, 231–32, 240–42
system versions, 323–325
system vision document, 565–66
systems (software systems). *See also* distributed systems; embedded software systems; systems of systems (SoS)
activity models (diagram), 60, 61
agile methods for, 93–96
analysis for architectural design, 169
case study types, 31–32
complexity of, 18, 93–96, 274–75, 278, 552–53, 558–59
cost effectiveness of, 22–23
dependability, 268, 286–91, 303
engineering fundamentals for, 26, 40
large-scale, 93–94, 556
modeling, 25, 138–166
sociotechnical, 291–95, 303, 556–63
software design and, 47
specification requirements, 120–29
state representation, 155
systems of systems (SoS) *v*., 581–82
types of, 18, 20–21, 24–26, 32, 40, 552
systems engineering, 20, 23, 40, 551–79
conceptual design, 553, 563–66, 577
development processes, 570–74, 577
enterprise systems, 552
lifetimes and, 575–76
range of disciplines, 554–55
sociotechnical systems, 552, 556–63, 577
software engineering *v*., 20, 23, 40, 554
spiral model for requirements, 572
stages of, 553–54
system evolution, 575–76
system procurement (acquisition), 453–54, 566–70, 577
technical computer-based systems, 552
systems of systems (SoS), 25, 256, 442, 556, 580–609
architectural design, 595, 599–606, 607
classification of systems, 587–90, 606
container systems, 603–05
data-feed systems, 602–03
deployment and integration of, 595, 597–99
engineering, 593–99
governance complexity, 586–87, 588–90, 606
interface development, 595–97
large-scale systems, 556
management complexity, 585, 586–87, 587–90, 606
reductionism, 590–93, 606
software systems, 582
system complexity, 584–87, 606
system *v*., 581–82
technical complexity, 585, 586–87, 590
trading systems, 605–106

T

tabular specification, 124
task cards, 79–80, 82. *See also* user stories
teamwork, 656–64
  development team, 85, 90, 92–93
  group cohesion, 658
  group communication, 662–64
  group member selection, 659–60
  group organization, 660–62
  hierarchical groups, 661–62
  hiring people, 661
  physical work environment and, 663
technical complexity, SoS, 585, 586–87, 590
technical computer-based systems, 552
test cases, 130, 234–37, 252
test-driven development (TDD), 242–45
test-first development, 59, 78, 81–83, 252
test planning, 231
testing (software testing), 58–60, 226–54, 402–04, 427–28
  acceptance, 77, 82, 249, 250–51, 252
  agile methods for, 59, 78, 81–83, 251
  alpha, 249
  assurance and, 402–04
  automated, 78, 81–83, 233–34, 242, 252
  beta, 58, 60, 249–250
  choosing test cases, 234–37, 252
  component testing, 59, 232, 237–39
  customer, 58, 59
  debugging *v*., 58, 232, 244
  defect, 58, 227–28, 232, 245, 248
  development and, 59–60, 81–83, 570
  development testing, 231–42, 252
  goals of, 227
  incremental approach, 59
  inspections *v*., 229–30
  model of, 230–31
  penetration, 403–04
  plan-driven phases, 59–60
  process, 58–60
  release testing, 245–48
  reliability and, 332–33, 336
  resilience, 427–428
  security, 402–04
  services, 543, 546–47
  stages in, 59, 231
  statistical, 332–33, 336
  system, 59, 232, 240–42
  test-driven development (TDD), 242–45
  tool-based analysis, 404
  unit testing, 47, 232–37
  user testing, 249–51
  validation, 58–60, 227–29
threats, 377, 378, 404, 413, 414–15
timeouts, 330–31
timestamps, 744
timing analysis, 626–31, 635
timing errors, 238–39
TOGAF, 600, 601
tool-based analysis, 404
tool support, 132, 743, 744, 746
traceability (requirements), 132, 133
trading systems, 605–06
transaction-based applications, 25
transaction processing systems, 185, 186–87, 192
transition phase (RUP), 46–47
triple modular redundancy (TMR), 322
trust, security and, 22, 24
two-tier client-server architecture, 501, 503–05

U

UML (Unified Modeling Language), 140
  activity diagrams, 33–34, 141, 143–44
  architectural design and, 139, 175, 205
  behavioral models, 155–57
  business processes and, 143–44
  class diagrams, 141, 149–50
  component interface diagram, 469
  deployment diagrams, 149, 218
  diagram types, 139, 140–41, 205
  event-driven, 156–57
  executable (xUML), 162
  generalization and, 152
  interaction models, 144–49
  object oriented metrics and, 721
  object-oriented systems and, 140, 198–209
  package symbol, 37
  sequence diagrams, 141, 146–49, 155, 163, 205, 206–07
  state diagrams, 141, 205, 207–08

UML (*continued*)
subsystem models, 205–06
system modeling using, 139, 140
use cases, 125–26, 141, 144–46, 163, 205
workflow models, 143–44, 544
unified user interface (UI), 596–97
Uniform Resource Locator (URL), 530–32, 539
unit testing, 47, 231, 232–37
Universal Description, Discovery, and Integration (UDDI), 526
Universal Resource Identifiers (URIs), 471, 527
Unix systems, 183, 401
urgent changes, 260
usability
error tolerance, 289
patterns, 175
requirements, 109–10
security guideline, 397–98
usage, component models and, 471
use cases, 125–26, 141, 144–46
interaction models, 144–46, 163, 200–01
requirements specification and, 125–26
testing, 240–41
UML diagram models, 141
user access, 392
user actions, logging, 398
user-defined error checking, 360
user expectations, 228–29
user interface design, 62
user requirements, 55, 73–74, 102–03
user stories, 79–80, 82, 86, 247, 681–82
conceptual design and, 565–66
project planning (agile method) with, 681–82
task cards, 79–80
user testing, 249–51
utility services, 534, 548

V

V & V (verification and validation), 58, 227–29, 356. *See also* testing; validation
V-model, 60
vacation package workflow, 542, 544–45
validation (software validation), 20, 69, 58–60
engineering activities for, 23, 44
requirements, 55, 129–30, 135
testing, 58–60, 227–29
verification *v*., 227–29
validity checks, 129, 326–27, 399
vehicle dispatcher system, 448–49
velocity (Scrum), 85
verifiability, 129
verification (software verification)
cost effectiveness of, 357
formal methods and, 300, 356–59
goal of, 228
levels of confidence, 228–29
model checking, 300, 358–59
safety engineering, 356–59
validation *v*., 227–29
version control (VC) systems, 731, 735, 753
version management (VM), 215, 216, 731, 735–40, 753
vertical software packages, 20
views, architectural, 173–175, 192
Virtual Learning Environment (VLE), 38
virtual systems, 588
visibility of information, 325–26
volatile requirements, 132
VOLERE requirements engineering method, 123–24
vulnerability, 377, 378, 391, 401, 402

W

waterfall model, 45, 47–49
weather information database, 531–32
weather stations, *see* wilderness weather stations
web application frameworks (WAFs), 444
web-based systems, 27–28
web services, 27, 52, 521, 524–33. *See also* services; WSDL
browser development, 27, 521
business process model and, 544–46, 548
business, 534, 541–47, 548
classification of, 534, 548
clouds, 27, 532
components for, 526–29
composition (construction) of, 541–47
coordination, 534, 548
defined, 27, 521
http and https protocols, 530–31
interactive transaction-based applications, 25

interfaces, 28, 528
resource operations, 530
RESTful approach and, 529–33, 544
reusable components as, 52, 526–29, 542
service-oriented architecture (SOA) and, 524–29
SOA approach, 524
software development and, 541–47, 548
standards, 525–26
testing, 543, 546–47
utility, 534, 548
WSDL interface, 528
'wicked' problems, 130–31, 286, 301
wilderness weather stations, 36–38
architectural design of, 201–02
availability and reliability of, 289
'collect weather data' sequence chart for, 241
context model for, 199
data collection (sequence diagram) in, 206
data collection system architecture in, 202
data management and archiving system, 36
environment of, 36–37
high-level architecture of, 201
interface specification, 208–09
object class identification, 202–04
object interface of, 233
objects, 203–04
sequence diagram for, 241
sociotechnical system of, 291–92
state diagram, 207–08
station maintenance system, 37
system testing, 240–41
use case model for, 200–01
work environments, 663
work flow representation (UML), 143–44
workflow, 83, 452, 542, 543, 544–46, 548
wrapping, legacy system, 278, 442, 540
WS-BPEL, 525, 526, 544, 546
WSDL (Web Service Definition Language), 526, 527–29, 537, 540, 544
message exchange, 527–29, 537
model elements, 527–28
service deployment and, 540
web service interface, 528

X

XML, 470, 525, 527–529
language processing, 186, 189, 191, 470, 544
namespaces, 528–29
service descriptions, 528–29
web services and, 525
WS-BPEL workflow models, 544, 546
WSDL message exchange, 527–29
XML-based protocols, 521



Author Index

A

Abbott, R., 202, 224
Abdelshafi, I., 87, 100
Abrial, J. R., 49, 71, 300, 304, 357, 370
Abts, C., 459, 460, 462, 594, 608, 684, 688, 691, 694, 699
Addy, E., 476, 489
Aiello, B., 731, 754, 755
Alexander, C., 209, 224
Alford, M., 552, 579
Ali Babar, M., 169, 194
Allen, R., 459, 460, 463
Ambler, S. W., 89, 95, 98, 99, 140, 162, 165
Ambrosio, A. M., 341, 372
Amelot, A., 300, 304
Anderson, E. A., 300, 305
Anderson, R. J., 495
Anderson, R., 376, 402, 405, 406
Andrea, J., 244, 254
Andres, C., 98, 680, 699
Appleton, B., 175, 194, 754
Arbon, J., 252
Arisholm, E., 84, 99
Armour, P., 696
Arnold, S., 552, 579
Ash, D., 275, 282
Atlee, J. M., 135
Avizienis, A. A., 286, 303, 304, 323, 338

B

Badeau, F., 300, 304
Balcer, M. J., 162, 165
Ball, T., 300, 305, 358, 361, 370
Bamford, R., 709, 729, 734, 755
Banker, R. D., 275, 282
Basili, V. R., 73, 100
Bass, B. M., 655, 666
Bass, L., 169, 170, 175, 192, 194
Baumer, D., 446, 462
Baxter, G., 559, 579
Bayersdorfer, M., 221, 224
Beck, K., 71, 77, 80, 98, 99, 100, 203, 224, 242, 254, 279, 282, 680, 699
Beedle, M., 71, 85, 100
Behm, P., 356, 371
Belady, L., 271
Bell, R., 347
Bellouiti, S., 87, 100
Bennett, K. H., 257, 282
Benoit, P., 356, 371
Bentley, R., 125, 137
Berczuk, S. P., 175, 194, 754
Bernstein, A. J., 186, 195
Bernstein, P. A., 498, 519
Berry, G., 612, 637
Bezier, B., 235, 254
Bicarregui, J., 300, 302, 303, 305
Bird, J., 280

Bird, J., 90, 100
Bishop, P., 361, 371
Bjorke, P., 563, 579
Blair, G., 491, 517, 519
Bloomfield, R. E., 361, 371
Bochot, T., 300, 305, 358, 371
Boehm, B. W., 40, 45, 48, 71, 98, 227–28, 254, 459, 460, 462, 594, 608, 649, 666, 683, 684, 687, 688, 691, 694, 695, 697, 699
Bollella, G., 619, 637
Booch, G., 140, 165, 166, 170, 193, 194
Bosch, J., 169, 173, 180, 194
Bott, F., 31, 42
Bounimova, E., 300, 305
Brambilla, M., 139, 159, 163, 165
Brant, J., 80, 100, 279, 282
Brazendale, J., 347
Brereton, P., 517
Brilliant, S. S., 324, 338
Brook, P., 552, 579
Brooks, E. P., 665
Brown, A. W., 98, 684, 688, 699
Brown, L., 376, 405, 407
Bruno, E. J., 619, 637
Budgen, D., 517
Burns, A., 619, 631, 634, 635, 637
Buschmann, F., 175, 194, 195, 209, 224, 225
Buse, R. P. L., 726, 729

C

Cabot, J., 139, 159, 163, 165, 488
Calinescu, R. C., 300, 305, 583, 592, 609
Carollo, J., 252
Cha, S. S., 349, 371
Chapman, C., 220, 225
Chapman, R., 300, 305, 404, 406
Chaudron, M. R. V., 175, 195
Checkland, P., 559, 579
Chen, L., 169, 194
Cheng, B. H. C., 135
Chidamber, S., 721, 729
Chrissis, M. B., 67, 71, 734, 755
Christerson, M., 125, 137, 144, 165
Chulani, S., 684, 688, 699
Clark, B. K., 683, 684, 688, 691, 694, 699
Cleaveland, R., 371
Clements, P., 169, 170, 175, 192, 194
Cliff, D., 583, 592, 609
Cloutier, R., 563, 579
Cohn, M., 680, 697, 699
Coleman, D., 275, 282
Collins-Sussman, B., 216, 225, 735, 755
Connaughton, C., 727
Conradi, R., 69
Cook, B., 300, 305
Cooling, J., 627, 637
Coplien, J. O., 175, 194
Coulouris, G., 491, 517, 519
Councill, W. T., 467, 489
Crabtree, A., 117, 137
Cranor, L., 398, 406
Crnkovic, I., 487, 488
Cunningham, W., 84, 100, 203, 224
Curbera, F., 544, 550
Cusamano, M., 231, 254

D

Daigneau, R., 548
Dang, Y., 719, 726, 729
Datar, S. M., 275, 282
Davidsen, M. G., 272, 282
Davis, A. M., 102, 137
Deemer, P., 88, 100
Dehbonei, B., 356, 371
Deibler, W. J., 709, 729, 734, 755
Delmas, D., 356, 372
Delseny, H., 356, 357, 372
DeMarco, T., 665
den Haan, J., 159, 165
Devnani-Chulani, S., 688, 691, 694, 699
Dijkstra, E. W., 227, 254
Dipti, 282
Dollimore, J., 491, 517, 519
Douglass, B. P., 299, 305, 617, 620, 637
Duftler, M., 544, 550
Dunteman, G., 655, 666
Duquenoy, P., 31, 42
Dybä, T., 69, 84, 99

E

Ebert, C., 611, 635, 637
Edwards, J., 507, 519
El-Amam, K., 721, 729
Ellison, R. J., 425, 432, 434
Erickson, J., 140, 165
Erl, T., 526, 534, 548, 550
Erlikh, L., 256, 282

F

Fagan, M. E., 230, 254, 713, 729
Fairley, R. E., 563, 579
Faivre, A., 356, 371
Fayad, M. E., 446, 462
Fayoumi, A., 602, 609
Feathers, M., 280
Fielding, R., 530, 550
Firesmith, D. G., 383, 406
Fitzgerald, J., 300, 302, 303, 305, 735, 755
Fitzpatrick, B., 216, 225
Fogel, K., 222
Fowler, M., 80, 100, 279, 282
Fox, A., 517
Frank, E., 726, 729
Freeman, A., 28, 42

G

Gabriel, R. P., 581, 583, 607, 609
Gagne, G., 616, 637
Galin, D., 727
Gallis, H., 84, 99
Galvin, P. B., 616, 637
Gamma, E., 175, 194, 209, 210, 222, 225, 444, 463
Garfinkel, S., 398, 406
Garlan, D., 172, 175, 191, 192, 195, 459, 460, 461, 463
Gokhale, A., 443, 445, 463
Gotterbarn, D., 29, 40, 42
Graydon, P. J., 362, 371
Gregory, G., 82, 100, 233, 243, 254
Griss, M., 443, 463, 478, 489
Gryczan, G., 446, 462

H

Hall, A., 300, 305, 404, 406
Hall, E., 644, 666
Hall, M. A., 726, 729
Hamilton, S., 358, 371
Han, S., 719, 726, 729
Harel, D., 156, 165, 617, 637
Harford, T., 726, 729
Harkey, D., 507, 519
Harrison, N. B., 175, 194
Hatton, L., 325, 338
Heimdahl, M. P. E., 300, 305
Heineman, G. T., 467, 489
Helm, R., 175, 194, 209, 210, 222, 225, 444, 463
Henney, K., 175, 194, 209, 224
Heslin, R., 663, 666
Hitchins, D., 581, 608
Hnich, B., 487
Hofmeister, C., 174, 195
Holdener, A. T., 28, 42, 445, 463, 512, 519
Hollnagel, E., 409, 417–18, 434
Holtzman, J., 552, 579
Holzmann, G. J., 336, 358, 371
Hopkins, R., 94, 100, 256, 282
Horowitz, E., 683, 684, 688, 699
Howard, M., 405
Hudepohl, J. P., 360, 372
Hull, R., 151, 165
Humphrey, 67
Humphrey, W., 702, 713, 729
Hutchinson, J., 162, 165

I

Ince, D., 709, 729

J

Jackson, K., 552, 579
Jacobson, I., 24, 41, 42, 125, 137, 140, 144, 165, 166, 443, 463, 478, 489
Jain, P., 175, 195, 209, 225
Jeffrey, R., 727
Jeffries, R., 81, 84, 100, 140, 165, 242, 254
Jenkins, K., 94, 100, 256, 282
Jenney, P., 404, 407
Jhala, R., 300, 305, 358, 371
Joannou, D., 602, 609
Johnson, D. G., 31, 42
Johnson, R., 175, 194, 209, 210, 222, 225, 444, 463
Jones, C., 280, 611, 635, 637
Jones, T. C., 256, 282
Jonsson, P., 125, 137, 144, 165, 443, 463, 478, 489
Jonsson, T., 487

K

Kaner, C., 246, 254
Kawalsky, R., 602, 609
Kazman, R., 169, 170, 175, 192, 194
Keen, J., 583, 592, 609
Kelly, T., 583, 592, 609
Kemerer, C. F., 275, 282, 721, 729
Kennedy, D. M., 563, 579
Kerievsky, J., 279, 282
Kessler, R. R., 84, 100
Khalaf, R., 544, 550
Kifer, M., 186, 195
Kilner, S., 90, 100
Kindberg, T., 491, 517, 519
King, R., 151, 165
Kircher, M., 175, 195, 209, 225
Kitchenham, B., 718, 727, 729
Kiziltan, Z., 487
Klein, M., 581, 583, 607, 609
Kleppe, A., 485, 489
Knight, J. C., 324, 338, 362, 371
Knoll, R., 446, 462
Koegel, M., 161, 165
Konrad, M., 67, 71, 734, 755
Kopetz, H., 635
Korfiatis, P., 563, 579
Koskela, L., 59, 71
Koskinen, J., 275, 282
Kotonya, G., 473, 489
Kozlov, D., 275, 282
Krogstie, J., 272, 282
Krutchen, P., 46, 71, 173, 175, 195
Kuehl, S., 552, 579
Kumar, Y., 280
Kwiatkowska, M. Z., 300, 305, 358, 371, 583, 592, 609

L

Lamport, L., 495
Landwehr, C., 286, 303, 304
Lane, A., 392, 407
Lange, C. F. J., 175, 195
Laprie, J. C., 286, 303, 304, 409, 434
Larman, C., 73, 100, 222
Larsen, P.G., 300, 302, 303, 305
Lau, K-K., 466, 470, 487, 489
Laudon, K., 31, 42
LeBlanc, D., 405
Ledinot, E., 357, 371
Lee, E. A., 612, 637
Leffingwell, D., 95, 100
Lehman, M., 271
Leme, F., 82, 100, 233, 243, 254
Leveson, N. G., 324, 338, 368
Leveson, N. G., 349, 371
Levin, V., 300, 305, 358, 361, 370
Lewis, B., 521, 550
Lewis, P. M., 186, 195
Leymann, F., 532, 550
Lichtenberg, J., 300, 305
Lidman, S., 41, 42
Lientz, B. P., 256, 282
Lilienthal, C., 446, 462
Linger, R. C., 230, 254, 332, 338, 425, 432, 434
Lipson, H., 425, 434
Lister, T., 665

Loeliger, J., 216, 225, 735, 755
Lomow, G., 524, 550
Longstaff, T., 425, 432, 434
Loope, J., 753, 755
Lopes, R., 359, 371
Lou, J-G., 719, 726, 729
Lovelock, C., 521, 550
Lowther, B., 275, 282
Lutz, R. R., 238, 254, 371
Lutz, R. R., 340, 371
Lyu, M. R., 336, 338

M

Madachy, R., 683, 684, 688, 699
Madeira, H., 341, 372
Maier, M. W., 582, 583, 588, 589, 599–600, 607, 609
Majumdar, R., 300, 305, 358, 371
Marciniak, J. J., 69
Markkula, J., 275, 282
Marshall, J. E., 663, 666
Martin, D., 117, 137, 175, 195
Martin, R. C., 244, 254
Maslow, A. A., 383, 666
Massol, V., 82, 100, 233, 243, 254
McCay, B., 552, 579
McComb, S. A., 563, 579
McConnell, S., 713, 729
McCullough, M., 216, 225, 735, 755
McDermid, J., 583, 592, 609
McDougall, P., 510, 519
McGarvey, C., 300, 305
McGraw, G., 333, 338, 396, 407
McMahon, P. E., 41, 22
Mead, N. R., 425, 434
Mejia, F., 356, 371
Mellor, S. J., 159, 162, 165
Melnik, G., 81, 100, 242, 254
Menzies, T., 719, 725, 726, 727, 729
Meunier, R., 175, 194, 209, 225
Meyer, B., 485, 489
Meynadier, J-M., 356, 371
Miers, D., 544, 550
Mili, A., 476, 489
Mili, H., 476, 489
Miller, K., 29, 40, 42
Miller, S. P., 300, 305
Mitchell, R. M., 263, 282
Monate, B., 357, 371
Monk, E., 455, 463
Moore, A., 425, 432, 434
Morisio, M., 453, 460, 461, 463
Mostashari, A., 563, 579
Moy, Y., 357, 371
Mulder, M., 87, 100
Musa, J. D., 334, 338
Muskens, J., 175, 195

N

Nagappan, N., 360, 372
Nascimento, L., 461
Natarajan, B., 443, 445, 463
Naur, P., 19, 42
Newcomer, E., 524, 550
Ng, P-W., 41, 42
Nii, H. P., 180, 195
Nord, R., 174, 195
Norman, G., 358, 371
Northrop, L., 581, 583, 607, 609
Nuseibeh, B., 169, 194

O

O'Hanlon, C., 274, 282
Ockerbloom, J., 459, 460, 463
Oliver, D., 552, 579
Oman, P., 275, 282
Ondrusek, B., 300, 305
Opdahl, A. L., 386, 407
Opdyke, W., 80, 100, 279, 282
Oram, A., 510, 517, 519
Orfali, R., 507, 519
Ould, M., 644, 666
Overgaard, G., 125, 137, 144, 165
Owens, D., 552, 579

P

Paige, R., 583, 592, 609
Paries, J., 432, 434
Parker, D., 358, 371
Parnas, D., 296, 302, 305
Patel, S., 162, 166
Patterson, D., 517
Pautasso, C., 532, 550
Perrow, C., 342, 343, 371
Pfleeger, C. P., 376, 377, 407
Pfleeger, S. L., 376, 377, 407
Pilato, C., 216, 225, 735, 755
Pooley, R., 126, 137, 163
Poore, J. H., 230, 254, 332, 338
Pope, A., 466, 489, 493, 519
Prowell, S. J., 230, 254, 332, 338
Pullum, L., 318, 336, 338

Q

Quinn, M. J., 40

R

Rajamani, S. K., 300, 305, 358, 361, 370
Rajlich, V. T., 257, 282
Randell, B., 19, 42, 286, 303, 304, 307, 338
Ray, A., 368
Rayhan, S., 727
Raymond, E. S., 219, 225
Reason, J., 418, 420–21, 434
Regan, P., 358, 371
Reifer, D., 684, 688, 699
Richardson, L., 531, 550
Riehle, D., 446, 462
Rittel, H., 130, 137, 562, 579, 592, 609
Ritter, G., 637
Roberts, D., 80, 100, 279, 282
Robertson, J., 123, 135, 137
Robertson, S., 123, 135, 137
Rodden, T., 125, 137
Rodriguez, A., 548
Rogerson, S., 29, 40, 42
Rohnert, H., 175, 194, 195, 209, 225
Rosenberg, F., 544, 550
Rouncefield, M., 162, 165
Royce, W. W., 47, 71, 98, 687, 699
Rubin, K. S., 78, 85, 98, 100, 680, 699
Ruby, S., 531, 550
Rumbaugh, J., 140, 165, 166
Ryan, P., 754

S

Sachs, S., 731, 754, 755
Sakkinen, M., 275, 282
Sametinger, J., 470, 472, 489
Sami, M., 69
Sanderson, D., 516, 519
Sarris, S., 445, 463, 512, 519
Sawyer, P., 125, 137
Scacchi, W., 69
Schatz, B., 87, 100
Schmidt, D. C., 175, 194, 195, 209, 224, 225, 443, 445, 446, 462, 463, 581, 583, 607, 609
Schneider, S., 357, 371
Schneier, B., 384, 396, 407
Schoenfield, B., 392, 407
Schuh, P., 754
Schwaber, K., 71, 85, 100
Scott, J. E., 456, 463
Scott, K., 159, 165
Selby, R. W., 231, 254, 683, 699
Shaw, M., 172, 175, 191, 192, 195
Shimeall, T. J., 349, 371
Shou, P. K., 296, 305
Shrum, S., 67, 71, 734, 755
Siau, K., 140, 165
Silberschaltz, A., 616, 637
Sillitto, H., 578, 596, 600, 609
Silva, N., 341, 359, 371, 372
Sindre, G., 386, 407
Sjøberg, D. I. K., 69, 84, 99
Smart, J. F., 743, 755
Snipes, W., 360, 372
Sommerlad, P., 175, 194, 209, 225
Sommerville, I., 117, 135, 137, 175, 195, 461, 559, 579, 583, 592, 607, 609
Soni, D., 174, 195

Souyris, J., 356, 372
Spafford, E., 399, 401, 407
Spence, I., 41, 42
St. Laurent, A., 220, 225
Stafford, J., 488
Stahl, T., 159, 166
Stal, M., 205, 209, 225
Stallings, W., 376, 405, 407, 616, 637
Stapleton, J., 71, 100
Steece, B., 684, 688, 699
Stevens, P., 126, 137, 163
Stevens, R., 552, 579, 583, 609
Stewart, J., 574, 579
Stoemmer, P., 697
Storey, N., 349, 372
Strunk, E. A., 362, 371
Suchman, L., 117, 137
Swanson, E. B., 256, 282
Swartz, A. J., 294, 305
Szyperski, C., 467, 474, 487, 488, 489

T

Tahchiev, P., 82, 100, 233, 243, 254
Tanenbaum, A. S., 491, 519
Tavani, H. T., 31, 42
Thayer, R. H., 552, 579
Thayer, R. H., 563, 579
Tian, Y., 602, 609
Torchiano, M., 453, 460, 461, 463
Torres-Pomales, W., 318, 338
Trammell, C. J., 230, 254, 332, 338
Trimble, J., 299, 305
Tully, C., 552, 579
Turner, M., 517
Turner, R., 45, 71
Twidale, M., 125, 137

U

Ulrich, W. M., 276, 282
Ulsund, T., 69
Ustuner, A., 300, 305

V

Valeridi, R., 697
van Schouwen, J., 296, 305
Van Steen, M., 491, 519
van Vliet, M., 87, 100
Vandermerwe, S., 521, 550
Veras, P. C., 341, 372
Vicente, D., 359, 371
Viega, J., 396, 405, 407
Vieira, M., 341, 372
Villani, E., 341, 372
Viller, S., 117, 137
Virelizier, P., 300, 305, 358, 371
Vlissides, J., 175, 194, 209, 210, 222, 225, 444, 463
Voas, J., 333, 338
Voelter, M., 159, 166
Vogel, L., 32, 42, 218, 225
Vouk, M. A., 360, 372

W

Waeselynck, H., 300, 305, 358, 371
Wagner, B., 455, 463
Wagner, L. G., 300, 305
Wallach, D. S., 512, 519
Wang, Z., 466, 470, 487, 489
Warmer, J., 485, 489
Warren, I., 266, 282
Webber, M., 130, 137, 562, 579, 592, 609
Weils, V., 356, 357, 358, 372
Weinberg, G., 83, 100
Weiner, L., 203, 225
Weinreich, R., 470, 472, 489
Weise, D., 159, 165
Wellings, A., 619, 631, 634, 635, 637
Westland, C., 683, 699
Whalen, M. W., 300, 305
Wheeler, D. A., 396, 407
Wheeler, W., 52, 71, 473, 489
White, J., 52, 71, 473, 489
White, S. A., 544, 550
White, S., 552, 579
Whittaker, J. A., 237, 242, 252, 254

Whittle, J., 162, 165
Wiels, V., 300, 305, 371
Wilkerson, B., 203, 225
Willey, A., 552, 579
Williams, L., 84, 100, 360, 372
Williams, R., 574, 579
Wimmer, M., 139, 159, 163, 165
Wirfs-Brock, R., 203, 225
Witten, I. H., 726, 729
Woodcock, J., 300, 302, 303, 305
Woods, D., 432, 434
Wreathall, J., 432, 434
Wysocki, R. K., 665

X

Xie, T., 719, 726, 729

Y

Yacoub, S., 476, 489
Yamaura, T., 252

Z

Zelkowitz, M., 637
Zhang, D., 719, 726, 729
Zhang, H., 719, 726, 729
Zhang, Y., 162, 166
Zheng, J., 360, 372
Zimmermann, O., 532, 550
Zimmermann, T., 719, 725, 726, 727, 729
Zullighoven, H., 446, 462
Zweig, D., 275, 282