Introduction. Basic concepts: maintenance, evolution, models and processes, software configuration management, reengineering, legacy systems, impact analysis, refactoring, program comprehension, reuse [1], p. 1-16

Contents

[Introduction. Basic concepts 1](#_Toc177913544)

[1. Maintenance versus evolution 1](#_Toc177913545)

[2. Software Evolution 2](#_Toc177913546)

[3. Software Maintenance 3](#_Toc177913547)

[4. Software maintenance and evolution models and processes 4](#_Toc177913548)

[5. Reengineering 7](#_Toc177913549)

[6. Legacy Systems 8](#_Toc177913550)

[7. Impact Analysis 9](#_Toc177913551)

[8. Refactoring 10](#_Toc177913552)

[9. Program Comprehension 11](#_Toc177913553)

[10. Software Reuse 11](#_Toc177913554)

# Introduction. Basic concepts

## Maintenance versus evolution



<https://en.wikipedia.org/wiki/Software_maintenance>

The concept of *software maintenance* means preventing software from failing to deliver the intended functionalities by means of bug fixing.

The concept of *software evolution* means a continual change from a lesser, simpler, or worse state to a higher or better state

All support activities carried out *after* delivery of software are put under the category of *maintenance*.

All activities carried out to effect changes in requirements are put under the category of *evolution*

Maintenance of software systems primarily means fixing bugs but preserving their functionalities. Maintenance tasks are very much planned. For example, bug fixing must be done and it is a planned activity. In addition to the planned activities, unplanned activities are also necessitated. For example, a new usage of the system may emerge. Generally, maintenance does not involve making major changes to the architecture of the system. In other words, maintenance means keeping an installed system running with no change to its design.

Evolution of software systems means creating new but related designs from existing ones. The objectives include supporting new functionalities, making the system perform better, and making the system run on a different operating system.

## 2. Software Evolution

Based on empirical studies, **Lehman** and his collaborators formulated some observations and they introduced them as *laws of evolution*. The “laws” themselves have “evolved” from *three* in 1974 to *eight* by 1997. Those laws are the results of studies of the evolution of large-scale proprietary or

closed source software (CSS) systems. The laws concern a category of software systems called *E-type* systems.

In his 1980 article,[[1]](https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution#cite_note-laws1-1) Lehman qualified the application of such laws by distinguishing between three categories of software:

* An *S*-program is written according to an exact specification of what that program can do. For example, a program to find solutions to the [eight queens puzzle](https://en.wikipedia.org/wiki/Eight_queens_puzzle) would be an *S*-program. These programs are mostly static and shouldn't evolve much.
* A *P*-program is written to implement certain procedures that completely determine what the program can do (the example mentioned is a program to play chess)
* An *E*-program is written to perform some real-world activity; how it should behave is strongly linked to the environment in which it runs, and such a program needs to adapt to varying requirements and circumstances in that environment

The eight laws are briefly explained as follows:

1. *Continuing change*. Unless a system is continually modified to satisfy emerging needs of users, the system becomes increasingly less useful.

2. *Increasing complexity*. Unless additional work is done to explicitly reduce the complexity of a system, the system will become increasingly more complex due to maintenance-related changes.

3. *Self-regulation*. The evolution process is self-regulating in the sense that the measures of products and processes that are produced during the evolution follow close to *normal* distributions.

4. *Conservation of organizational stability*. The average effective global activity rate on an evolving system is almost constant throughout the lifetime of the system. In other words, the average amount of additional effort needed to produce a new release is almost the same.

5. *Conservation of familiarity*. As a system evolves all kinds of personnel, namely, developers and users, for example, must gain a desired level of understanding of the system’s content and behaviour to realize satisfactory evolution. A large incremental growth in a release reduces that understanding. Therefore, the average incremental growth in an evolving system remains almost the same.

6. *Continuing growth*. As time passes, the functional content of a system is continually increased to satisfy user needs.

7. *Declining quality*. Unless the design of a system is diligently fine-tuned and adapted to new operational environments, the system’s qualities will be perceived as declining over the lifetime of the system.

8. *Feedback system*. The system’s evolution process involves multi-loop, multi-agent, multi-level feedback among different kinds of activities. Developers must recognize those complex interactions in order to continually evolve an existing system to deliver more functionalities and higher levels of qualities.

## Software Maintenance

More likely than not, there are defects in delivered software applications, because defect removal and quality control processes are not perfect. Therefore, maintenance is needed to repair those defects in released software.

Initially three categories of software maintenance activities, namely, ***corrective*, *adaptive*, and *perfective*** were defined. Those definitions were later incorporated into the standard software engineering–software life cycle processes–Maintenance and introduced a fourth category called ***preventive***maintenance. But some researchers and developers view preventive maintenance as a subset of perfective maintenance.

An alternative intention-based way of classifying modifications to software is to simply categorize the modifications in terms of activities performed:

*Activities to make corrections.* If there are discrepancies between the expected behaviour of a system and the actual behaviour, then some activities are performed to eliminate or reduce the discrepancies.

*Activities to make enhancements.* A number of activities are performed to implement a change to the system, thereby changing the behaviour or implementation of the system. This category of activities is further refined into three subcategories:

**–** enhancements that modify existing requirements;

**–** enhancements that create new requirements; and

**–** enhancements that modify the implementation without changing the requirements

Later the typology of was extended 12 different types of software maintenance: **training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive**.

***Maintenance of COTS-Based Systems***

Many present-day software systems are built from components previously developed for other systems or to be reused in many systems. In this approach, new components are developed by combining commercial off-the-shelf (COTS) components, custom-built (in-house) components, and open source software components. The components are obtained from a variety of sources and maintained by different vendors, possibly from different countries. The motivations for performing software maintenance are the same for both component based software systems (CBS) and custom-built software systems. However, there are noticeable differences between the activities in the two approaches. The major sources of the differences are as follows

* *Skills of system maintenance teams.* Maintenance of CBS requires specialized skills to monitor and integrate COTS products. Those skills are different than the skills required to perform the more traditional maintenance functions: analyse and modify source code developed in-house. Maintainers view a CBS as a group of black-box components, and not as a compiled set of source code modules, thereby requiring a different set of maintenance skills. The differences in skills are neither pros nor cons, but it is important that the differences are taken into consideration for planning, staffing, and training.
* *Infrastructure and organization.* Running a support group for in-house products is necessary to manage a large product. This additional cost may be shared with other projects.
* *COTS maintenance cost.* This cost includes the costs of purchasing components, licensing components, upgrading components, and training maintenance personnel. From the perspective of a system’s life cycle, much cost is shifted from in-house development to license and maintenance fees, thereby increasing the overall maintenance cost.
* *Larger user community.* COTS users are part of a broad community of users, and the community of users can be considered as a resource, which is a positive factor. However, being part of a community means having less control over changes and improvements to COTS products.
* *Modernization.* In general, vendors of COTS components keep pace with changing technology and continually update the components. As a result, the system does not become obsolete. However, the flip side is that the costs and risks of making changes keep increasing even if the application does not require any changes. In general, control over the evolution and maintenance of significant portions of the system is relinquished to third-party COTS developers. Those third-party developers may be motivated to pursue their own commercial self-interest. In addition, the third-party vendors control not only the nature of maintenance to be done on the products, but also when it is to be done. Therefore reliance on third-party products impacts both the type and timing of the maintenance performed by COTS-based developers. In a nutshell, unfortunately, upgrades to products are necessitated by technology and vendor economics.
* *Split maintenance function.* A COTS product is maintained by its vendor, whereas the overall system that uses the COTS product is maintained by the system’s host organization. As a result, multiple, independent maintenance teams exist. The advantage of COTS-based development is that the system maintainers receive additional support from the COTS vendors. On the other hand, the drawback of the approach is that the different COTS pieces need tighter coordination, and the product vendors may stray in all directions with respect to functionality and standard.
* *More complex planning.* If a system depends upon multiple technologies and COTS products, the unpredictability and risk of change become high, and planning becomes complicated because coordination among a large number of vendors is more difficult.

## Software maintenance and evolution models and processes

The final phase of the Waterfall model is known as maintenance, which implies that software maintenance is a part of software development. But software maintenance should have its own software maintenance life cycle (SMLC) model. Three common features of the SMLC models found in the literature are:

* understanding the code;
* modifying the code; and
* revalidating the code.

Other models view software development as *iterative* processes and based on the idea of *change mini-cycle* as explained in the following:

* *Iterative models.* The iterative models share the ideas that a complete set of requirements for a system cannot be completely understood, or the developers do not know how to build the full system. Therefore, systems are constructed in builds, each of which is a refinement of requirements of the previous build. A build is refined by considering feedback from users. One may note that maintenance and evolution activities do not exist as distinct phases. Rather, they are closely intertwined.
* *Change mini-cycle models.* These models consist of five major phases: change request, analyze and plan change, implement change, verify and validate, and documentation change. In this process model, several important activities were identified, such as **program comprehension, impact analysis, refactoring, and change propagation.**

A different kind of software evolution model, called *staged model of maintenance and evolution*, is descriptive in nature, and its primary objective is to improve the understanding of how long-lived software evolves. The model considers four distinct, sequential stages of the lifetime of a system, as explained below:

1. *Initial development.* When the initial version of the system is produced, detailed knowledge about the system is fresh. Before delivery of the system, it undergoes many changes. Eventually, a system architecture emerges and soon it stabilizes.

2. *Evolution.* After the initial stability, it is easy to perform simple changes to the system. Significant changes involve higher cost and higher risk. In the period immediately following the initial delivery, knowledge about the system is still almost fresh in the minds of the developers. It is possible that the development team as a whole does not exist, because many original developers have taken up new responsibilities in the organization and some might have left the organization. In general, for many systems, their lifespan are spent in this stage, because the systems continue to be of importance to the organizations.

3. *Servicing.* When the knowledge about the system has significantly decreased, the developers mainly focus on maintenance tasks, such as fixing bugs, whereas architectural changes are rarely effected. The developers do not consider the system to be a key asset. In this stage, the effects of changes are very difficult to predict. Moreover, the costs and risks of making changes are very significant.

4. *Phaseout.* When even minimal servicing of a system is not an option, the system enters its very final stage. The organization decides to replace the system for various reasons: (i) it is too expensive to maintain the system; or (ii) there is a newer solution available. Therefore, the organization develops an exit strategy to move from the current system to a new system. Moving from an existing, difficult-to-maintain system to a modern solution system has its own challenges involving wrapping and data migration. After the new system keeps running satisfactorily, sometimes in parallel with the old system, the old system is finally completely shut down.

***Software Maintenance Standards***

A well-defined process for software maintenance can be observed and measured, and thus improved. In addition, adoption of processes allows the dissemination of effective work practices more quickly than gaining personal experience. Process centric software maintenance is more of an engineering activity, with predictable time and effort constraints, and less of an art. Therefore, software maintenance standards have been formulated by ISO and IEEE. The maintenance standard document from ISO is called ISO/IEC 14764 which is a part of the standard document ISO/IEC 12207 for life cycle processes. The maintenance standard document from IEEE is called IEEE/EIA 1219. Both the standards describe processes for managing and executing activities for maintenance. The IEEE/EIA 1219 standard organizes the maintenance process in seven phases: **problem identification, analysis, design, implementation, system test, acceptance test, and delivery**. As a quick summary, the standard identifies the different phases and the sequence of their executions. Next, for each phase, the standard identifies the input and output deliverables, the supporting processes and the related activities, and a set of evaluation metrics. Both the standards, namely ISO/IEC 14764 and IEEE/EIA 1219, use the same terminology to describe software maintenance, with a little difference in their depictions. An iterative process has been described in ISO/IEC 14764 to manage and execute maintenance activities. The activities comprising the maintenance process are:

* process implementation;
* problem and modification analysis;
* modification implementation;
* maintenance review/acceptance;
* migration; and
* retirement.

Each of the aforementioned activities is made up of tasks described with specific inputs, outputs, and actions

***Software Configuration Management***

Configuration management (CM) is the discipline of managing changes in large systems. The goal of CM is to manage and control the various extensions, adaptations, and corrections that are applied to a system over its lifetime. It handles the control of all products/configuration items and changes to those items. Software configuration management (SCM) is the configuration management applied to software systems. SCM is the means by which the process of software evolution is managed. SCM has been defined in the IEEE 1042 standard as “software configuration management (SCM) is the discipline of managing and controlling change in the evolution of software systems.” SCM provides a framework for managing changes in a controlled manner. The purpose of SCM is to reduce communication errors among personnel working on different aspects of the software project by providing a central repository of information about the project and a set of agreed upon procedures for coping with changes. It ensures that the released software is not contaminated by uncontrolled or unapproved changes. Early SCM tools had limited capabilities in terms of functionality and applicability. However, modern SCM systems provide advanced capabilities through which many different artefacts are managed. For example, modern SCM systems support their users in building an executable program out of its versioned source files. Moreover, it must be possible to regenerate old versions of the software system. In general, an SCM system has four different elements, each element addressing a distinct user need as follows:

* *Identification of software configurations.* This includes the definitions of the different artefacts, their baselines or milestones, and the changes to the artefacts.
* *Control of software configurations.* This element is about controlling the ways artefacts or configurations are altered with the necessary technical and administrative support.
* *Auditing software configurations.* This element is about making the current status of the software system in its life cycle visible to management and determining whether or not the baselines meet their requirements.
* *Accounting software configuration status.* This element is about providing an administrative history of how the software system has been altered, by recording the activities necessitated by the other three SCM elements.

## Reengineering

Software evolution can be defined as the process of conducting continuous software reengineering. Reengineering implies a single cycle of taking an existing system and generating from it a new system, whereas evolution can go forever. In other words, to a large extent, **software evolution can be seen as repeated software reengineering**. Reengineering is done to transform an existing “lesser or simpler” system into a new “better” system. Reengineering can be defined as the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Therefore, reengineering includes some kind of reverse engineering activities to design an abstract view of a given system. The new abstract view is restructured, and forward engineering activities are performed to implement the system in its new form. The aforementioned process is captured by the following expression:

Reengineering = Reverse engineering+Δ+Forward engineering*.*

Let us analyze the right-hand portion of the above equation. The first element “reverse engineering” is the activity of defining a more abstract and easier to understand representation of the system. For example, the input to the reverse engineering process is the source code of the system, and the output is the system architecture. The core of reverse engineering is the process of examination of the system, and it is not a process of change. Therefore it does not involve changing the software under examination. The third element “forward engineering” is the traditional process of moving from a high-level abstraction and logical, implementation-independent design to the physical implementation of the system. The second element “Δ” captures alterations performed to the original system. While performing reverse engineering on a large system, tools and methodologies are generally not stable. Therefore, a high-level organizational paradigm enables repetitions of processes so that maintenance engineers learn about the system. A repeatable paradigm, called Goals/Models/Tools, describes reverse engineering in three successive stages, namely, Goals, Models, and Tools.

*Goals.* In this phase, one analyzes the motivations for setting up the process to identify the information needs of the process and the abstractions to be produced.

*Models.* In this phase, one analyzes the abstractions to construct representation models that capture the information needed for their production.

*Tools.* In this phase, software tools are defined, acquired, enhanced, integrated, or constructed to: (i) execute the Models phase and (ii) transform the program models into the abstractions identified in the Goals phase.

It is important to note that fact-finding and information gathering from the source code are keys to the Goal/Models/Tools paradigm. In order to extract information that is not explicitly available in source code, automated analysis techniques, such as *lexical analysis, syntactic analysis, control flow analysis, data flow analysis,* and *program slicing* are used to facilitate reverse engineering.

The increased use of data mining techniques in support systems have given rise to an interest in data reverse engineering (DRE) technology. DRE tackles the question of what information is stored and how this information can be used in a different context. DRE is defined as the use of structured techniques to reconstitute the data assets of an existing system. The two vital aspects of the DRE process are to: (i) recover *data assets* that are useful or valuable and (ii) *reconstitute* the recovered data assets to make them more useful. Therefore, DRE can be regarded as adding value to the existing data assets, making it easier for organizations to conduct business efficiently and effectively.

## Legacy Systems

A legacy software system is an old program that continues to be used because it still meets the users’ needs, in spite of the availability of newer technology or more efficient methods of performing the task. More often than not, a legacy system includes outdated procedures or terminology, and it is very difficult for new developers to understand the system. Organizations continue to use legacy systems because those are vital to them and the systems significantly resist modification and evolution to meet

new and constantly changing business requirements.

Organizations in business for a long time generally possess a sizable number of legacy systems. To manage legacy systems, a number of options are available. Some commonly chosen options are as follows:

* *Freeze.* An organization decides to perform no further work on a legacy system. This implies that either the services of the system are no longer needed or a new system completely replaces a legacy system.
* *Outsource.* An organization may decide that supporting legacy software—or for that matter any software—is not its core business. As an alternative, it may outsource the support service to a specialist organization.
* *Carry on maintenance.* In this approach, the organization continues to maintain the system for another period of time, despite all the difficulties in doing so.
* *Discard and redevelop.* In this approach, the application is redeveloped once again from scratch, using new hardware and software platforms, new software architecture and databases, and modern tools. When the new system is available, the legacy system is simply discarded.
* *Wrap.* In this approach, a legacy system is wrapped around with a new software layer, thereby hiding the unwanted complexity of the existing data, individual programs, application systems, and interfaces. The old system performs the actual computations, but users interact with the system in better ways. The notion of “wrapper” was first introduced at IBM in the late 1980s. Wrapping is a black-box reengineering task, because only the legacy interface is analyzed while ignoring the system’s internals. A wrapper does not directly modify the source code, but it indirectly modifies the software functionality of the legacy component. Wrapping lets organizations reuse well tested components that they trust and leverage their massive investments in the system. As a result, the lifetime of the legacy system is increased.
* *Migrate.* In this approach, an operational legacy system is moved to a new hardware and/or software platform, while still retaining the legacy system’s functionality. The idea is to minimize any disruption to the existing business environment.

## Impact Analysis

Impact analysis is the task of identifying portions of the software that can potentially be affected if a proposed change to the system is affected. The outcome of impact analysis can be used when planning for changes, making changes, and tracking the effects of changes in order to localize the sources of new faults. Impact analysis techniques can be categorized into two classes as follows:

* *Traceability analysis.* In this approach, the high-level artefacts, such as requirements, design, code, and test cases related to the feature to be changed, are identified. A model of associations among artefacts, such that each artefact links to other artefacts, is constructed. This helps in locating the corresponding portions of the design, code, and test cases that need to be maintained.
* *Dependency analysis.* Dependency analysis attempts to assess the effects of a change on the semantic dependencies between program entities. This is achieved by identifying the syntactic dependencies that may signal the presence of such semantic dependencies. The two dependency-based impact analysis techniques are: call graph-based analysis and dependency graph-based analysis. Dependency analysis is also known as source code analysis.

The following two additional notions are found to be keys to understanding impact analysis:

* *Ripple effect analysis.* Ripple effect analysis emphasizes the tracing repercussions in source code when the code is changed. It measures the impact of a change to a particular module on the rest of the program. Impact can be stated in terms of the problems being created for the rest of the program because of the change. Analysis of ripple effect can provide information regarding what changes are occurring and where they are occurring. Measurement of ripple effect can provide knowledge about the system as a whole through its evolution: (i) the amount of increase or decrease of its complexity since the previous version; (ii) the levels of complexity of individual parts of a system in relation to other parts of the system; and (iii) the effect that a new module has on the complexity of a system as a whole when it is added.
* *Change propagation.* Change propagation activities ensure that a change made in one component is propagated properly throughout the entire system. Misunderstanding, lack of experience, and unexpected dependencies are some reasons for failing to propagate changes throughout the development and maintenance cycles of source code. If a change is not propagated correctly, the project risks the introduction of new defects.

## Refactoring

Refactoring means performing changes to the structure of software to make it easier to comprehend and cheaper to make subsequent changes without changing the observable behavior of the system. A similar idea for non-object-oriented systems is called restructuring. Refactoring is achieved through removal of duplicate code, simplification of code, and moving code to a different class, among others. Without continual refactoring, the internal structure of software will eventually deform beyond comprehension, due to periodic maintenance. Therefore, regular refactoring helps the system to retain its basic structure. In an agile software methodology, such as eXtreme Programming (XP), refactoring is continuously applied to: (i) make the architecture of the software stable; (ii) render the code readable; and (iii) make the tasks of integrating new functionalities into the system flexible. An important characteristic of refactoring is that it must preserve the “observable behavior” of the system. Preservation of the observable behavior is verified by ensuring that all the tests passing before refactoring must pass after refactoring.

Regression testing is used to ensure that the system did not deviate from the original system during refactoring. Refactoring does not normally involve code transformation to implement new requirements. Rather, it can be performed without adding new requirements to the existing system. Another aspect of refactoring is to enhance the internal structure of the system. In addition, the concept of program restructuring can be applied to transform legacy code into a more structured form and migrate it to a different programming language. That is, restructuring and refactoring can be used to reengineer software systems.

Refactoring techniques put emphasis on the development of a list of basic refactorings, which can be combined to form complex refactorings. The original list of basic refactorings contained transformations on object-oriented code: (i) add a class, method, or attribute; (ii) rename a class, method, or attribute; (iii) move an attribute or method up or down the hierarchy; (iv) remove a class, method, or attribute; and (v) extract chunks of code into separate methods. Most complex refactoring scenarios require small code changes for the refactorings to work correctly. Primitive refactorings are rarely used in isolation.

## Program Comprehension

The purpose of program comprehension is to understand an existing software system for planning, designing, coding, and testing changes. It was observed in 1989 that *program comprehension* accounts for 50% of the total effort expended throughout the life cycle of a software system. Therefore, good understanding of the software is key to raising its quality by means of maintenance at a lower cost. In terms of concrete activities, program comprehension involves building mental models of an underlying system at different levels of abstractions, varying from low-level models of the code to very high-level models of the underlying application domain. Mental models have been studied by cognitive scientists to understand how human beings know, perceive, make decisions, and construct behavior in a real world. In the domain of program comprehension, a mental model describes a programmer’s mental representation of the program to be comprehended. Program comprehension involves constructing a mental model of the program by applying various cognitive processes. A key step in developing mental models is generating hypotheses, or conjectures, and investigating their validity. Hypotheses are a way for a programmer to understand code in an incremental manner. After some understanding of the code, the programmer forms a hypothesis and verifies it by reading code. Verification of hypothesis results in either accepting the hypothesis or rejecting it. Sometimes, a hypothesis may not be completely correct because of incomplete understanding of code by the programmer. By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

One can apply several strategies to arrive at meaningful hypotheses, such as bottom–up, top–down, and opportunistic combinations of the two. A bottom–up strategy works by beginning with the code, whereas a top–down strategy operates by working from a high-level goal. A strategy is formulated by identifying actions to achieve a goal. Strategies guide two mechanisms, namely, chunking and cross-referencing to produce higher-level abstraction structures. Chunking creates new, higher-level abstraction structures from lower-level structures. Cross-referencing means being able to make links between elements of different abstraction levels. This helps in building a mental model of the program under study at different levels of abstractions. In general, understanding a program involves a knowledge base, which represents the expertise and background knowledge a programmer brings to the table, a mental model, and an assimilation process. The assimilation process guides the programmer to look at certain pieces of information, such as a code segment or a comment, and move forward/backward while reading the code. The assimilation process can work in three ways: top–down, bottom–up, and opportunistic.

## Software Reuse

The idea of software reuse was first introduced in 1968 and proposed to realize reuse by means of library components and automated ways for customizing components to varying degrees of robustness and precision. Other ideas include program families and domain analysis. A program family is a set of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analysing individual differences. On the other hand, domain analysis is an activity of identifying objects and operations of a class of similar systems in a particular problem domain. Simply stated, software reuse means using existing software knowledge or artefacts during the development of a new system. Reusable assets can include both artefacts and software knowledge. Note that reuse is not constrained to source code fragments. Rather, there were identified four broad types of artefacts for reuse:

* *Data reuse.* This involves a standardization of data formats. Reusable functions imply a standard data interchange format. Therefore, one of the critical precursors to full reusable software is that of reusable data.
* *Architectural reuse.* This consists of standardizing a set of design and programming conventions dealing with the logical organization of software. The goal is to define a complete set of functional elements which will be needed to create new systems from standard components.
* *Design reuse.* This deals with the reuse of abstract designs that do not include implementation details. These are then implemented specifically to fit the application requirements.
* *Program reuse.* This deals with reusing running code. The software units that are reused may be of different sizes. The whole of software system may be reused by incorporating it without change into other system (COTS product reuse)

Reusability is a property of software assets, which indicates the degree to which the software can be reused. For a software component to be reusable, it must exhibit the following characteristics: high cohesion (**Cohesion** refers to the degree to which elements within a module work together to fulfil a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes), low coupling (**Coupling**refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules), adaptability, understandability, reliability, and portability. Those characteristics encourage the component’s reuse in similar situations. There are two advantages of reusing previously written code:

* *Better quality.* If previously tested modules are reused in a new software project, the reused modules are likely to have less faults than new modules. This reduces the overall failure rate of the new software.
* *Increased productivity.* Organizations can save time and other resources by reusing operational modules, thereby increasing their overall productivity. However, the quantum of increase depends upon the size and complexity of the components being reused and the overall size and complexity of the new software which reuses those components. The development cost of any software project is only about 40% of the total cost over its entire life cycle. Significant maintenance benefit also results from reusing quality software. The empirical study shows that the cost savings during maintenance, as a consequence of reuse, are nearly twice the corresponding savings during development.