Taxonomy of software maintenance and evolution, classification of software maintenance, types of software programs, software evolution laws, maintenance of COTS-based systems [1], p. 25-75

Contents

[Taxonomy of software maintenance and evolution 1](#_Toc178258653)

[1. Software maintenance classification 1](#_Toc178258654)

[1.1. Intention-Based Classification of Software Maintenance 2](#_Toc178258655)

[1.2. Activity-Based Classification of Software Maintenance 3](#_Toc178258656)

[1.3. Evidence-Based Classification of Software Maintenance 4](#_Toc178258657)

[2. CATEGORIES OF MAINTENANCE CONCEPTS 11](#_Toc178258658)

[3. EVOLUTION OF SOFTWARE SYSTEMS 15](#_Toc178258659)

[3.1 SPE Taxonomy 15](#_Toc178258660)

[3.2. Practical Implications of the Lehman’s Laws 19](#_Toc178258661)

[4. MAINTENANCE OF COTS-BASED SYSTEMS 20](#_Toc178258662)

# Taxonomy of software maintenance and evolution

## Software maintenance classification

The ISO/IEC 14764 standard defines software maintenance as “…the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage”. Post-delivery activities include changing software, providing training, and operating a help desk. Pre-delivery activities include planning for post-delivery operations. During the development process, maintainability is specified, reviewed, and controlled. If this is done successfully, the maintainability of the software will improve during the post-delivery stage. The standard further defines software maintainability as “…the capability of the software product to be modified. Modification may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specification”.

A major difference exists between software maintenance and software development: maintenance is event driven, whereas development is requirements driven. A process for software development begins with the objective of designing and implementing a system to deliver certain functional and non-functional requirements. On the other hand, a maintenance task is scheduled in response to an event. Reception of a *change request* from a customer is a kind of event that can trigger software maintenance. Similarly, recognition of the needs to fix a set of bugs is considered another kind of event. A maintenance activity accepts some existing artifacts as inputs and generates some new and/or modified artifacts. In general, an investigation activity is the first activity in a maintenance process. In an investigation activity, a maintenance engineer evaluates the nature of the events, say, a change request (CR). Finding the impact of executing the CR is an example of investigation activity. Upon completion of the first activity, the organization decides whether or not to proceed with the modification activity.

In the following, maintenance activities are explained from three viewpoints:

* Intention-based classification of software maintenance activities;
* Activity-based classification of software maintenance activities; and
* Evidence-based classification of software maintenance activities.

### Intention-Based Classification of Software Maintenance

In the intention-based classification, one categorizes maintenance activities into four groups based on what we intend to achieve with those activities. Based on the Standard for Software Engineering–Software Maintenance, ISO/IEC 14764, the four categories of maintenance activities are corrective, adaptive, perfective, and preventive as explained in the following.

*Corrective maintenance.* The purpose of corrective maintenance is to correct failures: processing failures and performance failures. A program producing a wrong output is an example of processing failure. Similarly, a program not being able to meet real-time requirements is an example of performance failure. The process of corrective maintenance includes isolation and correction of defective elements in the software. The software product is repaired to satisfy requirements. There is a variety of situations that can be described as corrective maintenance such as correcting a program that aborts or produces incorrect results. Basically, corrective maintenance is a *reactive* process, which means that corrective maintenance is performed after detecting defects with the system.

*Adaptive maintenance.* The purpose of adaptive maintenance is to enable the system to adapt to changes in its data environment or processing environment. This process modifies the software to properly interface with a changing or changed environment. Adaptive maintenance includes system changes, additions, deletions, modifications, extensions, and enhancements to meet the evolving needs of the environment in which the system must operate. Some generic examples are: (i) changing the system to support new hardware configuration; (ii) converting the system from batch to online operation; and (iii) changing the system to be compatible with other applications.

*Perfective maintenance.* The purpose of perfective maintenance is to make a variety of improvements, namely, user experience, processing efficiency, and maintainability. For example, the program outputs can be made more readable for better user experience; the program can be modified to make it faster, thereby increasing the processing efficiency; and the program can be restructured to improve its readability, thereby increasing its maintainability. In general, activities for perfective maintenance include restructuring of the code, creating and updating documentations, and tuning the system to improve performance. It is also called “maintenance for the sake of maintenance” or “reengineering”.

*Preventive maintenance.* The purpose of preventive maintenance is to prevent problems from occurring by modifying software products. Basically, one should look ahead, identify future risks and unknown problems, and take actions so that those problems do not occur. For example, good programming styles can reduce the impact of change, thereby reducing the number of failures. Therefore, the program can be restructured to achieve good styles to make later program comprehension easier. Preventive maintenance is very often performed on safety critical and high available software systems. The concept of “software rejuvenation” is a preventive maintenance measure to prevent, or at least postpone, the occurrences of failures due to continuously running the software system. *Software rejuvenation* is a proactive fault management technique aimed at cleaning up the system internal state to prevent the occurrence of more severe crash in the future. It involves occasionally terminating an application or a system, cleaning its internal state, and restarting it. Rejuvenation may increase the downtime of the application; however, it prevents the occurrence of more severe and costly failures. In a safety critical environment, the necessity of performing preventive maintenance is evident from the example of control software for Patriot missile: “On 21 February, the office sent out a warning that ‘very long running time’ could affect the targeting accuracy. The troops were not told, however, how many hours ‘very long’ was or that it would help to switch the computer off and on again after 8 hours”. The purpose of software maintenance activities of preventive maintenance of a safety critical software system is to eliminate hazard or reduce their associated risk to an acceptable level. Note that a *hazard* is a state of a system or a physical situation which, when combined with certain environment conditions, could lead to an accident. A hazard is a prerequisite for an accident or mishap

## Activity-Based Classification of Software Maintenance

Maintenance activities are put into two categories: corrections and enhancements.

* *Corrections.* Activities in this category are designed to fix defects in the system, where a defect is a discrepancy between the expected behavior and the actual behavior of the system.
* *Enhancements.* Activities in this category are designed to effect changes to the system. The changes to the system do not necessarily modify the behavior of the system. This category of activities is further divided into three subcategories as follows:

**–** enhancement activities that modify some of the existing requirements implemented by the system;

**–** enhancement activities that add new system requirements; and

**–** enhancement activities that modify the implementation without changing the requirements implemented by the system.

Now one can find a mapping between intention-based and activity-based terminology. Enhancement activities which are necessary to change the implementations of existing requirements are similar to perfective maintenance. Enhancement activities, which add new requirements to a system, are similar to adaptive maintenance. Enhancement activities which do not impact requirements but merely affect the system implementation appear to be similar to preventive maintenance.

## Evidence-Based Classification of Software Maintenance

The intention-based classification is further refined by evidence-based classification. The objectives of this classification are as follows:

* base the classification on objective evidence that can be measured from observations and comparisons of software before and after modifications;
* set the coarseness of the classification to truly reflect a representative mix of observed activities;
* make the classification independent of the execution and development environments: hardware, operating system (OS), organizational practices, design methodology, implementation language, and personnel involved in maintenance.

Modifications performed, detected, or observed on four aspects of the system being maintained are used as the criteria to cluster the types of maintenance activities:

* the whole software;
* the external documentation;
* the properties of the program code; and
* the system functionality experienced by the customer.

Classification of maintenance activities is based on changes in the aforementioned four kinds of entities. Evidence of changes to those entities is gathered by comparing the appropriate portions of the software *before* the activity with the appropriate parts *after* the execution of the activity. In general, software maintenance or evolution involves many activities that may result in some kind of modifications to the software. A dominant categorization of activities emerges from all the modifications made, detected, or observed. The classification has mutually exclusive types grouped into clusters as illustrated in Figure 2.1. The definitions of the 12 types of maintenance activities are given in Table 2.1.



**TABLE 2.1 Evidence-Based 12 Mutually Exclusive Maintenance Types**

|  |  |  |
| --- | --- | --- |
| **#** | Types ofMaintenance | Definitions |
|  | **Training** | This means training the stakeholders about the implementation of the system. |
|  | **Consultive** | In this type, cost and length of time are estimated for maintenance work, personnel run a help desk, customers are assisted to prepare maintenance work requests, and personnel make expert knowledge about the available resources and the system to others in the organization to improve efficiency. |
|  | **Evaluative** | In this type, common activities include reviewing the program code and documentations, examining the ripple effect of a proposed change, designing and executing tests, examining the programming support provided by the operating system,and finding the required data and debugging. |
|  | **Reformative** | Ordinary activities in this type improve the readability of the documentation, make the documentation consistent with other changes in the system, prepare training materials, and add entries to a data dictionary. |
|  | **Updative** | Ordinary activities in this type are substituting out-of-date documentation with up-to-date documentation, making semi-formal, say, in UML to document current program code, and updating the documentation with test plans. |
|  | **Groomative** | Ordinary activities in this type are substituting components and algorithms with more efficient and simpler ones, modifying the conventions for naming data, changing access authorizations, compiling source code, and doing backups. |
|  | **Preventive** | Ordinary activities in this type perform changes to enhance maintainability and establish a base for making a future transition to an emerging technology. |
|  | **Performance** | Activities in performance type produce results that impact the user. Those activities improve system up time and replace components and algorithms with faster ones. |
|  | **Adaptive** | Ordinary activities in this type port the software to a different execution platform and increase the utilization of COTS components. |
|  | **Reductive** | Ordinary activities in this type drop some data generated for the customer, decreasing the amount of data input to the system and decreasing the amount of data produced by the system. |
|  | **Corrective** | Ordinary activities in this type are correcting identified bugs, adding defensive programming strategies and modifying the ways exceptions are handled. |
|  | **Enhancive** | Ordinary activities in this type are adding and modifying business rules to enhance the system’s functionality available to the customer and adding new data flows into orout of the software. |

Figure 2.1 illustrates the relationship among the different types of activities in terms of impacts.

The classification is based on modifications performed in various physical and conceptual entities:

* A—the whole software system.
* B—the program code.
* C—the functionalities experienced by customers

The idea of objective evidence about activities is key to making the classification. The fundamental acts of observation of behavior and comparison of behavior of two entities reveal the evidence, and evidence of change-producing activities serves as the criteria. Note that observation is performed on the artifacts and the activities operating on them. On the other hand, comparison is performed on the relevant parts of the software before and after a maintenance activity is performed. Activities are classified into different types by applying a two-step decision process:

* First, apply criteria-based decisions to make the clusters of types.
* Next, apply the type decisions to identify one type within the cluster.

Figure 2.2 summarizes the aforementioned two steps in a hierarchical manner. The decision tree shown in Figure 2.2 is an objective evidence-based classification of maintenance types. The three-criteria decision involving A, B, and C lead to the right types of cluster. The decisions characterize the types within each cluster. In a maintenance process, we are interested in the impacts of maintenance activities not only on the software but also the business processes of the software. One reads Figure 2.2 by beginning on the left-hand side and moving toward the right for increasing impact on the software and/or the business processes. First, clusters of maintenance types are identified by using the type (namely, A, B, and C) decisions. Next, the types within a cluster are identified by using type decisions. To make type decisions, one asks questions about a specific evidence. A type is only applicable if the answer to the type decision question is “Yes.” Specific type decision questions are in Table 2.3.



**TABLE 2.3 Summary of Evidence-Based Types of Software Maintenance**

|  |  |  |  |
| --- | --- | --- | --- |
| # | Criteria | Type Decision Question | Type |
|  | A-1 | To train the stakeholders, did the activities utilize the software as subject? | Training |
|  | A-2 | As a basis for consultation, did the activities employ the software? | Consultive |
|  | A-3 | Did the activities evaluate the software? | Evaluative |
|  | B-1 | To meet stakeholder needs, did the activities modify the noncode documentation? | Reformative |
|  | B-2 | To conform to implementation, did the activities modify the noncode documentation? | Updative |
|  | C-1 | Was maintainability or security changed by the activities? | Groomative |
|  | C-2 | Did the activities constrain the scope of future maintenance activities? | Preventive |
|  | C-3 | Were performance properties or characteristics modified by the activities? | Performance |
|  | C-4 | Were different technology or resources used by the activities? | Adaptive |
|  | D-1 | Did the activities constrain, reduce, or remove some functionalities experienced by the customer? | Reductive |
|  | D-2 | Did the activities fix bugs in customer experienced functionality? | Corrective |
|  | D-3 | Did the activities substitute, add to, or expand some functionalities experienced by the customers? | Enhancive |

Next, we explain the four kinds of clusters, namely, *support interface*, *documentation*, *software properties*, and *business rules* one by one.

***Support Interface Cluster*** This cluster relates to the modifications on how service and/or maintenance personnel interact with stakeholders and customers. The support interface cluster is invoked if the answer to the A criteria decision question “Was software changed?” is a “No.” It consists of maintenance type decision in the order of A-1, A-2, and A-3, because of their increasing impact. The default type here is *Evaluative*. In the following, we explain the three type decisions one by one.

* *Type decision A-1.* “To train the stakeholders, did the activities utilize the software as subject?” is the A-1-type decision. In the training type, common activities include: (i) in-class lessons for customers and (ii) a variety of training spanning from on-site to web-based training using training materials from the documentations. The idea is to provide training to the stakeholders in the details of the system that has been implemented by the software.
* *Type decision A-2.* This type decision is “As a basis for consultation, did the activities employ the software?”, and it is of *consultive* type. This type decision is commonly performed as it involves such activities as estimating the cost of the planned maintenance task, providing support from a help desk, helping customers in preparing a CR (change request), and making specific knowledge about the system or resources available to others in the organization.
* *Type decision A-3.* This decision, which is of *evaluative* type, is “Did the activities evaluate the software?”. A-3 is a commonly used decision as it includes the following activities: searching, examining, auditing, diagnostic testing, regression testing, understanding the software without modifying it, and computing different types of metrics.

***Documentation Cluster*** For the A criterion decision, if assessment of the objective evidence is “Yes,” then it implies that the software was modified. Next, we analyse the B decision, which is about source code. The documentation cluster is invoked by a “No” answer to the B criterion question “Did the activities change the code?” It concerns modifications in the documentation except source code. The cluster comprises two decisions, namely, B-1 and B-2. Documentation cluster activities normally appear after the software interface cluster.

* *Type decision B-1.* The B-1 decision is “To meet stakeholder needs, did the activities modify the noncode documentation?” Ordinary activities in *reformative* type improve the readability of the documentation, change the documentation to incorporate the effects of modifications in the manuals, prepare training materials, and change the style of the documentation for noncode entities. In other words, it involves reformulation of documentation for noncode entities by modifying its style while preserving the code.
* *Type decision B-2.* The B-2 decision is “To conform to implementation, did the activities modify the noncode documentation?” This *updative* type involves activities for replacing out-of-date documentation with up-to-date documentation, making semi-formal models to describe current source code, and combining test plans with the documentation, without modifying the code. Out of the two types, the default type is *update*.

***Software Properties Cluster*** The code is said to be modified if the B criterion decision produces a “Yes” outcome. Next, one analyzes decision C which queries about modifications in the functionality of the system observed by the user. The software properties cluster is invoked by a “No” answer to the C criteria decision “Did the activities change the customer-experienced functionality?” This cluster comprises four type decisions, namely, C-1, C-2, C-3, and C-4, with increasing impact in that order. The cluster concerns modifications in the attributes of the software without involving modifications in the functionality delivered by the software. Activities in this group commonly follow the documentation cluster and the support interface cluster. The default type here is *adaptive*, and the details of the type decisions in this cluster are as follows:

* *Type decision C-1.* “Was maintainability or security changed by the activities?” is theC-1-type decision, and it is of *groomative* type. The decision involves activities for source code grooming, such as substituting algorithms and modules with better ones, modifying conventions for data naming, making backups, changing access authorizations, altering code understandability, and recompiling the code.
* *Type decision C-2.* C-2 is a *preventive* type decision asking the question “Did the activities constrain the scope of future maintenance activities?” This type of activities makes modifications to the code without changing either the existing functionality experienced by the customers or the resources utilized or the existing technology. The impacts of such activities are generally not visible to the customer. The common activities are making changes to improve maintainability.
* *Type decision C-3.* “Were performance properties or characteristics modified by the activities?” is a C-3-type decision, and it is of *performance* type. This involves improving system up time, substituting algorithms and modules with the ones with better efficiency, reducing the demand for storage, and improving the system’s robustness and reliability. The customer often observes the changes in those properties.
* *Type decision C-4.* “Were different technology or resources used by the activities?” is a C-4-type decision, and it is an *adaptive* type. This type includes activities such as porting the software to a new execution platform, increased utilization of commercial off-the-shelf (COTS), changing the supported communication protocols, and moving to object-oriented technologies. Those activities can change customer-perceivable system properties, but similar to C-1, C-2, and C-3, type C-4 does not modify the functionality experienced by the customers. Type C-4 is the default in this group.

***Business Rules Cluster*** This cluster is invoked by a “Yes” answer to the C criteria decision question “Did the activities change the customer-experienced functionality?” This cluster comprises the D-1, D-2, and D-3-type decisions. These types of activities occur most frequently, and activities from other clusters are needed to support these activities. This cluster involves the user- and business-level functionalities.

* *Type decision D-1.* Type decision D-1 is “Did the activities constrain, reduce, or remove some functionalities experienced by the customers?” and it is of *reductive* type. This type of activities delete portions or all of the modules to constrain or remove some business rules. When organizations merge, their business rules undergo such actions as elimination, restriction, and reduction.
* *Type decision D-2.* Type decision D-2 is “Did the activities fix bugs in customer experienced functionality?” The major tasks fix defects, introduce defensive programming, and modify the ways exceptions are handled.
* *Type decision D-3.* Type decision D-3 is “Did the activities substitute, add to, or expand some functionalities experienced by the customers?,” and it is of *enhancive* type. This type implements modifications by enhancing the business rules to support more functionalities of the system. The major tasks are to add new subsystems and algorithms and modify the current ones to enhance their scope. The changes may affect customer experience of system functionality. D-3 is the default type in this group.

**Example:** A maintenance engineer, after analyzing all the documentation along with the program code, modified the program code for one component without modifying other documentation, built the rewritten component, executed the regression test suite, checked it into the version control, and embedded it into the production system. The only consequence the customer observed was improved latency. Question: Identify the type of software maintenance performed by the engineer.

From the activities reported it is apparent that criteria A and B evaluate to “Yes,” whereas criterion C evaluates to “No.” The given evidence leads to a “Yes” decision for the *performance* type in the Software Properties cluster. In addition, the evidence leads us to the *evaluative* type in the cluster Support Interface. In Figure 2.1, we identify the two types, namely, *performance* and *evaluative*, and note that the first type is higher up than the second one. Because *performance* type is higher up than *evaluative* type, one expects evidence of the *consultive, training, reformative, updative, preventive,* and *groomative* type, but not of the *adoptive, enhancive, corrective,* or *reductive* types. Therefore, *performance* is the dominant maintenance type for this example.

## CATEGORIES OF MAINTENANCE CONCEPTS

The domain concepts that influence software maintenance process can be classified into four categories (Fig. 2.3):

* the product to be maintained;
* the types of maintenance to be performed;
* the maintenance organization processes to be followed; and
* the peopleware involved, that is, the people in the maintenance organization and in the customer/client organization.



*Product*. A product is a coherent collection of several different artifacts. Source code is the key component of a software product. Other artifacts of interest include print manuals and online help.

*Product upgrade*. Baseline is an arrangement of related entities that make up a particular software configuration. Any change or upgrade made to a software product relates to a specific baseline. Note that an upgrade can create a new version of the system being maintained, a patch code for an object, or even a notice explaining a restriction on the use of the system. A restriction notice can be a release note saying that the product may not work with a specific version of a hardware.

*Artifact*. A number of different artifacts are used in the design of a software product and, similarly, a number of artifacts simultaneously exist along side a software product. One can find the following types of artifacts: textual and graphical documents, component off-the-shelf products, and object code components. Textual documents are readily understood by human readers: requirements specifications, plans, designs, and source code listings.

The key elements of the *maintained product* are size, age, application type, composition, and quality. The key characteristics of the aforementioned elements affecting maintenance performance are explained below.

The *size* of a software system affects the number of personnel needed to maintain the system. A small-size product can be maintained by a single person, whereas a medium-size product needs a team. However, for a large product one may need multiple maintenance teams. Maintenance activities on relatively large systems are generally less efficient than activities on small systems, because in a large product it is more difficult to: (i) conduct root cause analysis of some problems and (ii) identify ripple effects on the modules to support large enhancements.

The age of a software product, also known as *software geriatrics*, is the number of calendar years elapsed since its first release. The age of a software product can affect maintenance activities in various ways. It is difficult to maintain too old products for the following reasons: (i) it is difficult to find maintenance personnel for too old products, because of changes in development technologies; (ii) finding tools, namely, compilers and code analyzers, for very old systems is difficult; and (iii) the original or up-to-date development documentation may not be available for old products.

Knowledge about the *application domain* influences productivities of software development and maintenance activities. For example, a computer engineer will have stronger knowledge of IP (Internet Protocol) networking than an aerospace engineer. Hence, a computer engineer will be more productive in maintaining an IP network, whereas an aerospace engineer will be more productive in maintaining a software product to design airplane wings. In addition, application domains put constraints on maintenance of products and artifacts. For example, while maintaining a safety critical system, such as air traffic control, one must preserve—and, even increase—the product’s reliability. On the other hand, in another application domain the concept of time-to-market may cause early deployment of a newer version of a system. Therefore, different application domains affect the same aspect of maintenance to different degrees.

The level of abstraction of the *product component* determines the skills required by the maintenance personnel and the tools they need to support the component. If the product has been derived from an in-house design, the maintenance personnel need access to the Computer-Aided Software Engineering (CASE) tools used by the software developers. On the other hand, if the product is composed of COTS components obtained from third parties, the maintenance engineers need integration and acceptance testing skills, and the required skills include development of supporting components such as *wrapper, glue,* and *tailoring*.

The *product quality* initially delivered to the customer places constraints on the subsequent maintenance activities. Intuitively, good quality artifacts are easier to maintain than poor quality ones. In the absence of communication between maintenance personnel and the original developers, quality of the product essentially determines the level of difficulty of maintaining the product. Documentation is often poor or even non-existent for old products so maintenance personnel need specialized tools to reengineer the system.

*Activity*. A number of different broad classes of maintenance activities are performed on software products, including investigation, modification, management, and quality assurance. An activity may be composed of several smaller sub-activities. Usually, an activity accepts some artifacts as inputs and produces new or changed artifacts. In the following, we briefly explain the four kinds of activities.

*Investigation activity*. This kind of activities evaluate the impact of making a certain change to the system.

*Modification activity*. This kind of activities change the system’s implementation.

*Management activity*. This kind of activities relate to the configuration control of the maintained system or the management of the maintenance process.

*Quality assurance activity*. This kind of activities ensure that modifications performed on a system do not damage the integrity of the system. For example, regression testing is an example of quality assurance activities.

*Service-level agreement (SLA)*. This is a contract between the customers and the providers of a maintenance service. Performance targets for the maintenance services are specified in the SLA.

*Maintenance management*. This process is used to manage the maintenance service, which is not the same as managing individual CRs. An organization process is set up and run by the senior management. They create a structure of the maintenance team so that service-level agreement can be executed. In addition to fulfilling the roles of regular processes, such as project management and quality assurance, maintenance management handles events, change control, and configuration control.

*Event management*. The stream of events, namely, all the CRs from various sources, received by the maintenance organization, is handled in an event management process.

*Change control*. Evaluation of results of investigations of maintenance events is performed in a process called change control. Based on the evaluation, the organization approves a system change.

*Configuration management*. A system’s integrity is maintained by means of a configuration management process. Integrity of a product is maintained in terms of its modification status and version number.

*Maintenance organization structure*. This is the hierarchy of roles assigned to maintenance personnel to perform administrative tasks.

*Maintenance event*. This is a problem report or a CR originating from within the maintenance organization or from the customers.

*Investigation report*. This is the outcome of assessing the cause and impacts of a maintenance event.

After an initial investigation of a CR, a management process is put in place for approving change activities. Approval of a CR is normally the responsibility of a change control board. A change control board is organized as a formal process with meetings between maintenance managers, clients, users, and customers. A proposed modification activity is scheduled only after the modification is approved by the board and an SLA is signed with the client. The level of formality adopted in change control board procedures can affect the quality and efficiency of performing changes. A formal change control board generally slows down the maintenance process but is better at protecting the integrity of the systems being maintained.

Maintenance activities cannot ignore the human element, because software production and maintenance are human-intensive activities. The three people-centric concepts related to maintenance are as follows:

*Maintenance organization*. This is the organization that maintains the product(s).

*Client organization*. A client organization uses the maintained system and it has a clear relationship with the maintenance organization. The said relationship is described in the SLA.

*Human resource*. Human resource includes personnel from the maintenance and client organizations. Maintenance organization personnel include managers and engineers, whereas client organization personnel include customers and users. The management negotiates with the customers to find out the SLA, scheduling of requirement enhancements, and cost.

The following user and customer issues affect maintenance:

* *Size.* The size of the customer base and the number of licenses they hold affect the amount of effort needed to support a system.
* *Variability.* High variability in the customer base impacts the scope of maintenance tasks.
* *Common goals.* The extent to which the users and the customer have common goals affects the SLAs. Ultimately, customers fund maintenance activities. If the customers do not have a good understanding of the requirements of the actual users, some SLAs may not be appropriate to the end users.

## EVOLUTION OF SOFTWARE SYSTEMS



* *Maintenance* means preserving software from decline or failure.
* *Evolution* means a continuously changing software from a worse state to a better state.

Software evolution is studied with two broad, complementary approaches, namely, *explanatory* and *process improvement*, and those describe the *what* and *how* aspects, respectively, of software evolution.

* *Explanatory (what/why).* This approach attempts to explain the causes of software evolution, the processes used, and the effects of software evolution. The explanatory approach studies evolution from a *scientific* view point. In this approach, the *nature* of the evolution phenomenon is studied, and one strives to understand its driving factors and impacts.
* *Process improvement (how).* This approach attempts to manage the effects of software evolution by developing better methods and tools, namely, design, maintenance, refactoring, and reengineering. The process improvement approach studies evolution from an *engineering* view point. It focuses on the more pragmatic aspects that assist the developers in their daily routines. Therefore, methods, tools, and activities that provide the means to direct, implement, and control software evolution are at the core of the process improvement approach.

### SPE Taxonomy

The abbreviation SPE refers to S (Specified), P (Problem), and E (Evolving) programs.

**S-type programs:** S-type programs have the following characteristics:

* All the non-functional and functional program properties that are important to its stakeholders are *formally* and *completely* defined.
* Correctness of the program with respect to its formal specification is the *only* criterion of the acceptability of the solution to its stakeholders.

A formal definition of the problem is viewed as the specification of the program. S-type programs solve problems that are fully defined in abstract and closed ways. Examples of S-type programs include calculation of the lowest common multiple of two integers and to perform matrix addition, multiplication, and inversion. If the real world changes, the original problem turns into a completely *new problem* that must be re-specified. But then it has a *new program* to provide a solution. It may be possible to derive a new program from the old one, but it is a *different* program that defines a solution to a *different* problem.



In the real world, S-type systems are rare. However, it is an important concept that evolution of software does not occur under some conditions.

**P-type programs:** With many real problems, the system outputs are accurate to a constrained level of precision. The concept of correctness is difficult to define in those programs. Therefore, approximate solutions are developed for pragmatic reasons. Numerical problems, except computations with integers and rational numbers, are resolved through approximations. For example, consider a program to play chess. Since the rules of chess are completely defined, the problem can be completely specified. At each step of the game a solution might involve calculating the various moves and their impacts to determine the next best move. However, complete implementation of such a solution may not be possible, because the number of moves is too large to be evaluated in a given time duration. Therefore, one must develop an approximate solution that is more practical while being acceptable.



Note that the program resulting from the changes cannot be considered a new solution to a new problem. Rather, it is a modification of the old solution to better fit the existing problem.

**E-type programs:** An E-type program (Fig. 2.7) is one that is embedded in the real world and it changes as the world does. These programs mechanize a human or society activity, make simplifying assumptions, and interface with the external world by requiring or providing services. An E-type system is to be regularly adapted to: (i) stay true to its domain of application; (ii) remain compatible with its executing environment; and (iii) meet the goals and expectations of its stakeholders.



For E-type programs, the concept of correctness is left up to the stakeholders. E-type program execution changes its operational domain, and the evolution process is viewed as a feedback system (Figure 2.8).



***Software evolution obeys Lehman laws considered in Lecture 1 Introduction,***

There are two types of aging in software life cycles: software process: execution aging and software product aging. The first one manifests in degradation in performance or transient failures in continuously running the software system. The second one manifests in degradation of quality of software code and documentation due to frequent changes. Aging-related *symptoms* in software are:

* *Pollution.* Pollution means that there are many modules or components in a system which are not used in the delivery of the business functions of the system.
* *Embedded knowledge.* Embedded knowledge is the knowledge about the application domain that has been spread throughout the program such that the knowledge cannot be precisely gathered from the documentation.
* *Poor lexicon.* Poor lexicon means that the component identifiers have little lexical meaning or are incompatible with the commonly understood meaning of the components that they identify.
* *Coupling.* Coupling means that the programs and their components are linked by an elaborate network of control flows and data flows

### 3.2. Practical Implications of the Lehman’s Laws

Based on the eight laws, Lehman suggested more than 50 rules for management, control, and planning of software evolution.

*Assumptions management.* Several assumptions are made by different personnel involved throughout the life cycle of a project. When a software project fails, the primary source of failure can be traced back to those assumptions. It is generally found that some of those assumptions were never valid in the first place, or it is more likely that some of the assumptions became invalid as a result of changes outside the software system. Therefore, management of assumptions plays a key role in successful execution of projects involving E-type software. The following is a list of activities for managing assumptions.

* Identify and capture the assumptions pertinent to the project. The difficulty lies in completely identifying all the assumptions.
* Initiate periodic reviews to assess any need to correct or update the list of assumptions.
* Review and revalidate the assumptions whenever a change occurs in the specification, design, implementation, or operational domain.
* Where software operates in a rapidly changing environment, complement detailed assumption review process with re-writing of appropriate components of the software.
* Develop and use tools to track all the above activities.

*Evolution management.*

* Consistently assess and pursue antiregressive work such as complexity control, restructuring, and full documentation. The phrase antiregressive work means the work to be performed to reduce a program’s complexity with no modifications to the user perceived functionality delivered by the system. As part of the development and maintenance responsibility, carry out antiregressive activities. This may not have an immediate impact on stakeholders, but this will facilitate future evolvability.
* Ensure that documentation includes identification and recording of assumptions.
* Assess the trends in the evolutions of the functional and non-functional requirements of the software product in advance. Review those trends during the release planning while taking the operational domain into consideration.
* Involve application and operational domain specialists in the assessment.
* Use tools to support data collection, modeling, and related activities.
* Acquire, plot, model, and interpret historical evolution metrics to project trends, patterns, growth, and their rate of changes in order to improve planning and processes.
* When validating incremental growth, assess the impact on the unchanged parts of the system and assumptions.
* Establish baselines of key measures over time to support evolution and maintenance planning and control.

*Release management.* A software release can be categorized as *safe, risky*, or *unsafe* according to the condition described as follows. Let *m* be the mean of the incremental growth *mi* of the system in going from release *i* to release *i* + 1 and *s* be the standard deviation of the incremental growth. The release is *safe* if the content of the *i*th desired release (say, *mi*) is less than or equal to *m*. The release is said to be *risky* if the content of the desired release is greater than *m* but less than *m* + 2*s*. Finally, the release is *unsafe* if the content of the desired release is close to or greater than *m* + 2*s*. Based on the aforementioned concepts of safety, concrete activities for release management are as follows:

* Ensure that the release is *safe*.
* When the release is not *safe*, then distribute the growth across several releases to make individual releases safe
* If excessive functional increments are unavoidable, plan for follow-on clean-up releases with a focus on fixing defects and updating documentation
* Follow established software engineering principles, namely, information hiding to minimize spread of changes between system elements
* By allocating resources, put emphasis on antiregressive work, namely, restructuring, eliminating dead code, and reengineering
* Consider the alternation of enhancement and extension with clean-up and restructuring releases.

## MAINTENANCE OF COTS-BASED SYSTEMS

Component-based development has an intuitive underlying idea. Instead of developing a system by programming it entirely from scratch, develop it by using pre-existing building blocks, components, and plug them together as required to build the target system. Components are nearly independent and replaceable parts of a system. Special components called Commercial-off-the-shelf (COTS) can be purchased on a component market. Often, these types of components are delivered without their source code. The use of COTS components is increasing in modern software development because of the following reasons: (i) there is significant gain in productivity due to reusing commercial components; (ii) the time to market the product decreases; (iii) the product quality is expected to be high, assuming that the COTS components have been well tested; and (iv) there is efficient use of human resources due to the fact that development personnel will be freed up for other tasks. However, many difficulties are to be overcome while using COTS compared to using in-house components. The black-box nature of COTS components prevents system integrators from modifying the components to better meet user’s requirements. Moreover, the integrators have no visibility of the details of the components, their evolutions, or their maintenance; rather, they are solely dependent on the developers and suppliers of the components. The only source code being written and modified by the integrators is what is needed for integrating the COTS-based systems. This includes code for tailoring and wrapping the individual components, as well as the “glue” code required to put the components together. Wrapper code X combined with another piece of code Y determines how code Y is executed. The wrapper acts as an interface between its caller and the wrapped code Y. Wrapping may be done for compatibility. Irrespective of a software system being in-house developed or COTS based, maintenance is the most expensive phase of the system’s life cycle. There are key differences in the activities executed to maintain component-based software (CBS), even though the motivations behind system maintenance remain the same. The differences are due to the following major sources:

* Maintainers perceive a CBS system as an interacting group of large-scale black-box components, instead of a compiled set of source modules. The two views require different maintenance skill sets.
* Most of the source code implementing the wrapper, glue, and tailoring modules are used to integrate the system, instead of delivering services and functions.
* The maintenance organization largely loses control over the precise evolution of the system, because COTS developers focus on their own business interests.

The difficulties of CBS maintenance are due to:

* Frozen functionality
* Incompatibility of upgrades
* Trojan horses
* Unreliable COTS components
* Defective middleware.

*Frozen functionality.* The functionalities of a COTS component are rendered to be frozen when the vendor stops enhancing the product or stops providing further product support. This occurs if the vendor or the supplier discontinues to support the component. The host system becomes unmaintainable due to the components becoming frozen. The host organization will have a serious problem if periodic updates are required to be performed on those components.

*Incompatibility of upgrades.* The host organization integrates the components and upgrades the software product to meet the needs of its customers. If a modified component becomes inconsistent with the remaining components of the host system, then integration of the modified component with the host system may not be possible. For example, the new version of a component may require new data formats, which, in turn, requires modifications to be done to the contents and formats of the current files that were generated by earlier versions of the COTS software. The problem then becomes similar to the frozen functionality problem.

*Trojan horses.* A software Trojan horse is a piece of code that has been programmed into a component to make it behave in a malicious way. For instance, deleting all files after switching to a privileged directory can be considered an example of Trojan horse functionality. Determining a functionality to be a Trojan horse is a difficult task. Making the Trojan horse dynamically context sensitive is one way to hide it. For instance, “delete all files” can be a valid command if it refers to entities in a temporary directory. On the other hand, it can have devastating consequences if it is executed in a system context. Therefore, deleting system files can be classified as Trojan horse behavior, whereas deleting temporary files is a normal function.

*Unreliable COTS components.* Today, no uniform standard exists to test software components to certify their reliability. By paying software certification laboratories (SCL) to grant software certificates, independent vendors partially shift their responsibility to the SCL. However, the dependability of a component is not known to its customers. Even if a score for dependability is provided by the vendor, it is likely that the score was computed based on intricacies that do not broadly reflect the customer’s execution environment.

*Defective middleware.* COTS components are primarily integrated by analysing the syntax and semantics of their interfaces. However, the integrators have several means for integrating COTS components into a host system, and designing middleware is a straightforward approach. Whenever concerns exist regarding the behaviour of a COTS component in the context of a whole system, it is prudent to write middleware to ensure that certain constraints are satisfied. For example, wrappers are a kind of middleware which can be used to constrain the functionalities of components. The main ideas in wrapper design is to: (i) restrict the inputs; (ii) perform pre-processing on the inputs; (iii) restrict the outputs of a component; or (iv) perform post-processing on the outputs. All those kinds of processing have the potential of modifying the semantics of a component. The key problem in designing wrappers is that it is not completely known what behavior to protect against.

**Maintenance activities** for CBS:

* **Component reconfiguration**. Component reconfiguration means adding, removing, and replacing components of a system.
* **Testing and debugging**: fixing defects in an in-house developed system typically involves: (i) executing the system with a debugger to locate the problem and (ii) modifying the source code to fix the defects. On the other hand, maintenance personnel cannot modify source code of COTS components. Rather, they become dependent upon the component vendors to understand the internal details of the product. Consequently, maintenance personnel and COTS vendors frequently exchange detailed messages
* **Monitoring of systems**: Monitoring for the purpose of maintenance is a difficult task because of the low visibility of the internal operations of COTS software
* **Enhancing functionality for users**: In the absence of access to source code, tailoring is done by means of two techniques: 1) write additional glue code to hold the system together and provide enhanced functionality; and 2) use vendor supplied tailoring techniques to customize the products, because integration personnel have no access to program code.
* **Configuration management**: The following five configuration management activities are specifically done for CBS products:
1. Track the versions of the COTS products, and retain the following details for each component in the version archive:
* Save the name of the developer of the component if available.
* Save the contact information of the person or organization supplying the component.
* Archive the source code of the component if available.
* Archive the working versions of all tools, namely, compilers and linkers, necessary to rebuild a component.
* Make a detailed rationale for including the component in the system, including any previous use of the component and known facts about its quality attributes. For instance, BSD Unix was used by Sun Microsystems to build their proprietary OS. Therefore, the information “BSD Unit” is an instance of “previous use.”
* Obtain the contact information of some of those using the component.
1. Perform configuration management on the individually tailored COTS elements.
2. Track the configuration history of a product at all its deployment sites.
3. Find the compatible versions of the various COTS elements.
4. Manage support and licenses for each COTS element.