2

TAXONOMY OF SOFTWARE MAINTENANCE AND EVOLUTION

Evolution is not a force but a process. Not a cause but a law.

—John Morley

2.1 GENERAL IDEA

In the early 1970s, the term "maintenance" was used to refer to tasks for making intentional modifications to the existing software at IBM. Those who performed maintenance tasks had not carried out the software development work. The idea behind having a different set of personnel to carry out maintenance work was to free the development engineers from support activities. The aforementioned model continued to influence the activities that are collectively known as "software maintenance." In circa 1972, in his landmark article "The Maintenance 'Iceberg'," R. G. Canning [1] used the iceberg metaphor to describe the enormous mass of potential problems facing practitioners of software maintenance. Practitioners took a narrow view of maintenance as correcting errors and expanding the functionalities of the system. In other words, maintenance consisted of two kinds of activities: correcting errors and enhancing functionalities of the software. Hence, maintenance can be inappropriately seen as a continuation of software development with an extra input—the existing software system [2].

The ISO/IEC 14764 standard [3] defines software maintenance as "… the totality of activities required to provide cost-effective support to a software system. Activities

Software Evolution and Maintenance: A Practitioner's Approach, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.

are performed during the pre-delivery stage as well as the post-delivery stage" (p. 4). 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" (p. 3).

A major difference exists between software maintenance and software development: maintenance is event driven, whereas development is requirements driven [4]. A process for software development begins with the objective of designing and implementing a system to deliver certain functional and nonfunctional 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. Events originate from both the customers and from within the developed organization. Generally, the inputs that invoke maintenance activities are unscheduled events; execution of the actual maintenance activities might be scheduled according to a plan, but the events that initiate maintenance activities occur randomly. A maintenance activity accepts some existing artifacts as inputs and generates some new and/or modified artifacts.

Now we further explain the idea of a maintenance activity taking in an input and producing an output. 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 subsections, we explain maintenance activities 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.

2.1.1 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 [5–7]. Based on the Standard for Software Engineering–Software Maintenance, ISO/IEC 14764 [3], 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. A more concrete example is: an application software on a smartphone can be enhanced to support WiFi-based communication in addition to its present Third Generation (3G) cellular communication.

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" [8].

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 [9]. 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 [10–13]. 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." (p. 1347 of Ref. [14]). 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.

2.1.2 Activity-Based Classification of Software Maintenance

In the intention-based classification of maintenance activities, the intention of an activity depends upon the reason for the change [7]. On the other hand, Kitchenham et al. [4] organize maintenance modification activities based on the maintenance activity. The authors classify the maintenance modification activities 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 Swanson's terminology and Kitchenham's terminology. Enhancement activities which are necessary to change the implementations of existing requirements are similar to Swanson's idea of perfective maintenance. Enhancement activities, which add new requirements to a system, are similar to the idea of adaptive maintenance. Enhancement activities which do not impact requirements but merely affect the system implementation appear to be similar to preventive maintenance.

2.1.3 Evidence-Based Classification of Software Maintenance

The intention-based classification of maintenance activities was further refined by Chapin et al. [15]. The objectives of the 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 proposed by Chapin [16] was exhaustive in nature. His mutually exclusive types were grouped into clusters as illustrated in Figure 2.1. The definitions of the 12 types of maintenance activities are given in Table 2.1.

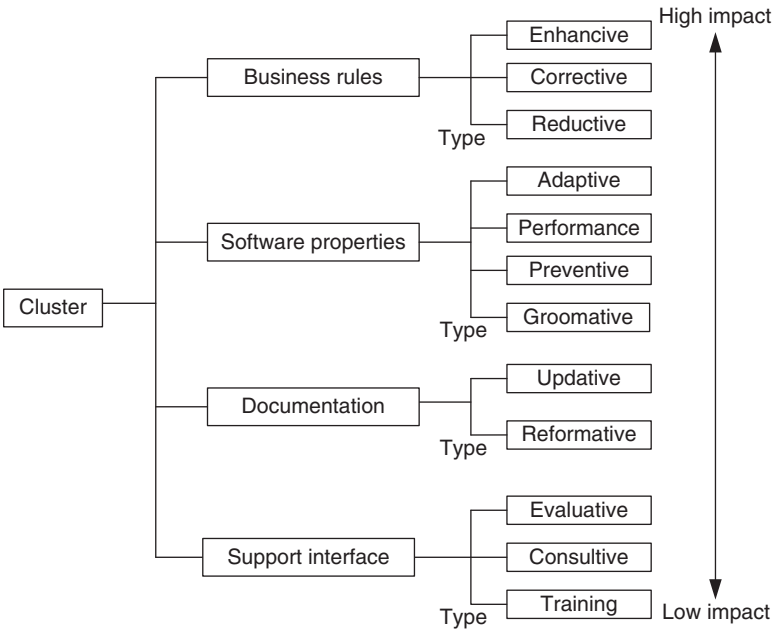

FIGURE 2.1 Groups or clusters and their types

TABLE 2.1 Evidence-Based 12 Mutually Exclusive Maintenance Types

Types of Maintenance	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 or out of the software.

TABLE 2.2 Impact of the Types

Impact on Software	Impact on Business Low ← – – – – – – → High	Cluster and Type
Low		**Support interface**
↑	◇ ◇ ◇ ◇ ◇	Training
|	◇ ◇ ◇◇	Consultive
|	◇◇	Evaluative
|		**Documentation**
|	◇◇	Reformative
|	◇◇	Updative
|		**Software properties**
|	◇◇	Groomative
|	◇ ◇ ◇	Preventive
|	◇ ◇ ◇	Performance
|	◇◇	Adaptive
|		**Business rules**
|	◇	Reductive
|	◇ ◇ ◇	Corrective
↓	◇ ◇ ◇ ◇ ◇◇	Enhancive
High		

Source: From Reference 15. © 2001 John Wiley & Sons.

The impacts of the different types or clusters of maintenance activities are summarized in Table 2.2. The first dimension of the impact of the evolution is the customer's ability to perform its business functions effectively while continuing to use the system. The impact is represented as low or high. The number of diamonds in a row indicates the more probable range of impact on the business process of the customer. As an example, if the software is enhanced with new functionalities, then the customer is more likely to be able to achieve its business goals than modifications on noncode documentation. The second dimension is the software itself. This is arranged from top to bottom. As an example, rewriting a few pages of a user's manual has almost no impact on the software compared to rewriting a block of code to fix a defect. Figure 2.1 illustrates the relationship among the different types of activities in terms of impacts.

Decision Tree-Based Criteria The classification of maintenance activities have been illustrated in Figure 2.1. The classification is based on modifications performed —modifications deliberately done, modifications observed, modifications manifested, and modifications detected—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. Due to the impact characteristics summarized in Table 2.2, 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

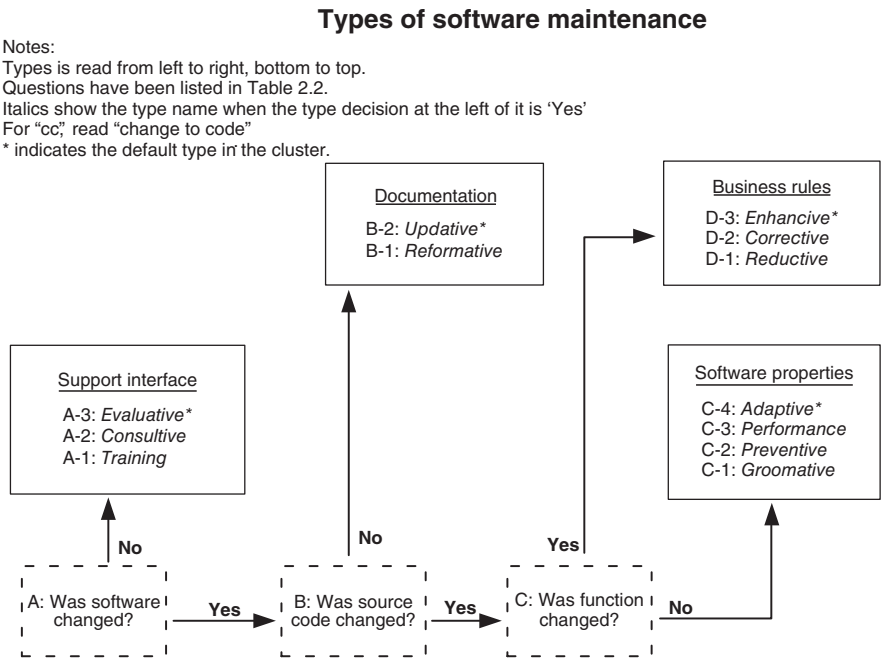

FIGURE 2.2 Decision tree types. From Reference 15. © 2001 John Wiley & Sons

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

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." For specific type decision questions, the reader is referred to Table 2.3. Sometimes, an objective evidence may be found to be ambiguous. In that case, clusters have their designated default types for use. The overall default type is evaluative if there are ambiguities in an activity. Ambiguities can be present in observations and the available documentation evidence. Trivially, no type of software maintenance has been conducted if observations indicate that no activities are performed on the software, except merely executing it.

Since software maintenance involves many activities, a variety of questions are answered to determine their types, as illustrated in Figure 2.2, in the structure of a decision tree. All the three cluster decision criteria A, B, and C shown in the three dotted boxes of Figure 2.2 and, listed below as well, must be asked:

- A: Was software changed?
- B: Was source code changed?
- C: Was function changed?

A "No" response to any of the aforementioned questions leads to a cluster, where one or more type decisions result in an "Yes." As an example, if the answer to question

A is "No," then at least one question in the box labeled "Support interface" produces a "Yes" answer. In addition, if the answer to question C is "Yes," then at least one question in the box labeled "Business rules" produces a "Yes" answer. The basic idea is to move upward from the bottom of the rightmost column of leaves in Figure 2.1 as far up as possible. The aforementioned traversal is similar to traversing the tree shown in Figure 2.2. 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, 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 analyze 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 the C-1-type decision, and it is of *groomative* type. The decision involves "anti-regression" 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 [15].

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.

2.2 CATEGORIES OF MAINTENANCE CONCEPTS

In the previous section, we discussed different views of software maintenance. In this section, we describe some ways to organize maintenance activities. Organization of maintenance activities is conceptually similar to the organization of activities for software development. However, maintenance activities focus on product correction and adaptation, whereas development activities transform high-level requirements into working code. In this section, the key factors influencing the maintenance process are identified.

The domain concepts that influence software maintenance process can be classified into four categories:

- 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.

The four categories and the concepts that influence the maintenance process have been illustrated in Figure 2.3. The concepts in each category are defined to understand the relationships among maintenance concepts. Next, we describe the characteristics of these concepts and their impacts on maintenance activities. This categorization enables one to know to what degree methods, tools, and skills for maintenance differ from those for software development [4]. In the following subsections, we explain the four dimensions of maintenance as depicted in Figure 2.3.

2.2.1 Maintained Product

The *maintained product* dimension is characterized by three concepts: *product, product upgrade*, and *artifacts*. We explain the characteristics of these concepts and their impacts on maintenance process.

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

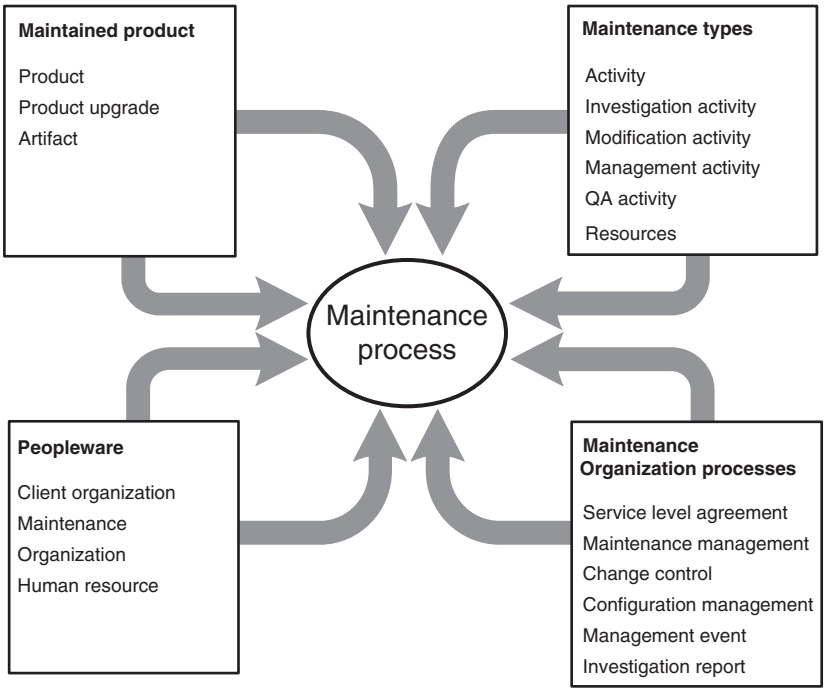

FIGURE 2.3 Overview of concept categories affecting software maintenance

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 [17, 18]. 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.

The life cycle stages of a software product have been listed in the first column of Table 2.4. The maintenance life cycle starts after the initial development and ends when the product is withdrawn from the market, as discussed in Section 3.3. The second column represents the corresponding maintenance tasks for each stage of maintenance. The third column represents typical size of user population associated with the product in that stage of its life cycle. It may be noted that large enhancements at the evolution stage can cause several patch releases at the servicing stage, considering the need for fixing faults (See Lehman's fifth law "conservation of familiarity.") Table 2.4 shows different types of maintenance tasks performed by an organization that are related to the maturity of a system.

Knowledge about the *application domain* influences productivities of software development and maintenance activities [19]. 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

TABLE 2.4 Staged Model Maintenance Task

Life Cycle Stage	Maintenance Task	User Population
Initial development	–	–
Evolution	Corrections, enhancements	Small
Servicing	Corrections	Growing
Phaseout	Corrections	Maximum
Closedown	Corrections	Declining

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 nonexistent for old products so maintenance personnel need specialized tools to reengineer the system.

2.2.2 Maintenance Types

We discussed different types of maintenance activities in Section 2.1. In this subsection, different types of maintenance activities are defined, followed by the impacts of those activities on maintenance performance.

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.

Maintenance personnel require different levels of understanding of the product by means of development tools so that they can execute the different maintenance modifications. For example, a corrective maintenance activity requires the ability to find the precise location of the faulty code and perform localized modifications. The maintenance engineers need to reproduce the bug in the test environment and may want to use tools to step through the suspected portion of code. On the other hand, an activity for quality or functionality enhancement requires a broad comprehension of large portions of the system. In addition, the maintenance engineer needs the development environment, such as version control, to check out and check in

the code. The maintenance engineer may require re-engineering tools if the documentation is poor. The size and priority of the change are important features that impact the productivity and efficiency of the activities. Specifically, an enhancement with a large scope will involve a large number of maintenance personnel and it will incur more communication and coordination overhead. The priority of a maintenance activity affects the length of time needed for the change to be delivered to clients.

The efficiency and quality of investigation activities depend upon the knowledge of the maintenance engineers possess about the current status of the release, outstanding issues, and any planned modifications about the portion of the system involved. The effectiveness of the configuration control mechanism and control of the change process impact the availability of those information. To identify the status of each system component, a good configuration control process is required.

2.2.3 Maintenance Organization Processes

Two different levels of maintenance processes are followed within a maintenance organization:

- the individual-level maintenance processes followed by maintenance personnel to implement a specific CR and
- the organization-level process followed to manage the CRs from maintenance personnel, users, and customers/clients; this higher level of process is referred to as the maintenance organization process.

The software maintenance concepts defined below are used to modify one or more artifacts to implement a CR:

Development technology. This refers to the technology that was used to develop the original system and its constituent artifacts. The development technology constraints the maintenance procedures.

Paradigm. This refers to the philosophy adopted at the time of developing the original system. Some examples are procedural paradigm and object-oriented paradigm. This too constraints the possible maintenance procedures.

Procedure. A procedure can be a method, a technique, or a script. From a set of procedures, one is chosen to perform a specific activity.

Method. A method is a general, systematic procedure giving clear steps and heuristics to implement some activities.

Technique. A technique is a less rigorously defined procedure to realize an activity.

Script. A script is a general guideline to construct or amend a specific type of document.

Artifacts include plans, documents, system representations, and source and object code items. Artifacts are created during the software development process and changed during maintenance. A multitude of different scripts, techniques, and methods are used to create and change such artifacts. The performance of maintenance activities are impacted by the selection of development paradigm and development technology. It is also impacted by the degree of automation of procedures. In general, development paradigm and development technologies put limitations on maintenance activities and skill requirements. For example, if the service-oriented architecture (SOA) paradigm is used for the software development process, then the maintenance engineers must be well versed with the SOA.

Next, we briefly define the concepts of the maintenance organization processes and then discuss the impact of these concepts on maintenance.

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.

Three major elements of a maintenance organization are event management, configuration management, and change control system, as explained before in this section. A maintenance organization handles maintenance requests from users, customers, and maintainers. If any of the three elements is not adequate, maintenance

will be inefficient and product quality is likely to be compromised. The efficiency of maintenance activities is highly influenced by support tools. There are many tools, namely, ManageEngine and Zendesk, to assist event management. The type and volume of CRs affect the performance of the maintenance organization. As an example, if many defects are reported in a short time, there may not be enough resources to carry out modifications.

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.

SLAs describe the maintenance organization's performance targets. It can be used by maintenance personnel as a guidance to meet customer's expectations. SLAs should be based on results rather than effort, and maintenance organizations must be prepared to meet their SLAs. In general, maintenance organizations use three different support levels to organize the staff:

- *Level 1*. This group files problem reports and identifies the technical support person who can best assist the person reporting a problem.
- *Level 2*. This level includes experts who know how to communicate with users and analyze their problems. These people recommend quick fixes and temporary workarounds.
- *Level 3*. This level includes programmers who can perform actual changes to the product software.

Note that not all maintenance works cause changes in the system. In many situations, users may need advice on how to continue to use the system or in what ways they can bypass a problem.

2.2.4 Peopleware

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.

In general, maintenance tasks are perceived to be less challenging, and, hence, less well rewarded than original work. Often maintenance tasks are partly assigned to newly recruited programmers, which has a significant impact on productivity and quality. A novice programmer may introduce new defects while resolving an incident because of an absence in understanding the whole system. Normally, more skilled maintenance personnel produce more and better quality works.

Separation between development staff and maintenance staff impacts the maintenance process. On one hand, there is no real separation between maintenance and development. In such a scenario, the product undergoes continual evolution. The developers incorporate maintenance activities into a continuing process for planned enhancements. In this case, the tools and procedures are the same for both the development and maintenance activities. On the other hand, there are maintenance organizations which operate with minimal interactions with the development departments. Occasionally, the maintenance group may not be located in the same organization that produced the software. In such a scenario, maintenance engineers may need specially designed tools. Maintenance managers need to focus on the aforementioned issues when signing SLAs. Finally, 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.

2.3 EVOLUTION OF SOFTWARE SYSTEMS

The term *evolution* was used by Mark I. Halpern in circa 1965 to define the dynamic growth of software [20]. It attracted more attention in the 1980s after Meir M. Lehman proposed eight broad principles about how certain types of software systems evolve [21–24]. Bennett and Rajlich [25] researched the term “software evolution,” but found no widely accepted definition of the term. However, some researchers and

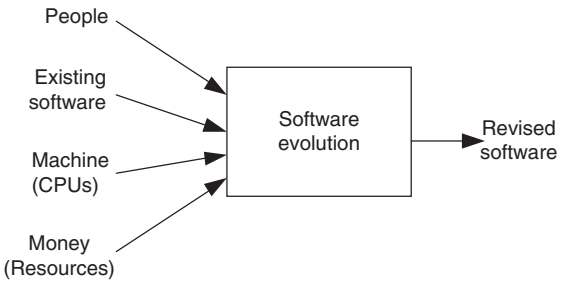

FIGURE 2.4 Inputs and outputs of software evolution. From Reference 26. © 1988 John Wiley & Sons

practitioners used the term software evolution as a substitute for the term software maintenance. Lowell Jay Arthur distinguished the two terms as follows:

- *Maintenance* means preserving software from decline or failure.
- *Evolution* means a continuously changing software from a worse state to a better state (p. 1 of Ref. [26]). Software evolution is like a computer program, with inputs, processes, and outputs (p. 246 of Ref. [26]) (See Figure 2.4).

Keith H. Bennett and Jie Xu [27] use "maintenance" for all post-delivery support, whereas they use "evolution" to refer to perfective modifications—modifications triggered by changes in requirements. Ned Chapin defines software evolution as:

> "the applications of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version, where the time period between versions may last from less than a minute to decades, together with the associated quality assurance activities and processes, and with the management of the activities and processes" (p. 21 of Ref. [15]).

The majority of software maintenance changes are concerned with evolutions triggered by user requests for changes in the requirements. The following are the key properties of software evolution as desired by the stakeholders:

- Changes are accomplished quickly and in a cost-effective manner.
- The reliability of the software should not be degraded by those changes.
- The maintainability of the system should not degrade. Otherwise, future changes will be more expensive to carry out.

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.

2.3.1 SPE Taxonomy

The abbreviation SPE refers to S (Specified), P (Problem), and E (Evolving) programs. In circa 1980, Meir M. Lehman [24] proposed an SPE classification scheme to explain the ways in which programs vary in their evolutionary characteristics. The classification scheme is characterized by: (i) how a program interacts with its environment and (ii) the degree to which the environment and the underlying problem that the program addresses can change. He observed a key difference between software developed to meet a fixed set of requirements and software developed to solve a real-world problem which changes with time. The observation leads to the identification of types S (Specified), P (Problem), and E (Evolving) programs. In the following, we explain the SPE concepts in detail.

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

- All the nonfunctional 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 [28]. The problem is completely defined, and there are one or more correct solutions to the problem as stated. The solution is well known, so the developer is concerned not with the correctness of the solution but with the correctness of the implementation of the solution. As illustrated in Figure 2.5, the specification directs and controls the programmers in creating the program that defines the desired solution. The problem statement, the program, and the solution may relate to a real world—and the real world can change. However, if the real world changes, the original problem turns into a completely *new problem* that must be respecified. But then it has a *new program* to provide a solution. It may be possible and time saving to derive a new program from

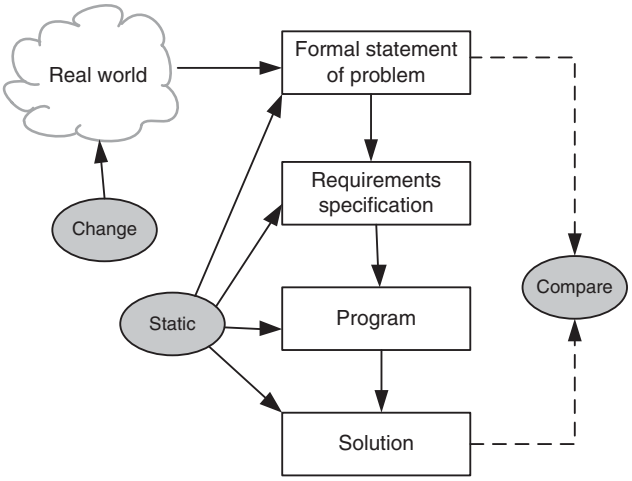

FIGURE 2.5 S-type programs

the old one, but it is a *different* program that defines a solution to a *different* problem. The program remains almost the same in the sense that it does not accommodate changes in the problem that generates it [29]. 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.

In order to develop this type of solution, we describe the problem in an abstract way and write the requirement specification accordingly. A program developed this way is of P-type because it is based on a practical abstraction of the problem, instead of relying on a completely defined specification. Even though an exact solution may exist, the solution produced by a P-type program is tampered by the environment in which it must be produced. The solution of a P-type program is accepted if the program outcomes make sense to the stakeholder(s) in the world in which the problem is embedded. As illustrated in Figure 2.6, P-type programs are more dynamic than S-type programs. P-type programs are likely to change in an incremental fashion. If the output of the solution is unacceptable, then the problem abstraction may

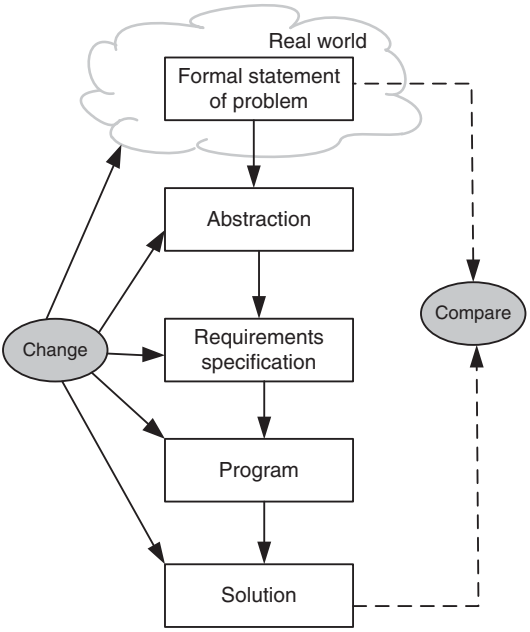

FIGURE 2.6 P-type programs

be changed and the requirements modified to make the new solution more realistic. 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. In addition, the real world may change, hence the problem changes.

E-type programs: An E-type program 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 [30].

Figure 2.7 illustrates the dependence of an E-type program on its environment and the consequent changeability. The acceptance of an E-type program entirely depends upon the stakeholders' opinion and judgment of the solution. Their descriptions cannot be completely formalized to permit the demonstration of correctness, and their operational domains are potentially unbounded. The first characteristic of an E-type program is that the outcome of executing the program is not definitely predictable. Therefore, for E-type programs, the concept of correctness is left up to the stakeholders. That is, the criterion of acceptability is the stakeholders' satisfaction with each program execution [31]. An E-type program's second characteristic

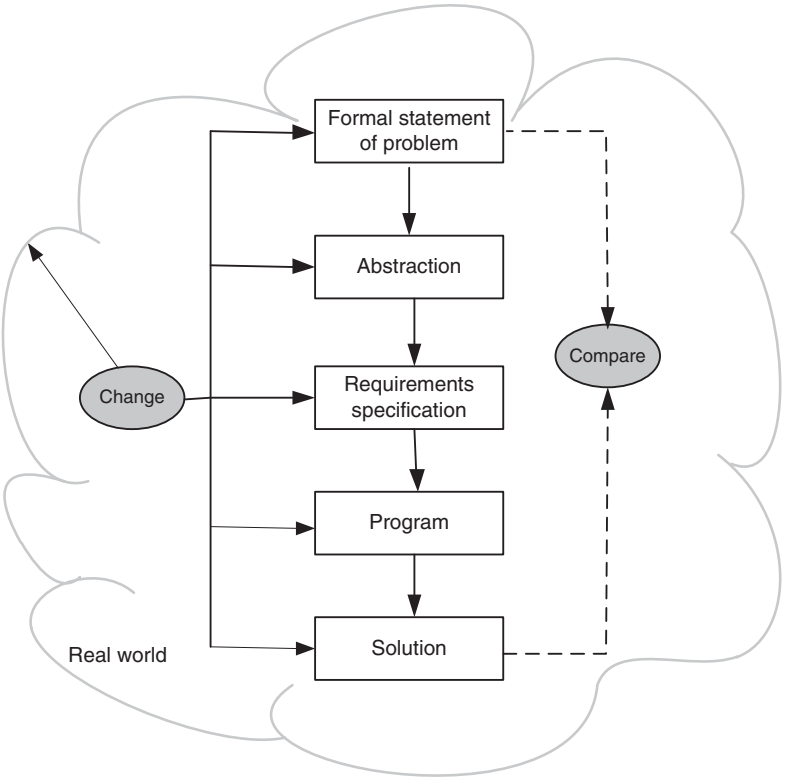

FIGURE 2.7 E-type programs

is that program execution changes its operational domain, and the evolution process is viewed as a feedback system [32]. Figure 2.8 [33] succinctly illustrates the feedback process.

2.3.2 Laws of Software Evolution

Lehman and his colleagues have postulated eight "laws" over 20 years starting from the mid-1970s to explain some key observations about the evolution of E-type software systems [34, 35]. The laws themselves have evolved from three in 1974 to eight by 1997, as listed in Table 2.5. The eight laws are the results of empirical studies of the evolution of large-scale proprietary software—also called closed source software (CSS)—in a variety of corporate settings. The laws primarily relate to perfective maintenance. These laws are largely based on the concept of feedback existing in the software environment. Their description of the phenomena are intertwined, and the laws are not to be studied separately. The numbering of the laws has no significance apart from the sequence of their development.

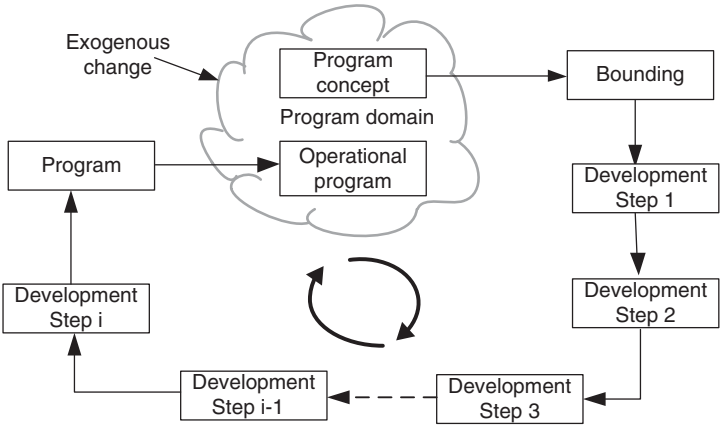

FIGURE 2.8 E-type programs with feedback. From Reference 33. © 2006 John Wiley & Sons

TABLE 2.5 Laws of Software Evolution

Names of the Laws	Brief Descriptions
I. Continuing change (1974)	E-type programs must be continually adapted, else they become progressively less satisfactory.
II. Increasing complexity (1974)	As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.
III. Self-regulation (1974)	The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.
IV. Conservation of organizational stability (1978)	The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.
V. Conservation of familiarity (1978)	The average content of successive releases is constant during the life cycle of an evolving E-type program.
VI. Continuing growth (1991)	To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.
VII. Declining quality (1996)	An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.
VIII. Feedback system (1971–1996)	The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

Source: Adapted from Lehman et al. [34]. ©1997 IEEE.

Lehman's laws were not meant to be used in a mathematical sense, as, say, Newton's laws are used in physics. Rather, those were intended to capture stable, long-term knowledge about the common features of changing software systems, in the same sense social scientists use laws to characterize general principles applying to some classes of social situations [30]. The term "laws" was used because the observed phenomena were beyond the influence of managers and developers. The laws were an attempt to study the nature of software evolutions and the evolutionary trajectory likely taken by software.

First Law *Continuing change: E-type programs must be continually adapted, else they become progressively less satisfactory.* Many assumptions are embedded in an E-type program. A subset of those assumptions may be *complete* and *valid* at the initial release of the product; that is, the program performed satisfactorily even if not all assumptions were satisfied. As users continue to use a system over time, they gain more experience, and their needs and expectations grow. As the application's environment changes in terms of the number of sophisticated users, a growing number of assumptions become *invalid*. Consequently, new requirements and new CRs will emerge. In addition, changes in the real world will occur and the application will be impacted, requiring changes to be made to the program to restore it to an acceptable model. When the updated and modified program is reintroduced into the operational domain, it continues to satisfy user needs for a while; next, more changes occur in the operation environment, additional user needs are identified, and additional CRs are made. As a result, the evolution process moves into a vicious cycle.

Second Law *Increasing complexity: As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.* As the program evolves, its complexity grows because of the imposition of changes after changes on the program. In order to incorporate new changes, more objects, modules, and sub-systems are added to the system. As a consequence, there is much increase in: (i) the effort expended to ensure an adequate and correct interface between the old and new elements; (ii) the number of errors and omissions; and (iii) the possibility of inconsistency in their assumptions. Such increases lead to a decline in the product quality and in the evolution rate, unless additional work is performed to arrest the decline. The only way to avoid this from happening is to invest in preventive maintenance, where one spends time to improve the structure of the software without adding to its functionality.

Third Law *Self-regulation: The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.* This law states that large programs have a dynamics of their own; attributes such as size, time between releases, and the number of reported faults are approximately invariant from release to release because of fundamental structural and organizational factors. In an industrial setup E-type programs are designed and coded by a team of experts working in a larger context comprising a variety of management entities, namely, finance, business, human resource, sales, marketing, support, and user process. The various groups within the large organization apply constraining information controls and reinforcing information controls influenced

by past and present performance indicators. Their actions control, check, and balance the resource usage, which is a kind of feedback-driven growth and stabilization mechanism. This establishes a self-controlled dynamic system whose process and product parameters are normally distributed as a result of a huge number of largely independent implementation and managerial decisions [36].

Fourth Law *Conservation of organizational stability: The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.* This law suggests that most large software projects work in a "stationary" state, which means that changes in resources or staffing have small effects on long-term evolution of the software. To a certain extent management certainly do control resource allocation and planning of activities. However, as suggested by the third law, program evolution is essentially independent of management decisions. In some instances, as indicated by Brooks [37], situations may arise where additional resources may reduce the effective rate of productivity output due to higher communication overhead or decrease in process quality. In reality, activities during the life cycle of a system are not exclusively decided by management but by a wide spectrum of controls and feedback inputs [38].

Fifth Law *Conservation of familiarity: The average content of successive releases is constant during the life cycle of an evolving E-type program.* As an E-type system evolves, both developers and users must try to develop mastery of its content and behavior. Thus, after every major release, established familiarity with the application and the system in general is counterbalanced by a decline in the detail knowledge and mastery of the system. This would be expected to produce a temporary slow down in the growth rate of the system as it is recognized that the system must be cleaned up to simplify the process of re-familiarization. In practice, adding new features to a program invariably introduces new program faults due to unfamiliarity with the new functionality and the new operational environment. The more changes are made in a new release, the more faults will be introduced. The law suggests that one should not include a large number of features in a new release without taking into account the need for fixing the newly introduced faults. Conservation of familiarity implies that maintenance engineers need to have the same high level of understanding of a new release even if more functionalities have been added to it.

Sixth Law *Continuing growth: To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.* It is useful to note that programs exhibit finite behaviors, which implies that they have limited properties relative to the potential of the application domain. Properties excluded by the limitedness of the programs eventually become a source of performance constraints, errors, and irritation. To eliminate all those negative attributes, it is needed to make the system grow.

It is important to distinguish this law from the first law which focuses on "Continuing Change." The first law captures the fact that an E-type software's operational domain undergoes continual changes. Those changes are partly driven by installation and operation of the system and partly by other forces; an example of other forces is human desire for improvement and perfection. These two laws—the first and

the sixth—reflect distinct phenomena and different mechanisms. When phenomena are observed, it is often difficult to determine which of the two laws underlies the observation.

Seventh Law *Declining quality: An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.* This law directly follows from the first and the sixth laws. An E-Type program must undergo changes in the forms of adaptations and extensions to remain satisfactory in a changing operational domain. Those changes are very likely to degrade the performance and will potentially inject more faults into the evolving program. In addition, the complexity (e.g., the cyclomatic measure) of the program in terms of interactions between its components increases, and the program structure deteriorates. The term for this increase in complexity over time is called *entropy*. The average rate at which software entropy increases is about 1–3 per calendar year [17]. There is significant decline in stakeholder satisfaction because of growing entropy, declining performance, increasing number of faults, and mismatch of operational domains. The aforementioned factors also cause a decline in software quality from the user's perspective [39]. The decline of software quality over time is related to the growth in entropy associated with software product aging [18] or code decay [8]. Therefore, it is important to continually undertake preventive measures to reduce the entropy by improving the software's overall architecture, high-level and low-level design, and coding.

Eighth Law *Feedback system: The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.* Several laws of software revolution refer to the role of information feedback in the life cycles of software. The eighth law is based on the observation that evolution process of the E-type software constitutes a multi-level, multi-loop, multi-agent feedback system: (i) multi-loop means that it is an iterative process; (ii) multi-level refers to the fact that it occurs in more than one aspect of the software and its documentation; and (iii) a multi-agent software system is a computational system where software agents cooperate and compete to achieve some individual or collective tasks. Feedback will determine and constrain the manner in which the software agents communicate among themselves to change their behavior [40].

Remark: 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. The following aging-related *symptoms* in software were identified by Visaggio [41]:

- *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.

Remark: The code is said to have decayed if it is very difficult to change it, as reflected by the following three key responses: (i) the cost of the change, which is effective only on the personnel cost for the developers who implement it; (ii) the calendar or clock time to make the changes; and (iii) the quality of the changed software. It is important to note that code decay is antithesis of evolution in the sense that while the evolution process is intended to make the code better, changes are generally degenerative thereby leading to code decay.

2.3.3 Empirical Studies

Empirical studies are aimed at acquiring knowledge about the effectiveness of processes, methods, techniques, and tools used in software development and maintenance. Similarly, the laws of software evolution are prime candidates for empirical studies, because we want to know to what extent they hold. In circa 1976, Belady and Lehman [22] studied 20 releases of the OS/360 operating system. The results of their study led them to postulate five laws of software evolution: continuing change, increasing complexity, self-regulation, conservation of organizational stability, and conservation of familiarity. Those laws were further developed in an article published in 1980 [24]. Yuen [36, 42] further studied their five laws of evolution. He re-examined three different systems from Belady and Lehman [22] and several other systems and examined a variety of dependent variables. The number and percentage of modules handled are examples of dependent variables. After re-examining the data from previous studies, he observed that the characteristics observed for OS/360 did not necessarily hold for other systems. Specifically, the first two laws were supported, while the remaining three laws were not. Yuen, a collaborator of Lehman, notes that these three laws are more based upon those of human organizations involved in the maintenance process rather than the properties of the software itself.

Later, in a project entitled FEAST (Feedback, Evolution, And Software Technology), Lehman and his colleagues studied evolution of releases from four CSS systems: (i) two operating systems (OS/360 of IBM and VME OS of ICL); (ii) one financial system (Logica's FW banking transaction system); and (iii) a real-time telecommunication system (Lucent Technologies). Their results are summarized as a set of growth curves, as described by Lehman, Perry, and Ramil [43]. The studies suggest that during the maintenance process a system tracks a growth curve that can be approximated either as linear or inverse square [44]. The inverse square model represents the growth phenomena as an inverse square of the continuing effort. Those trends increase the confidence of validity of the following six laws: Continuing change (I), Increasing complexity (II), Self-regulation (III), Conservation of familiarity (V),

Continuing growth (VI), and Feedback system (VII). Confidence in the seventh law "Declining quality" is based on the theoretical analysis, whereas the fourth law "Conservation of organizational stability" is neither supported nor falsified based on the metric presented. In 1982, there was an independent study by Lawrence [45], who took a statistical approach to observe some evidence supporting laws I and II, while laws III–V were not supported by the data.

Inverse square law. According to the fourth law, the incremental effort, denoted by E, expended on each release stays the same during the evolution of the system. It is assumed that the incremental effort expended on each release is almost the same during the system evolution. Let Δ_i denote the incremental change of the system size and assume that it is solely due to the effort E expended on the ith release. To relate E and Δ_i, we need to consider a conceptual factor s_i, which behaves in this context like mass in dynamic physical systems. A larger s_i implies a greater resistance to change, and a smaller Δ_i will be obtained from expending effort E. Thus, $\Delta_i = E/s_i$ is the first, simple relation that appears in the view. Assume that the size of the system is expressed in terms of *number of modules,* and complexity is expressed in terms of the *number of intermodule interactions.* By considering the number of intermodule interactions to be proportional to the square of the number of modules, one can obtain a relationship as follows: $\Delta_i = E/s_i^2$, which is called the *inverse square law*. We assume that the law is valid and obtain the following expressions:

$$
\begin{aligned}
s_1 &= s_1 \\
s_2 &= s_1 + E/s_1^2 \\
s_3 &= s_2 + E/s_2^2 \\
&\dots
\end{aligned}
$$

By means of substitution we get:

$$
\begin{aligned}
s_1 &= s_1 \\
s_2 &= s_1 + E/s_1^2 \\
s_3 &= s_2 + E/s_2^2 = s_1 + E\left(1/s_1^2 + 1/s_2^2\right) \\
&\dots
\end{aligned}
$$

Similarly, we get:

$$
\begin{aligned}
E_1 &= \frac{s_2 - s_1}{1/s_1^2} \\
E_2 &= \frac{s_3 - s_1}{1/s_1^2 + 1/s_2^2} \\
&\dots
\end{aligned}
$$

where the right-hand sides of the equations contain data about releases and nothing else. Thus, one can compute the mean of n values of E as $\overline{E} = (\sum_{i=1}^{n} E_i)/n$. $\overline{E}$ can be interpreted as the mean effort required per release, which can be considered to be a good approximation of the "constant" effort throughout the system evolution. Employing $\overline{E}$, system evolution is described by using the inverse square law as follows:

$$
\begin{aligned}
s_1 &= s_1 \\
s_2 &= s_1 + \overline{E}/s_1^2 \\
s_3 &= s_2 + \overline{E}/s_2^2 \\
&\cdots \\
s_n &= s_{n-1} + \overline{E}/s_{n-1}^2
\end{aligned}
$$

The aforementioned relationship is consistent with the view that increasing complexity, captured in the second law, restrains growth. The inverse square law has an interesting property. By showing that the model closely matches the evolution pattern of a system, one may accurately predict the size of the subsequent releases after the data about the first few releases are available.

2.3.4 Practical Implications of the Laws

Based on the eight laws, Lehman suggested more than 50 rules for management, control, and planning [35] of software evolution. Those 50+ rules are put into three broad categories: assumptions management, evolution management, and release management [46].

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. In this section, the discussion has mainly referred to the evolution of software as reflected in a series of releases or upgrades. Recommendation relating to evolution and maintenance process includes the following list of items:

- 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 nonfunctional 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 m_i 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 ith desired release (say, m_i) 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 $m + 2s$. Finally, the release is *unsafe* if the content of the desired release is close to or greater than $m + 2s$. 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.

The model discussed in this section concentrated on systems developed under industrial software process paradigm, namely, Closed Source Software (CSS) and extension of the waterfall model. The discussion has mainly referred to the evolution of software as reflected in a series of releases or upgrades. But general validity of the laws of Lehman in the context of newer paradigms, such as open source, agile programming, and COTS-based development, cannot be taken for granted. Hence, we discuss the evolution of free and open source software (FOSS) systems next.

2.3.5 Evolution of FOSS Systems

FOSS is a class of software that is both free software and open source. It is liberally licensed to grant users the right to use, copy, study, change, and improve its design through availability of its source code. The FOSS movement is attributed to Richard M. Stallman, who started the GNU project in circa 1984, and a supporting organization, the Free Software Foundation [47]. It is often emphasized that free software is a "matter of liberty not price" [48]. FOSS—also referred to as FLOSS (Free/Libre/Open Sources Software)—systems have attracted much academic and commercial interests due to the accessibility to large amounts of code and other free artifacts. Gradually, more and more software systems were developed by Open Source Community (OSC). Compared with CSS development methods, FOSS have lots of new characteristics. Eric Raymond concisely documented the FOSS approach in an article entitled "The Cathedral and the Bazaar" [49]. In this section, we briefly describe the differences between the evolutions of FOSS-based software and CSS-based software in terms of: (i) team structure; (ii) process; (iii) releases; and (iv) global factors [50–52].

Team structure. In traditional CSS development, organizations often have dedicated groups to carry out evolution tasks. These groups are staffed with specialist maintenance personnel. In contrast, FOSS development is very different. Even though several FOSS communities have core teams to manage evolution activities on a daily basis, most works are done voluntarily.

An onion model of FOSS development has been illustrated in Figure 2.9 [53, 54]. According to this model, a core sustainable community consists of a small group of key members, additional contributing developers, and a large number of active users who report defects. The outer layer represents those users who are not actively involved in the development process. The onion model has three primary

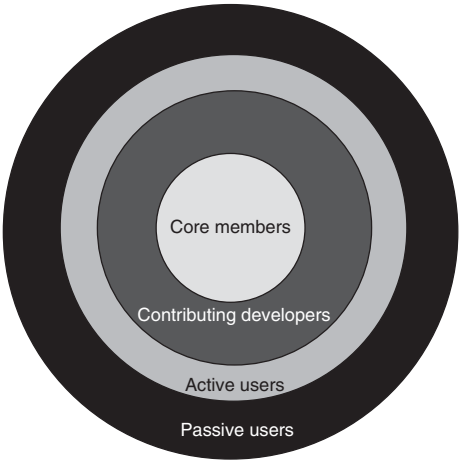

FIGURE 2.9 Onion model of FOSS development structure

characteristics: (i) a small core team; (ii) contributing developers add and maintain features; and (iii) active users take ownership of system testing and defect reporting. In the FOSS evolution model, numerous nomadic volunteers work together as a community. Therefore, one should consider the change of people in the evaluation of evolution of FOSS-based software.

Process. The FOSS development process is lighter than CSS development process followed in companies, where requirement documents and design specifications are indispensable. There are strict rules about coding style, documentation, and defect fixes. On the other hand, in FOSS development, requirement specification and detailed design documentations take a back seat, at least from the user's perspectives.

Though there are standards for coding and documenting, these are less relaxedly adhered compared to traditional CSS development. In FOSS development, source code comprises the main artifact for disseminating knowledge among the developers. Therefore, FOSS activities are largely confined to coding and testing. To overcome the ensuing difficulties due to not following a document-driven development process, an array of supplementary information is provided: release notes, defect databases, configuration management facilities, and email lists. For some projects, some developers act as the "gate keepers" for any revision to the code. Each project community makes their own rules to regulate the submission of bug fixes and new functionalities. Systematic testing is not always present compared to CSS-based projects. However, in FOSS projects, due to the large number of developers and beta-testers, almost all the issues can be quickly characterized and the fix is obvious to someone. As observed by Eric S. Raymond [49], "given enough eyeballs, all bugs are shallow." This is known as *Linus's Law*: the more widely available the source code is for public testing, scrutiny, and experimentation, the more rapidly all forms of defects are discovered.

Remark: Linus's law was attributed to a Finnish software engineer, Linus Benedict Torvals by Eric S. Raymond. He was the lead of the Linux kernel project and

later became the chief architect of the Linux operating system and the project coordinator.

Releases. A key attribute of FOSS is that code is shared with almost no constraints. Compared to CSS development, FOSS development generally do not have schedule for regular releases. However, larger FOSS projects do have stated goals and releases are generally scheduled in terms of functionalities to be delivered. In general, there are two related streams of source code: (i) a stable stream of code for distribution and (ii) a development stream. The latter one is currently being modified and improved. At some instant, the development stream becomes stable and is released. The development stream is frozen for a few days prior to a milestone release to identify critical problems. When it is determined that critical problems have been resolved and the code is indeed stable, then the code is released [55]. However, it is a common behavior in many FOSS projects to follow the rule: "release early, release often." The above rule means that the code is available to public well before it is stable.

Global factors. In the FOSS development paradigm, developers working on even a very small project might be living in many countries around the world, due to the pervasive use of the world-wide web (WWW). With globalization on the rise, the collaborators hail from many countries with a variety of cultural backgrounds. FOSS development becomes very challenging because of the need to coordinate the geographically distributed developers. Though many companies have their development teams spread out in many countries, most of the traditional companies develop systems with their local teams and, occasionally, the system is tested at an overseas location.

Empirical Studies of FOSS Evolution In circa 1988, Pirzada [56] analyzed the dissimilarities between the systems studied by Lehman and Belady [32] and the evolution of the Unix operating system. It was argued that the differences between commercial development and development for academic purposes could lead to differences in their evolutionary trajectories. In circa 2000, empirical study of FOSS evolution was conducted by Godfrey and Tu [57]. They provided the trend of growth between 1994 and 1999 for the Linux operating system (OS), which is a popular FOSS system, and showed its rate of growth to be superlinear. Specifically, they found that the size of the Linux followed a quadratic growth trend, and at that time the OS was about 2+ million lines of code (LOCs). In circa 2002, Schach et al. [58] studied the evolution of 365 versions of Linux and showed that module coupling, that is interconnection of modules, has been growing exponentially. They argued that unless efforts to alleviate this situation is undertaken, the Linux OS would become unmaintainable. Their argument was fully consistent with Lehman's sixth and seventh laws. However, it appears to be at odds with the third and the fifth laws, which are self-regulation and conservation of familiarity, respectively. Robles et al. [59], while replicating Godfrey and Tu's study, concluded that Lehman's fourth law (conservation of organizational stability) does not fit well with large-scale FOSS systems such as Linux. This behavior of exponential growth may be considered as an anomaly as pointed out by Lehman et al. [60]. The evolutionary behavior of other

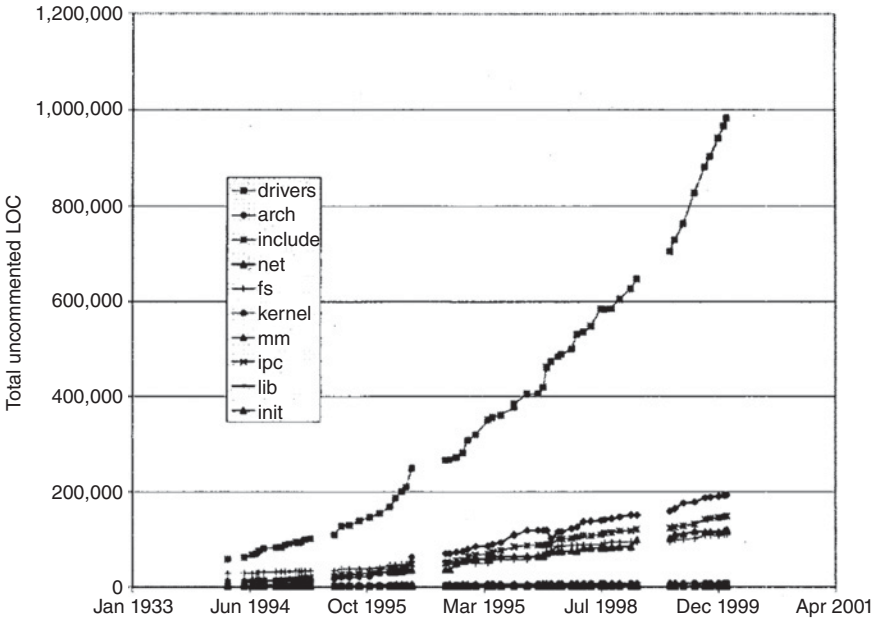

FIGURE 2.10 Growth of the major subsystems (development releases only) of the Linux OS. From Reference 57. © 2000 IEEE

FOSS systems such as *Gcc*, *Linux kernel*, *Apache*, *Brocade library*, and *Zlib* appears to follow Lehman's laws for software evolution [61, 62].

Godfrey and Tu observed that the growth rate was more for the device-driver subsystem of Linux as can be seen from Figure 2.10. As a matter of fact, the device-drivers appear to be mutually independent. Therefore, adding a new driver does not raise the subsystem's complexity. Another characteristic of Linux is that the system gives a false impression that it is larger than it really is. The "larger-than-real-size" impression is observed due to the fact that certain features, say, supporting different CPU types, are implemented with code replication (software clones).

Moreover, participation of an unrestricted pool of (novice) developers may explain this, that is, software clones. In order to have a better understanding of how a system is evolving, it is necessary to analyze and understand each subsystem within and across software releases irrespective of FOSS or CSS systems. This observation has been reported by Gall et al. based on the study of a CSS: a large telecommunication switching system [63].

2.4 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 preexisting building blocks, components, and plug them together as required to built 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 [64]. 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. A glue component is basically designed to combine the services from many components to provide a higher level of service. Component tailoring means enhancing the functionality of a component, and it is done by adding new elements to a component. Note that the source code is not changed by this activity. "Scripting" is an example of tailoring, because a program can be enhanced by having some event trigger a script.

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.

2.4.1 Why Maintenance of CBS Is Difficult?

The cost of maintaining COTS-based software systems represent a significant fraction of the total cost of developing software products. Studies show that CBSs incur more maintenance cost than in-house built software [65]. As a first step, reduction of the cost of COTS-based software requires understanding of what makes CBSs maintenance

difficult. In the following, we provide a list of those difficulties [66]. Next, we explain those difficulties one by one.

- 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. The term host organization refers to an individual, group or organization that applies components as a part of some software system, called host system. To find a solution, an integrator is faced with the following options: (i) attempt to implement the frozen functionalities; (ii) acquire a new but similar component from a different vendor; and (iii) acquire the source code from the present vendor to maintain it. The first option is the most difficult choice, unless the host organization has the necessary domain knowledge. The second one is likely to be opted if there are competing alternative vendors. Otherwise, the third one is the only option available. In order to exercise the third option, the integrator should develop a good understanding of the domain to maintain the component source code.

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. Assuming that the solutions available to handle the frozen functionality problem are not available, then, as a fourth solution, one may build wrappers around a component to refrain it from exhibiting the incompatibility creating behavior. If wrappers alone are insufficient to eliminate incompatibilities, one can rewrite the "glue" connecting the newly possessed components with the existing ones. Finally, if wrappers and the modified "glue" do not solve the upgrade problem, then the integrator may consider downgrading the functionalities of the system. Not upgrading to the next version can produce the following consequences:

- There may not be continued vendor support for prior versions.
- The host organization may be unable to purchase more copies of the version in operation. Additional copies of a product may be needed when the system is being incrementally deployed.

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. Detection of malicious behavior is difficult enough even with full access to program code. Detection of suspicious actions in a running component will require capturing the requests emanating from the component and verifying their contexts. For COTS components, this can be done at the wrapper level. Note that this approach to detecting malicious behavior is too expensive, because too many calls and context checks are involved. In a running component, Trojan horses go largely undetected. It is an issue that programmers must be aware of while substituting an old component with a new one. Component substitution requires specialized procedures for testing and certifying COTS components. Each time some COTS products are changed, the components and the host system may have to be recertified.

Unreliable COTS components. The scenario of incompatible upgrades has been discussed before in this section. Now we consider unreliable COTS components. Though incompatibility and unreliability are related, there is a distinction between the two. Today, no uniform standard exists to test software components to certify their reliability [67]. By paying software certification laboratories (SCL) to grant software certificates, independent vendors partially shift their responsibility to the SCL. However, there are several ramifications of using services from SCLs: cost, liability issues, developer resources needed to access SCLs, and applicability to safety-critical systems [68]. It may be argued that products with better reliability can be produced with good processes, which can be graded with Capability Maturity Model (CMM), Test Process Improvement (TPI) model, and Test Maturity Model (TMM) [69]. However, process quality does not guarantee product quality, and a vendor may not reveal the maturity level of their process. Though software reliability models exist for decades [70], generic assumptions are made in those models about execution environments, rate of defect, severity of defects, and sizes of faults. Consequently, it is difficult to apply those models. The assumptions may not reflect the individual peculiarities of different environments. Therefore, 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 analyzing 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 behavior 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 preprocessing 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. Querying the vendor could elicit some bits and pieces of information, but it is prudent to thoroughly test a component in its real environment. One can combine vendor supplied information with results of in-house testing to design better wrappers. But, wrappers can be complex, incomplete, and unreliable. Wrappers are discussed in Section 5.2 in great details.

2.4.2 Maintenance Activities for CBSs

It is necessary to identify the activities of the maintenance and management personnel to effectively manage COTS-based systems. Strategies can be formulated to facilitate those activities. Vigder and Kark [71] have surveyed several organizations maintaining systems with a significant portion of COTS elements. In their study, they identified the following cost-drivers:

- Component reconfiguration
- Testing and debugging
- Monitoring of systems
- Enhancing functionality for users
- Configuration management.

Component reconfiguration. Component reconfiguration means adding, removing, and replacing components of a system. The following actions lead to a system being reconfigured: (i) add new components to increase the capability of the system; (ii) delete components as requirements change; (iii) replace a component with a newer version; (iv) replace an old component with a better one; and (v) replace an in-house built component with COTS components. Often component vendors release software updates many times in a year. Therefore, integration of the enhanced components into the host system becomes an expensive task. The host organization continuously evaluates new component releases from the vendors. It establishes criteria by considering capabilities, risk, and cost to make a decision on system upgradation. If a decision is made to upgrade the system, then: (i) the components are analyzed within the context of the current host system and (ii) a new cycle for system integration and testing is planned. To determine the differences between the old version and the new version, system interfaces and behavior are tested. It is likely that the assumptions of the enhanced COTS system are not consistent with those of the other COTS elements. Hence, rigorous regression testing must be performed. In summary, performing component reconfiguration is a time-consuming process that requires an organization to move through a full release cycle: evaluate the product, obtain a design, perform integration, and execute system regression tests [72].

Testing and debugging. Every organization follows its own methodologies, strategies, processes, and techniques for performing testing and debugging on in-house developed code. However, testing and debugging of CBSs pose new challenges due to the absence of visibility into the internal details of third-party developed components. For example, 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.

System monitoring. While a system is in operation, maintainers need to closely monitor the system to be able to better understand the system performance. Continuous monitoring enables the maintenance personnel to enhance the performance of the system, measure usage of resources, keep track of the anomalies in the system behavior, and perform root cause analysis of system failures. Monitoring for the purpose of maintenance is a difficult task because of the low visibility of the internal operations of COTS software.

Enhancing the functionalities for users. COTS products are designed and implemented in a broad sense so that those can be adapted in various applications. Integration personnel need to customize and tailor COTS functionality to satisfy their user community. Therefore, successful host systems exhibit the properties of efficient modifiability and tailorability to incorporate evolving and new user requirements. Tailoring involves a continual process of configuring and customizing of products, combining services of multiple products, and adding new components to products. In the absence of access to source code, tailoring is done by means of two techniques:

- write additional glue code to hold the system together and provide enhanced functionality; and
- use vendor supplied tailoring techniques to customize the products, because integration personnel have no access to program code.

Configuration management. Configuration management of CBS systems is done at two levels: (i) source-code level to manage the in-house software, namely, wrapper, glue, and tailoring developed by the personnel performing system integration and (ii) component level to manage COTS, procured from third-party vendors [73]. The following five activities are specifically done for CBS products:

- 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 Unix" is an instance of "previous use."
  - Obtain the contact information of some of those using the component.
- Perform configuration management on the individually tailored COTS elements.
- Track the configuration history of a product at all its deployment sites.
- Find the compatible versions of the various COTS elements.
- Manage support and licenses for each COTS element.

Configuration management is a key activity over the life cycle of a large system. For COTS components, configuration management needs to be performed for each COTS software product and each platform on which the product is installed. The lists of those COTS software products and platforms are needed while (i) distributing software upgrades or fixes to multiple sites or (ii) restoring configurations that have been broken.

2.4.3 Design Properties of Component-Based Systems

The architecture of a CBS has significant impact on its maintainability. Component maintainability properties, such as minimal component coupling and visibility, cannot be enhanced after building a CBS. Rather, one must consider these properties at the time of the initial development of the CBS. The main areas influencing CBS maintainability are the choice of the components and the architecture and design used to perform system integration on the components.

Component Selection Though system integrators do not design and implement individual components, they do have a say while components are selected to be integrated into the host system. The CBS integrator must consider the CBS evolution factor when designing criteria for component selection. A number of attributes of components effect the evolution and maintenance of CBSs. These attributes are discussed in what follows.

Openness of components. A component is considered to be open if it is designed to be visible, extensible, adaptable, and easily integrated into a variety of different host systems. In general, the more the openness of a component, the easier it is for integrators and maintainers to monitor, manage, extend, replace, test, and integrate. The factors that make a component open are adherence to standards, availability of source code, and ability to inter-operate with products from other vendors. Source code can be made available through original equipment manufacturer (OEM) partnership.

Tailorability of components. Tailoring the functionality of the components to meet the evolving user requirements is a kind of maintenance effort for CBS systems. One criterion for component selection can be whether or not the component can be tailored to satisfy the end users' requirements. Though components are seen as black-boxes, vendors can apply many techniques to make components tailorable. Examples of tailoring techniques are: scripting interfaces and extendible frameworks through the use of plug-ins and inheritance.

Available support community. Host system builders need much support from external organizations to build and maintain commercial products. The external support comes from the user community and the vendors. Noting that external support is key to system maintenance, the host system builders need to evaluate the support available during the component evaluation process.

Design Properties of Maintainable CBS By analyzing and partly resolving the issues of maintainability in the design phase, one can build a system that facilitates the maintenance of CBSs. This requires the development of a set of criteria to evaluate maintainability. The following design attributes of a maintainable CBS have been identified by Vigder and Kark [71]:

- Encapsulated component collaborations
- Controlled component interfaces
- Controlled component dependencies
- Minimal component coupling
- Consistent failure handling
- High level of visibility
- Minimal build and deployment effort.

Encapsulated component collaborations. Collaborations are time-sequenced coordinated actions. Collaborations among components can involve many data and behavioral dependencies. A key design objective is to make collaborations explicit and encapsulate each collaboration within a single object, often called a *mediator*. A mediator can be implemented as part of the glue code, and it should be designed to (i) translate and transform data formats to enable data transfer and (ii) manage event sequencing for the components. By encapsulating collaborations the CBS will be more maintainable by supporting the following activities [74]:

- *Product reconfiguration.* It is much easier to understand and manage component dependencies by encapsulating component interactions within a separate object.
- *Troubleshooting.* Numerous problems in CBS are related to sequences of interactions among components. Problem isolation becomes easier if most of the interaction sequences happen within a single mediator.

- *Modifying and adding services.* By combining services from different components, mediators implement many business processes. Updating business processes become easier by confining services to mediators.
- *System monitoring.* Mediators can include instrumentation code for monitoring system behavior.

Controlled component interfaces. There are two main reasons to use integrator-controlled interfaces on COTS components: (i) facilitate component reconfiguration; and (ii) add visibility. As new components and component versions are combined with a system, an integrator-controlled interface can reduce the impact of frequent reconfigurations by means of isolation. In addition, management, instrumentation, and monitoring functionality can be included in an integrator. If integration personnel directly use interfaces supplied by vendors, they might face difficulties in: (i) reducing the consequences of reconfigurations and (ii) determining the dependencies between components. Integrators can use a number of approaches to turn interfaces into first-class objects and manipulate them. A first-class object is one that can be dynamically created, destroyed, or passed as an argument. One approach to creating a first-class object is designing a wrapper around all the components; a wrapper can be designed by using an adapter design pattern. The adapter design pattern translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. As the underlying component is modified, the ripple effect on the other components can be minimized. A second approach to creating a first-class object is to use standardized interfaces for COTS products.

Controlled component dependencies. The integrator identifies mutually dependent components and realizes strategies for controlling and managing those dependencies. Complex dependencies among components produce a fragile system. It becomes difficult to upgrade, add, and delete elements in a fragile system. To make maintenance easier for CBS systems, designers must reduce dependencies between components. Some dependencies are explicit and some are implicit. Flow of data through a visible interface is an example of direct dependency. Conflicting assumptions made by different software components are examples of implicit dependency. Dependencies can also result from resource contentions; an example of resource contention is when many components try to use the same TCP port number at the same time. Dependencies between COTS products include the following three broad kinds of dependencies:

- *Syntactic dependencies.* These occur when components make assumptions about the interface signature of the component.
- *Behavioral dependencies.* These occur when components are involved in two-way interactions or multi-way collaborations.
- *Resources dependencies.* These occur when multiple components compete for the same resources.

The designer must record, update, and verify all cross-component dependencies to perform risk analyses before a system upgrade involving those components. In addition, integrators can provide mechanisms for managing those dependencies: identification of deployment-time versions and verification of satisfaction of dependencies.

Minimal component coupling. It becomes more difficult to substitute a component if coupling of the component with other components is very high. Minimal component coupling is realized by: (i) constraining and controlling component access and (ii) isolating the resources used by separate components. A wrapper around a component can suppress undesired functionalities that introduce additional dependencies.

Consistent failure handling. In their tasks of testing, debugging, and monitoring system behavior, maintainers are assisted by failure handling. Therefore, integrators need consistent, complete, and effective means to detect and handle failures. A component can behave in an unpredictable manner if it is provided with faulty inputs. Therefore, it is important to identify and isolate faults sooner before they propagate as errors through other components. A consistent failure handling mechanism enables maintainers to detect errors when they occur, identify the root cause of the failure, and minimize the impact of the failure. For example, a consistent failure handling mechanism for any routines includes a *status* output argument, which is used to return error status codes. Status codes may be passed to another routine (viz. errormsgtext()) to extract an error message text from a message catalogue. Errors can be detected by the wrappers and handled in the glue code holding the system together.

High level of visibility. Visibility means that maintenance engineers are able to monitor the system, including the behavior of wrappers and glue components. Visibility can be added to the system by the integrators in several ways. Instrumentation and monitoring capabilities should be integrated into wrappers and glue code to support monitoring of interactions as part of the overall system design. Monitoring tools can support additional capabilities to gain visibility into a running system, namely, monitoring the input and output behavior of communication protocols with sniffer code.

Minimal build and deployment effort. CBSs are often built with many components that require frequent tailoring and reconfigurations. Therefore, the build process to install the software at all deployment sites is complex. On the other hand, it should be easily achievable to replace products or add new functionalities without going through an expensive build process. COTS elements may involve intricacies in their deployment processes, thereby contradicting the assumptions made by other products. As a result, the build itself can become an expensive and complex part of system maintenance [75, 76]. A build process becomes complicated if too many modules are new and have complex interfaces. A tool for version control is highly recommended for automating the build process.

2.5 SUMMARY

This chapter began with definitions of software maintenance from the perspectives of researchers, practitioners, and standardization groups. We differentiated development from maintenance: developing new software is a requirement-driven activity, whereas

maintaining an existing system is event driven. For example, when a request for change is received, the maintenance organization may modify the system. Therefore, the inputs that drive maintenance changes are random events, originating, for example, from an user in the form of a CR. Software maintenance consists of two primary activities: correcting errors and enhancing functionality of the software. Hence, it can be seen as continued development.

We identified and explained the following maintenance activities:

- intention-based classification of software maintenance activities;
- activity-based classification of software maintenance activities; and
- evidence-based classification of software maintenance activities.

To explain intention-based classification of maintenance tasks, we introduced Swanson's approach [5] which defined three kinds of maintenance activities: corrective, adaptive, and corrective. On the other hand, the maintenance classification of Kitchenham et al. [4] consists of two broad kinds of activities for corrections and enhancements. The category for enhancement is subdivided into three types as follows: (i) modifications that change some of the current requirements; (ii) modifications that add new requirements to the system; and (iii) modifications that alter the implementation but not the requirements. The evidence-based classification of Chapin [16] consists of 12 types of software maintenance tasks: training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive.

Next we explained various concepts that influence software maintenance processes. Those concepts were studied under four categories: (i) maintained product; (ii) maintenance types; (iii) organization process; and (iv) peopleware. Then, we described the characteristics of those concepts and their impact on maintenance activities. This discussion clarifies the difference between maintenance methods/tools/skills from those used for software development.

Next, we studied various ways in which researchers define software evolution, and differentiated it from software maintenance. Keith H. Bennett and Jie Xu [27] use the term maintenance to refer to all post-delivery support activities, and evolution to perfective changes, that is, those driven by changes in requirements. In addition, the authors further state that evolution addresses both functional and nonfunctional requirements. On the other hand, Ned Chapin defines software evolution as the applications of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version [15].

We described Lehman's classification of properties of CSS of S-type (Specified), P-type (Problem), and E-type (Evolving). The S-type programs implement solutions to the problems that can be completely and unambiguously specified, for which, in theory at least, a program implementation can be proven correct with respect to the specification. The definition of S-type requires that the program be correct in the full mathematical sense related to the specification. A P-type program is based on a

practical abstraction of the problem, rather than a completely defined specification. Even though the exact solution may exist, the solution produced by a P-type program is tampered by the environment in which it must be produced. The solution produced by a P-type program is acceptable if the results make sense to the stakeholder(s) in the world in which the problem is embedded. Finally, the distinctive attributes of E-type systems are as follows:

- The complex and large problems addressed by E-type systems cannot be completely and formally specified.
- The system has an incomplete model of the execution environment that embeds the program.
- The system makes a large number of simplifications and assumptions about the real world.
- Program execution modifies the operation domain.
- The development and evolution processes for E-type software are feedback driven.

Next, we discussed the following eight laws of software evolution for E-type CSS systems, including empirical studies and their practical implications.

I. Continuing change. E-type programs must be continually adapted, else they become progressively less satisfactory.

II. Increasing complexity. As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.

III. Self-regulation. The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.

IV. Conservation of organizational stability. The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.

V. Conservation of familiarity. The average content of successive releases is constant during the life cycle of an evolving E-type program.

VI. Continuing growth. To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.

VII. Declining quality. An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.

VIII. Feedback system. The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

Next, we described the origin of FOSS movement and the differences between CSS and FOSS systems with respect to: team structure, process, releases, and global factors. In addition, we discussed the empirical research results about the Linux FOSS system to study the laws of evolution, originally proposed for CSS systems. We concluded this chapter with a discussion on maintenance of COTS.

LITERATURE REVIEW

The book by Dennis D. Smith ("Designing Maintainable Software," Springer, New York, 1999) is an excellent starting point for understanding the many issues related to maintenance. With theoretical reasoning and observation, the book explains how maintainers undergo problem solving. He provides helpful tips for maintainers regarding cognitive structures, naming conventions, and the use of truncation. Though the book does not cover evolution, it clearly explains software evolvability. Software maintenance is usually considered in terms of corrections, improvements, and enhancements. The article by Dewayne E. Perry ("Dimensions of Software Evolution" by D. E. Perry, International Conference on Software Maintenance, Victoria, BC, IEEE Computer Society Press, Los Alamitos, CA, September, 1994, p. 296–303) looks at the other three dimensions: domain, experience, and process to gain insights into the sources of software evolution. These three dimensions are interrelated in various ways and interact with each other in a number of ways. One will be able to understand and manage effectively the evolution of software systems only when there is a deep understanding of these dimensions. This idea was further extended by Ciraci et al. ("A Taxonomy for a Constructive Approach to Software Evolution," by S. Ciraci, P. Broek, and M. Aksit, Journal of Software, Vol. 2, No. 2, August, 2007, p. 84–96) to 24 feasible contexts for the software evolution. The taxonomy is based on the fact that a change in one of these sources (e.g., domain, process, or experience) has occurred or is expected to occur.

The typology of Swanson has been influential among researchers and practitioners [7]. However very few researchers and practitioners have followed Swanson's typology. Rather, others have given new meanings to those terms. For example, the standard proposed by IEEE [77] defines the three terms "corrective," "adaptive," and "perfective," which are not completely compatible with Swanson's. The article by Chapin et al. [15] clearly identifies and compares the differences in a tabular form along with the definitions of 12 maintenance types: training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive. The authors excluded the "perfective" type from their classification and coined a new term called "groomative." The reader is urged to study the article on preventive maintenance by Kajko-Mattsson [13]. The author did a comprehensive literature study on preventive maintenance both within software and hardware engineering. In addition, Kajko-Mattsson et al. [78] have discussed a comprehensive taxonomy of activities performed for the corrective type.

In the article by Jim Buckley et al. [79], the authors took a complementary view toward a taxonomy of maintenance by focusing more on the *how, when, what,* and *where* aspects of software changes. The article proposed four logical themes and 15 dimensions. The four logical themes are: (i) temporal properties (*When* is the change made?); (ii) object of change (*Where* is a change made?); (iii) system properties (*What* is being changed?); and (iv) change report (*How* is the change accomplished?). The 15 dimensions are: time to change, change history, change frequency, anticipation, artifacts, granularity, impact, change propagation, availability, activeness, openness, safety, degree of freedom, degree of formality, and change type.

The revised form of the SPE taxonomy, called SPE+, was proposed by Stephen Cook et al. [30] to address some ambiguities and weaknesses in the original taxonomy. For example, Lehman did not choose to characterize the P-type software.

Kemerer and Slaughter [80] showed that problems in software maintenance can be traced to an absence of understanding of: (i) the maintenance process and (ii) the cause–effect relationships between software maintenance practices and outcomes. Their study focused on the effort expended, modification performed, and cost incurred to evolve the software.

Many researchers are further studying new topics related to laws of software evolution. Readers interested in a more detailed discussion of the topic may read the following books:

N. H. Madhavji, J. F. Ramil, and D. E. Perry, Ed. *Software Evolution and Feedback – Theory and Practice*, John Wiley, West Sussex, England, 2006.

T. Mens and S. Demeyer, Ed. *Software Evolution*, Springer-Verlag, Berlin Heidelberg, 2008.

The first book focuses on the *what and why* aspects of software evolution, that is, on the *nature* of software evolution phenomenon with emphasis on nontechnical aspects, such as complexity theory, social interactions, and human psychology. This book provides a depth of material in the field of software evolution and feedback. Specifically, it describes the phenomenological and technological underpinnings of software evolution, and it explains the impact of feedback on development and maintenance of software. Part I (Chapters 1–16) is "evolution" centered, whereas part II (Chapters 17–27) is "feedback" centered, though both the topics are often discussed in the same chapter. Within these partitions, the chapters are organized from one more conceptual to more concrete contents.

The second book entitled "Software Evolution" focuses on the *how* aspects of software evolution: methods, activities, tools, and technology that give the means to manage software evolution. The book has been structured into three parts. The first part focuses on: (i) analysis of release histories and version repositories and (ii) improvement of evolution by fixing defects and eliminating redundancies in software. The second part explains how one can reengineer a legacy system into a modern system that is easier to maintain. The third part discusses the relation between evolution and other main subjects in software development.

Those interested in knowing more details about the comparative empirical study of FOSS and CSS may refer to the article by Paulson et al. [61]. The authors studied and compared the results of the evolution of three CSS and FOSS projects. The three well-known FOSS projects are the Linux kernel, the GCC compiler, and the Apache HTTP web server. The authors chose to consider only the kernel part of Linux because, apparently, the three CSS systems were more comparable with the kernel than the whole Linux system. The three CSS projects are from the embedded real-time system domain described as "software protocol stacks in wireless telecommunication device." The five hypotheses studied in this article were: (i) FOSS grows more quickly than proprietary, that is CSS, systems; (ii) FOSS systems foster more creativity; (iii) FOSS systems are less complex than CSS systems; (iv) fewer bugs are there in

FOSS systems and those can be more rapidly located and fixed; and (v) FOSS systems are better modularized. Out of those five hypotheses, only (ii) and (iv) were supported with measurements. The following measurements were used to test the hypotheses:

1. For hypothesis (i), count the number of lines of code added over time. This measurement captures the size (or growth) metric.
2. For hypothesis (ii), count the number of functions added over time. This measurement reflects the creativity shown over time.
3. For hypothesis (iii), measure the average complexity of all the functions and the average complexity of all the newly added functions.
4. For hypothesis (iv), count the number of functions changed to fix bugs and represent the number of functions changed to fix bugs as a percentage of the total number of functions.
5. For hypothesis (v), compute the correlation between the number of functions added and the number of functions changed.

REFERENCES

[1] R. G. Canning. 1972. The maintenance 'iceberg'. *EDP Analyzer*, 10(10), 1–14.

[2] T. M. Pigoski. 2001. *Chapter 6, Software Maintenance, SWEBOK: A Project of the Software Engineering Coordinating Committee (Trial Version 1.00)*. IEEE Computer Society Press, Los Alamitos, CA.

[3] ISO/IEC 14764:2006 and IEEE Std 14764-2006. 2006. *Software Engineering – Software Life Cycle Processes – Maintenance*. Geneva, Switzerland.

[4] B. A. Kitchenham, G. H. Travassos, A. N. Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. 1999. Towards an ontologyy of software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 11, 365–389.

[5] E. B. Swanson. 1976. *The Dimensions of Maintenance*. Proceedings of the 2nd International Conference on Software Engineering (ICSE), October 1976, San Francisco, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 492–497.

[6] B. P. Lientz and E. B. Swanson. 1980. *Software Maintenance Management*. Addison-Wesley, Reading, MA.

[7] E. B. Swanson and N. Chapin. 1995. Interview with E. Burton Swanson. *Journal of Software Maintenance and Evolution: Research and Practice*, 7(5), 303–315.

[8] S. G. Eick, T. L. Graves, A. f. Karr, J. S. Marron, and A. Mockus. 2001. Does code decay? Assessing evidence from change management data. *IEEE Transactions on Software Engineering*, January, 1–12.

[9] K. J. Lieberherr and I. M. Holland. 1989. *Tools for Preventive Software Maintenance*. Proceedings of International Conference on Software Maintenance (ICSM), October 1989, Miami, FL. IEEE Computer Society Press, Los Alamitos, CA. pp. 1–12.

[10] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. 1995. *Software Rejuvenation: Analysis, Module and Applications*. Proceedings of the 25th symposium on Fault Tolerant

Computing, June 1995, Pasadena, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 381–390.

[11] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. 1998. Analysis of preventive maintenance in transactions based software systems. *IEEE Transactions on Computers*, January, 96–107.

[12] M. Grottke and K. S. Trivedi. 2007. Fighting bugs: remove, retry, replicate, and rejuvenate. *IEEE Computers*, February, 107–109.

[13] M. K. Mattsson. 2001. *Can We Learn Anything from Hardware Preventive Maintenance?* Seventh IEEE International Conference on Engineering of Complex Computer Systems, June 2001, Skovde, Sweden. IEEE Computer Society Press, Los Alamitos, CA. pp. 106–111.

[14] E. Marshall. 1992. Fatal error: how patriot overlooked a scud. *Science*, March 13, p. 1347.

[15] N. Chapin, J. F. Hale, K. M. Khan, J. F. Ramil, and W. G. Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 3–30.

[16] N. Chapin. 2000. *Software Maintenance Types—A Fresh View.* Proceedings of the International Conference on Software Maintenance (ICSM), October 2000, San Jose, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 247–252.

[17] C. Jones. 2007. Geriatric issues of aging software. *CrossTalk: The Journal of Defense Software Engineering*, December, 4–8.

[18] D. L. Parnas. 1994. *Software Aging*. Proceedings of 16th International Conference on Software Engineering, May 1994, Sorrento, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 279–287.

[19] K. Maxwell, L. V. Wassenhove, and S. Dutta. 1996. Software development productivity of European space, military and industrial applications. *IEEE Transactions on Software Engineering*, October, pp. 706–718.

[20] M. I. Halpern. 1965. Machine independence: its technology and economics. *Communications of the ACM*, 8(12), 782–785.

[21] R. F. Couch. 1971. Evolution of a toll mis – bell Canada. *Management Information Systems: Selected Papers from MIS Copenhagen 70—An IAG Conference* (Eds W. Goldberg, T. H. Nielsen, E. Johnson, and H. Josefsen), pp. 163–188. Auerbach Publisher Inc., Princeton, NJ.

[22] L. A. Belady and M. M. Lehman. 1976. A model of large program development. *IBM Systems Journal*, 15(1), 225–252.

[23] P. Wegner. 1978. *Research Direction in Software Technology*. Proceedings of the 3rd International Conference on Software Engineering (ICSE), May 1978, Atlanta, Georgia. IEEE Computer Society Press, Los Alamitos, CA. pp. 243–259.

[24] M. M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, September, 1060–1076.

[25] K. H. Bennett and V. T. Rajlich. 2000. *Software Maintenance and Evolution: A Roadmap.* ICSE, The Future of Software Engineering, June 2000, Limerick, Ireland. ACM, New York. pp. 73–87.

[26] L. J. Arthur. 1988. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons.

[27] K. H. Bennett and J. Xu. *Software Services and Software Maintenance*. Proceedings of 7th European Conference on Software Maintenance and Reengineering, March 2003, Benevento, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 3–12.

[28] M. M. Lehman and J. Ramil. 2002. Software evolution and software evolution processes. *Annals of Software Engineering*, 14, 275–309.

[29] S. L. Pfleeger. 1998. The nature of system change. *IEEE Software*, May/June, 87–90.

[30] S. Cook, R. Harrison, M. M. Lehman, and P. Wernick. 2006. Evolution in software systems: foundations of SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18, 1–35.

[31] M. M. Lehman. 1996. Feedback in the software evolution process. *Information and Software Technology*, 38, 681–686.

[32] M. M. Lehman and L. A. Belady. 1985. *Program Evolution: Processes of Software Change*. Academic Press, London.

[33] M. M. Lehman and J. F. Ramil. 2006. Software evolution. In: *Software Evolution and Feedback* (Eds N. H. Madhavvji, J. F. Ramil, and D. Perry). John Wiley & Sons, West Sussex.

[34] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. 1997. *Metrics and Laws of Software Evolution—The Nineties View*. Proceedings of 4th International Symposium on Software Metrics (Metrics 97), November 1997. IEEE Computer Society Press, Los Alamitos, CA. pp. 20–32.

[35] M. M. Lehman. 2001. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11, 15–44.

[36] C. K. S. Chong Hok Yuen. 1987. *A Statistical Rational for Evolution Dynamics Concepts*. Proceedings of the International Conference on Software Maintenance, September 1987, Austin, Texas. IEEE Computer Society Press, Los Alamitos, CA. pp. 156–164.

[37] F. Brooks. 1993. *The Mythical Man Month* (2nd Ed.). Addison-Wesley, Reading, MA.

[38] M. M. Lehman. 1996. *Laws of Software Evolution Revisited*. Proceedings of the 5th European Workshop on Software Process Technology, Lecture Notes in Computer Science, Vol. 1149. Springer, London, pp. 108–124.

[39] D. A. Garvin. 1984. What does product quality mean? *Sloan Management Review*, Fall, 25–45.

[40] N. H. Madhavji, J. F. Ramil, and D. E. Perry (Eds). 2006. *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, West Sussex.

[41] G. Visaggio. 2001. Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 281–308.

[42] C. K. S. Chong Hok Yuen. 1988. *On Analyzing Maintenance Process Data at the Global and Detailed Levels: A Case Study*. Proceedings of the International Conference on Software Maintenance, October 1988. Phoenix, Arizona. IEEE Computer Society Press, Los Alamitos, CA. pp. 248–255.

[43] M. M. Lehman, D. E. Perry, and J. F. Ramil. 1998. *On Evidence Supporting the Feast Hypothesis and the Laws of Software Evolution*. Proceedings of the 5th International Software Metrics Symposium (Metrics), November 1998. IEEE Computer Society Press, Los Alamitos, CA. pp. 84–88.

[44] W. M. Turski. 1996. Reference model for growth of software systems. *IEEE Transactions on Software Engineering*, August, 599–600.

[45] M. Lawrence. 1982. *An Examination of Evolution Dynamics*. Proceedings of International Conference on Software Engineering (ICSE), September 1982. IEEE Computer Society Press, Los Alamitos, CA. pp. 188–196.

[46] M. M. Lehman and J. F. Ramil. 2003. Software evolution—background, theory, practice. *Information Processing Letters*, 88(1–2), 33–44.

[47] S. Williams. 2002. *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly & Associates, Inc., Sebastopol, CA.

[48] R. M. Stallman, L. Lessig, and G. Gay. 2002. *Free Software, Free Society*. Free Software Foundation, Cambridge, MA.

[49] E. S. Raymond. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA.

[50] Y. Wang, D. Guo, and H. Shi. 2007. Measuring the evolution of open source software systems with their communities. *ACM SIGSOFT Software Engineering Notes*, 32(6), pp. 1–7.

[51] J. F. Ramil, A. Lozano, and M. Wermelinger. 2008. Empirical studies of open source evolution. In: *Software Evolution* (Eds T. Mens and S. Demeyer). Springer, Berlin.

[52] W. Scacchi. 2006. Understanding open source software evolution. In: *Software Evolution and Feedback* (Eds N. H. Madhavvji, J. F. Ramil, and D. Perry), pp. 181–2026. John Wiley & Sons, West Sussex.

[53] K. Crowston and J. Howison. 2005. The social structure of free and open source software devellopment. *First Monday*, 10(2).

[54] M. Aberdour. 2007. Achieving quality in open source software. *IEEE Software*, 24(1), 58–64.

[55] A. Mockus, R. T. Fielding, and J. D. Herbsleb. 2002. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, July, 309–346.

[56] S. S. Pirzada. 1988. A statistical examination of the evolution of the Unix system. PhD Thesis, Department of Computing, Imperical College, London, England.

[57] M. W. Godfrey and Q. Tu. 2000. *Evolution in Open Source Software: A Case Study*. Proceedings of the International Conference on Software Maintenance (ICSM), October 2000. IEEE Computer Society Press, Los Alamitos, CA. pp. 131–142.

[58] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. 2002. Maintainability of the linux kernel. *IEEE proceedings—Software*, 149(1), pp. 18–22.

[59] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. 2005. *Evolution and Growth in Large Libre Software Projects*. Proceedings of Eighth International Workshop on Principles of Software Evolution (IWPSE), September 2005, Lisbon, Portugal. IEEE Computer Society Press, Los Alamitos, CA. pp. 165–174.

[60] M. M. Lehman, J. F. Ramil, and U. Sandler. 2001. *An Approach to Modelling Long-term Growth Trends in Software Systems*. Proceedings of the International Conference on Software Maintenance (ICSM), November 2001, Florence, Italy. IEEE Computer Society Press, Los Alamitos, CA, pp. 219–228.

[61] J. W. Paulson, G. Succi, and A. Eberlein. 2004. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, April, 246–256.

[62] C. K. Roy and J. R. Cordy. 2006. *Evaluating the Evolution of Small Scale Open Source Software Systems*. 15th International Conference on Computing, Mexico City, November 2006. IEEE Computer Society Press, Los Alamitos, CA, Research in Computing Science, Vol. 23, pp. 123–136.

[63] H. Gall, M. Jazayeri, R. Klősch, and G. Trausmuth. 1997. *Software Evolution Observed Based on the Product Release History*. Proceedings of the International Conference on Software Maintenance (ICSM), October 1997, Bari, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 160–166.

[64] M. R. Vigder and J. Dean. 1997. *An Architectural Approach to Building Systems from COTS Software Components*. Proceedings of the 22nd Annual Software Engineering Workshop, December 1997, Greenbelt, MA. pp. 99–113.

[65] D. Reifer, V. Basili, B. Boehm, and B. Clark. 2003. Eight lessons learned during cots-based systems maintenance. *IEEE Software*, September/October, 94–96.

[66] J. Voas. 1998. Maintaining components-based systems. *IEEE Software*, July/August, 22–27.

[67] S. Beydeda and V. Gruhn (Eds.) 2005. *Testing Commerical-off-the-Shelf Components and Systems*. Springer, Germany.

[68] J. Morris, G. Lee, K. Parker, G. Bundell, and C. Lam. 2001. Software component certification. *IEEE Computer*, September, pp. 30–36.

[69] K. Naik and P. Tripathy. 2008. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Inc., Hoboken.

[70] C. V. Ramamoorthy and F. B. Bastani. 1982. Software reliability—status and perspectives. *IEEE Transactions on Software Engineering*, July, pp. 354–371.

[71] M. Vigder and A. Kark. 2006. *Maintaining Cots-based Systems: Start with Design*. Proceedings of the 5th International Conference on Commercial-Off-The-Shelf (COTS)-Based Software Systems, February 2006, Orlando, FL. IEEE Computer Society Press, Los Alamitos, CA. pp. 11–18.

[72] M. Vigder and J. Dean. 2000. Maintenance of cots-based systems. National Research Council of Canada, Institute for Information Technology, Ottawa, Ontario, Canada, NRC Report 43626, p. 6.

[73] D. J. Carney, S. A. Hissam, and D. Plakosh. 2000. Complex cots-based software practical steps for their maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 12, 357–376.

[74] M. Krieger, M. Vigder, J. Dean, and M. Siddiqui. 2003. *Coordination in COTS-based Development*. Second International Conference on COTS-Based Software Systems (ICCBSS), February 2003, Ottawa, Canada, LNCS-2580. Springer, pp. 123–133.

[75] D. Garlan, A. Robert, and J. Ockerbloom. 1995. Architectural mismatch: why reuse is so hard. *IEEE Software*, November, 17–26.

[76] D. Garlan, A. Robert, and J. Ockerbloom. 2009. Architectural mismatch: why reuse is still so hard. *IEEE Software*, July/August, 66–69.

[77] IEEE Standard 1219-1998. 1998. *Standard for Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA.

[78] M. K. Mattsson, U. Westblom, S. Forssander, G. Andersson, M. Medin, S. Ebarasi, T. Fahlgren, S. E. Johansson, S. Tornquist, and M. Holmgren. 2001. *Taxonomy of*

Problem Management Activities. Proceedings of the 5th European Conference on Software Maintenance and Reengineering, March, 2001, lisbon, Portugal, IEEE Computer Society Press, Los Alamitos, CA. pp. 1–10.

[79] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. 2005. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 309–332.

[80] C. F. Kemerer and S. Slaughter. 1999. An empirical approach to studying evolution. *IEEE Transactions on Software Engineering*, July/August, 493–509.

EXERCISES

1. Explain why it is important to make distinctions among the different types of software maintenance.

2. What are the causes of adaptive maintenance problems? What are the consequences of these problems?

3. Why should you fix something that is not broken? What are some reasons for performing perfective maintenance?

4. What are some common justifications for not doing perfective maintenance?

5. What is software aging? What are the common causes of software aging? How can those causes be eliminated? Discuss the answers in detail. (Hint: rejuvenate, software maintainability)

6. Explain the concept of software rejuvenation. Discuss the pros and cons of software rejuvenation.

7. Discuss the differences between hardware and software maintenance.

8. For each of the following situations, explain whether it is a hazard or a mishap.

(a) Water in a swimming pool becomes electrified.

(b) A room fills with carbon dioxide.

(c) A car stops abruptly.

(d) A long distance telephone company suffers an outage.

(e) A nuclear weapon is destroyed in an unplanned manner.

9. Compare the software maintenance classification based on intention, activity, and evidence.

10. Explain the rationale behind Lehman's laws. Under what circumstances those laws may not hold?

11. What is software entropy? Explain its relationship with software aging.

12. What is code decay? Discuss the symptoms of code decay. What are the causes of code decay?

13. Explain the term software cloning and discuss its significance with respect to software evolution.

14. Explain the inverse square model of system evolution. For the system data given in Table 2.6, plot the graph for the actual and calculated (inverse square model) sizes for the system.

TABLE 2.6 System Data to be Used in Question 14

RSN	Size	RSN	Size	RSN	Size
1	977	8	1800	15	2151
2	1344	9	1595	16	2091
3	1390	10	1897	17	2095
4	1226	11	1832	18	2101
5	1246	12	1897	19	2312
6	1492	13	1902	20	2167
7	1581	14	2087	21	2315

Source: Data, taken from Reference 44. © 1996 IEEE.

15. Discuss the differences between FOSS and CSS software evolution system.

16. Suppose that you wish to construct a system by combining the functionality of two COTS-based products together. Requests must be handled by sending them first to one of the COTS products, and sending the results from that product to the second. The results from this second product can then be returned to the original requester. Which standard middleware component would you use in building this system?

(a) glue

(b) wrapper

(c) mediator

(d) tailoring

17. Identify the cluster and evidence maintenance type for the following scenario. Explain your answer.

"A maintainer was assigned to install three upgraded COTS components, the first of which implements a hardware upgrade. Each COTS component was received from a different vender, but all are used in one system. In attempting to install a test version of the system, the maintainer found that one of the upgrades was incompatible with the other two, and that those other two would not work with the existing in-use version of the third. After considerable diagnostic test runs, and obtaining an "its your problem" response from the vendor of the third component, the maintainer got approval to write a wrapper for one of

the upgraded COTS components in order to fit with the continued use of the existing version of the third component. After successful regression testing, the maintainer had the configuration management data updated and the new wrapper recorded. The change, since it would be nearly transparent to the customer, was put into production at the time of the next scheduled release of a version of the systems, and the document was updated."