3

EVOLUTION AND MAINTENANCE MODELS

People seldom improve when they have no other model but themselves to copy after.
—Oliver Goldsmith

3.1 GENERAL IDEA

The software production processes comprise a set of activities starting from conception to retirement. There are many software processes, differing primarily in their classifications of phases and activities. One traditional software development life cycle (SDLC) is shown in Figure 3.1, which comprises two discrete phases, namely, development and maintenance, the latter commonly approaching two-thirds of the product life span. As this diagram shows, about one-fourth to one-third of all software life cycle costs are attributed to software development, and the remaining cost is due to operations and maintenance. Note that the percentages in Figure 3.1 indicate relative costs. As listed below [1], software maintenance has unique characteristics, although many activities related to maintaining and developing software are similar:

- *Constraints of an existing system.* Maintenance is performed on an operational system. Therefore, all modifications must be compatible with the constraints of the existing software architecture, design, and code.
- *Shorter time frame.* A maintenance activity may span from a few hours to a few months, whereas software development may span 1 or more years.

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

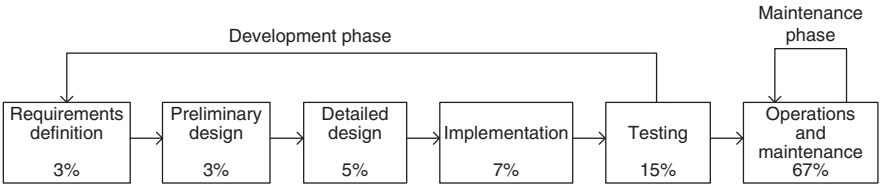

FIGURE 3.1 Traditional SDLC model. From Reference 1. © 1988 John Wiley & Sons

- *Available test data.* In software development, test cases are designed from scratch, whereas software maintenance can select a subset of these test cases and execute them as regression tests. Thus, the challenge is to select appropriate test cases from the existing test suite. In addition to the regression test cases, new test cases need to be created to adequately test the code changes.

Therefore, software maintenance should have its own software maintenance life cycle (SMLC) model as it involves many unique activities. On the other hand, software maintenance has got many similarities with software development, with a focus on product enhancement and correction, in addition to transforming requirements to software functionality. In this chapter, three maintenance models will be explained: reuse, simple staged, and change mini-cycle, representing, respectively, the old, relatively new, and still in research models. We examine in detail two standards, IEEE/EIA 1219 and ISO/IEC 14764, to manage and execute software maintenance activities.

Software maintenance is at the heart of an evolving software product. Evolution, change, and system configuration complicate maintenance activities. The software product which is released to a customer is in the form of executable code, whereas the corresponding "product" within the supplier organization is source code. Source code can be modified without affecting the executable version in use. Thus, strict control must be kept, otherwise exact source code representation of a particular executable version may not exist. In addition, documentation associated with the executable code must be compatible, otherwise the customer may not be able to understand the system. Therefore, tight documentation control is necessary. In other words, the set of products that are released to the customer must be controlled. Software configuration management (SCM) is the way by which the process of software evolution is controlled. SCM provides a framework for managing changes in an efficient way. The functionalities and best practices of SCM are discussed in this chapter. In addition, we discuss a state transition model of a modification (or, change) request, as it flows through the organization.

3.2 REUSE-ORIENTED MODEL

One obtains a new version of an old system by modifying one or several components of the old system and possibly adding new components. As a consequence, the new system is likely to reuse many components of the old system. A new version of

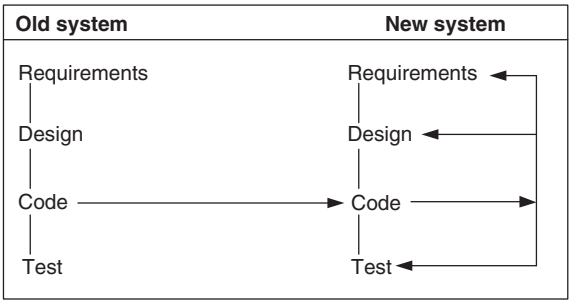

FIGURE 3.2 The quick fix model. From Reference 2. © 1990 IEEE

the system can be created after the maintenance activities are implemented on some of the old system's components. Based on this concept, three process models for maintenance have been proposed by Basili [2]:

- *Quick fix model.* In this model, necessary changes are quickly made to the code and then to the accompanying documentation (Figure 3.2).
- *Iterative enhancement model.* In this model, as illustrated in Figure 3.3, first changes are made to the highest level documents. Eventually, changes are propagated down to the code level.
- *Full reuse model.* In this model, as illustrated in Figure 3.4, a new system is built from components of the old system and others available in the repository.

The old system is reused by all of the three aforementioned models, and, therefore, those belong to the reuse-oriented paradigm. The models assume that the descriptions of the existing system are complete and consistent.

Quick fix model. This model embodies a commonly used approach to software maintenance. In this model, as illustrated in Figure 3.2, (i) source code is modified to

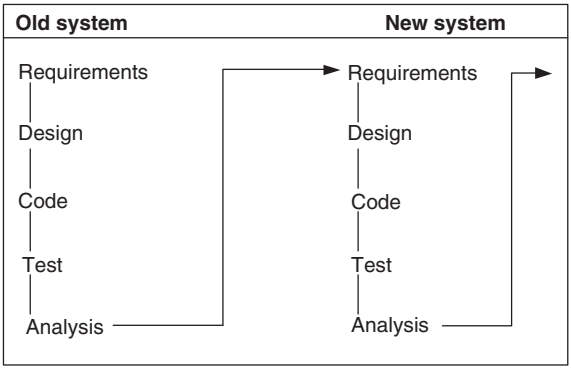

FIGURE 3.3 The iterative enhancement model. From Reference 2. © 1990 IEEE

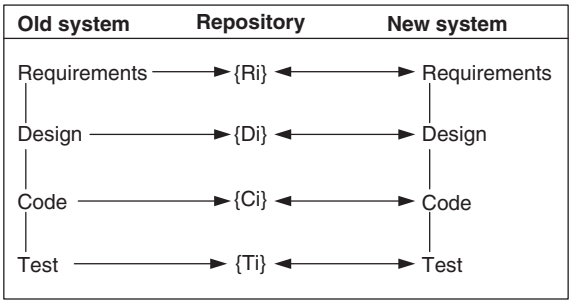

FIGURE 3.4 The full reuse model. From Reference 2. © 1990 IEEE

fix the problem; (ii) necessary changes are made to the relevant documents; and (iii) the new code is recompiled to produce a new version. Often changes to the source code are made with no prior investigation such as analysis of impact of the changes, ripple effects of the changes, and regression testing. Moreover, resource constraints often entail that modifications performed to the code are not documented.

Iterative enhancement model. This model is based on the Japanese principle of *Kaizen*, which means the incremental and progressive improvement of practices. Iterative and incremental development methodologies were practiced in early 1950s, before Winston Royce's Waterfall model [3] was widely used. An alternative approach to software maintenance is suggested by the iterative and incremental models. Those two models have the following ideas in common: (i) it is difficult to fully comprehend a large set of requirements for a system and (ii) developers may find it difficult to build the full system in one go. Therefore, a complete system is developed in progressively larger builds, where one build refines the requirements of the preceding build by taking user inputs into account [4]. The iterative enhancement model, explained in Figure 3.3, shows how changes flow from the very top-level documents to the lowest-level documents. The model works as follows:

- It begins with the existing system's artifacts, namely, requirements, design, code, test, and analysis documents.
- It revises the highest-level documents affected by the changes and propagates the changes down through the lower-level documents.
- The model allows maintainers to redesign the system, based on the analysis of the existing system.

Remark: The terms iteration and increment are liberally used when discussing iterative and incremental development. However, they are not synonyms in the field of software engineering. On the one hand, iteration implies that a process is basically cyclic, thereby meaning that the activities of the process are repeatedly executed in a structured manner. On the other hand, increment implies some quantifiable outcome of an iteration. Iterative development is based on scheduling strategies in which time is set aside to improve and revise parts of the system under development.

Incremental development is based on staging and scheduling strategies in which parts of the system are developed at different times and/or paces and integrated as they are completed.

The model is effectively a three-phase cycle: analysis, characterization of proposed enhancements, and redesign and implementation. A new build is constructed by starting with an analysis of the existing system's requirements, followed by design, coding, and testing. Next, documents at all levels, which are affected by the changes, are modified. Reuse, as explained in Chapter 9, is explicitly supported by the model. The model also accommodates the quick fix model. The iterative enhancement model gives us the key advantage that documentation is kept up-to-date with changes made to the code.

With replicated controlled experiments, Visaggio [5] compared the iterative model and the quick fix model with respect to maintainability. It has been shown that maintainability of systems degrade faster with the quick fix model. In addition, the iterative enhancement model enables organizations to perform maintenance modifications faster than those adopting the quick fix model. In general, an organization may adopt the quick fix model if they do not have time. Therefore, the latter observation is counterintuitive.

Full reuse model. The model illustrated in Figure 3.4 shows maintenance as a special case of reuse-based software development. The main assumption in this model is the availability of a repository of artifacts describing the earlier versions of the present and similar systems. Full reuse comprises two major steps:

- perform requirement analysis and design of the new system; and
- use the appropriate artifacts, such as requirements, design, code, and test from any earlier versions of the old system.

In the full reuse model, reuse is explicit and the following activities are performed:

- identify the components of the old system that are candidates for reuse;
- understand the identified system components;
- modify the old system components to support the new requirements; and
- integrate the modified components to form the newly developed system.

3.3 THE STAGED MODEL FOR CLOSED SOURCE SOFTWARE

Rajlich and Bennett [6] have defined a simple *staged* model to represent the traditional commercial Closed Source Software (CSS) life cycle. Their model comprises a sequence, as illustrated in Figure 3.5, of five stages:

- *Initial development.* Develop the first functioning version of the software.
- *Evolution.* The developers improve the functionalities and capabilities of the software to meet the needs and expectations of the customer.

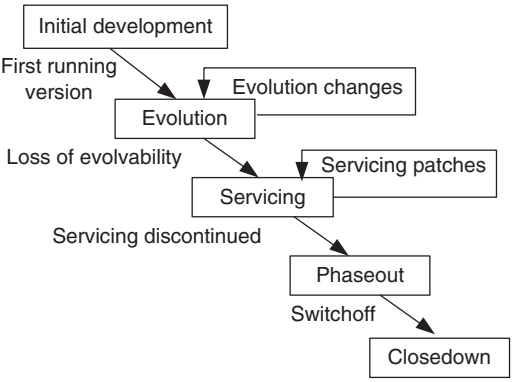

FIGURE 3.5 The simple staged model for the CSS life cycle. From Reference 6. © 2000 IEEE

- *Servicing.* The developers only fix minor and emergency defects, and no major functionality is included.
- *Phaseout.* In this phase, no more servicing is undertaken, while the vendors seek to generate revenue as long as possible.
- *Closedown.* The software is withdrawn from the market, and customers are directed to migrate to a replacement.

Initial development. Software developers build the first version of the system from scratch to satisfy the initial requirements. The initial development includes design, initial coding, and testing. Generally, no releases are made public to the customers in this stage. The first version may lack some functionality, but it lays two important foundations for future iterations, namely, *the software architecture* and *the team knowledge*:

- *The software architecture.* The components of the software, the interactions among them, and their desired properties, such as efficiency and functionality, continue to stay intact through the remains of the life cycle of the system.
- *The team knowledge.* During initial development, the software engineering team acquires knowledge about the application domain, user requirements, business process, data formats, algorithms, weaknesses and strengths of the software architectures, and execution environment. For the subsequent stages of the life-cycle of the software system, this knowledge is considered to be crucial.

Evolution. The software system moves to the evolution stage after the initial development is successful. Software developers extend the functionalities and capabilities of the system to meet the needs and expectations of the customers. In this stage: (i) quick patches and new releases are dispatched to the customers and (ii) feedback from the customers are received for additional enhancement to the software system.

Customer demands for additional functionalities and competitive products from other vendors cause the system to evolve. In addition, evolution of the system may occur due to changes in the operating environment and the business practice. An example of change in the business practice is to target enterprise markets instead of the service provider market segment. Sometimes, the developing company releases the software system right after the initial development. However, often a system is released in its evolution phase after it has undergone many quality improvement cycles. For example, reliability and stability are improved during a system's quality improvement cycle. The exact release date for the product is based on several factors such as timeliness, quality, innovation, and business goals of the company [7].

Servicing. For software to evolve easily, it has to have an appropriate architecture and the software team has to have the necessary expertise. When either architectural integrity or the expertise of the architecture is missing, the software ceases to easily evolve, and it makes a transition to its servicing stage. The system is viewed to have *aged* or *decayed* in the servicing stage. In this stage, the software is considered to have matured and simple modifications are made to the source code, without providing user perceivable enhancements. Changes in this stage are expensive and difficult. Therefore, software developers minimize the number of changes or use wrappers as a way to effect changes. Each of these changes further weakens the system architecture, thereby increasing the need for further servicing. Chapin et al. [8] refer to the servicing stage as the real maintenance phase. After considering the economic profitability of the system, a decision is made to transition the system from the evolution stage to the servicing stage. When new revenues from a software system do not justify the cost of performing modifications, the system is designated as a *legacy* system and it is no more evolved.

Phaseout. During the phaseout stage, the supplier may decide to not perform any more servicing. The software may still be in use, but because change requests (CRs) are no longer honored, it is becoming increasingly outdated. The users must work around the known deficiencies of the system more often. Going back to an earlier servicing stage becomes very difficult because of the increasingly large number of CRs. Eventually, the software system becomes a *legacy* system application.

Closedown. During the final shut down, the vendor pulls out the software product from the market and makes recommendations to the customers for alternative solutions. The supplier may have certain pending contractual responsibilities, namely legal obligations and source code retention. In the areas of outsourced software, source code retention is an important responsibility. As the software system moves from the phaseout to the closedown stage and if the software is still found to be businessworthy to its stakeholders, the system is called a legacy system. For a legacy system, it is prudent to move to a newer system which provides similar functionalities, without exhibiting the poor quality of the legacy system.

One version of the staged model for CSS is called *versioned staged model* and it has been illustrated in Figure 3.6. The model shown in Figure 3.6 has essentially the same stages as found in Figure 3.5, but separate evolution tracks from the initial development are found in Figure 3.6. The evolution process is the backbone of the model. Each evolution track includes servicing, phaseout, and closedown. At certain

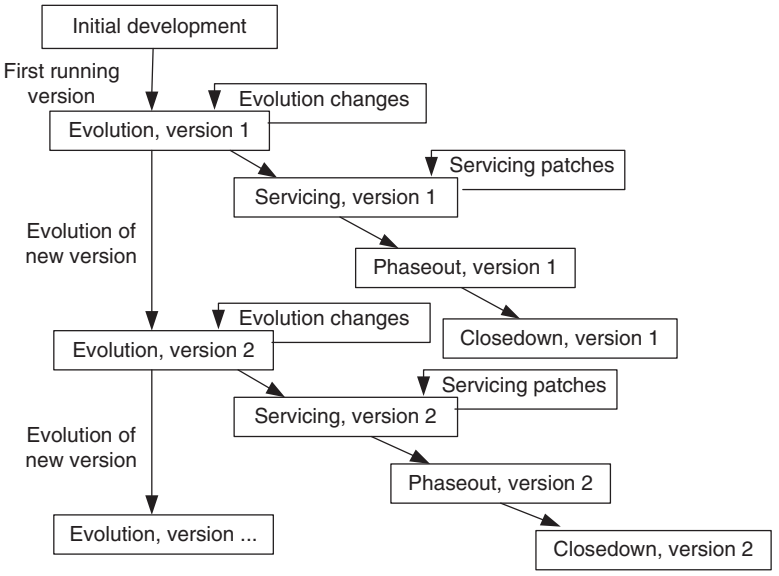

FIGURE 3.6 The versioned staged model for the CSS life cycle. From Reference 6. © 2000 IEEE

time frames, a version of the software is completed and released to the customers. The evolution of the software does not stop at that point; rather, it continues and eventually produces the next version. The released version is no longer evolved but only serviced. Many organizations use a scheme such as <product><version><release><build>, where version reflects the strategic changes made to the system during evolution, release reflects the servicing patches, and build reflects the, say, daily internal build of the software.

3.4 THE STAGED MODEL FOR FREE, LIBRE, OPEN SOURCE SOFTWARE

Capiluppi et al. [9] revised the staged model for its applicability to Free, Libre, Open Source Software (FLOSS) systems, as shown in Figure 3.7. The authors provide empirical evidence to justify the FLOSS model. The model benefits developers by characterizing FLOSS systems in terms of stages and indicating which stage the system is currently in and to which stage the system is more likely to transition.

Three major differences are identified between CSS systems and FLOSS systems. The first one is related to the availability of releases. CSS systems are available to the customers in a running condition after having been tested enough. On the other hand, a FLOSS system is posted on the versioning system repositories much before the official release. Therefore, binaries as well as source code can be downloaded not only by end users but by developers as well. The revised model shown in Figure 3.7 reflects

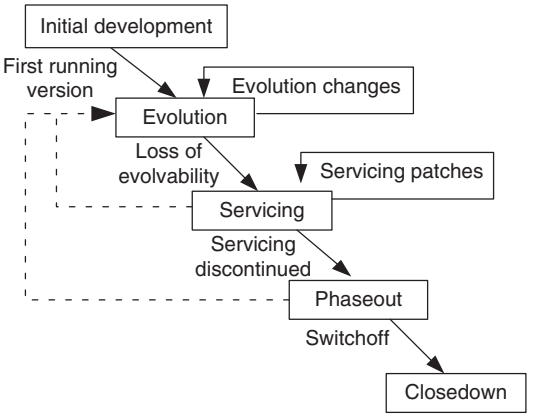

FIGURE 3.7 The staged model for the FLOSS system. From Reference 9. © 2007 ACM

the aforementioned difference between FLOSS and CSS systems. In Figure 3.7, the rectangle with the label "Initial development" has been visually highlighted because it can be the only initial development stage in the evolution of FLOSS systems. In other words, it does not have any evolution track for FLOSS systems.

The second difference concerns the transition from the evolution to the servicing stage. Based on the empirical data from several FLOSS systems, it was observed that a new development stage is reached following a phase without much enhancements. With some systems that were analyzed, after a transition from evolution to servicing, a new period of evolution was observed. This possibility is depicted in Figure 3.7 as a broken arc from the servicing stage to the evolution stage.

The third difference is a possibility of a transition from phaseout stage to evolution stage for FLOSS systems. A case study of a FLOSS system was illustrated by Capiluppi et al. [9]. In the said case study, a new team of developers took over the maintenance task that was abandoned by the previous developed team. In general, the active developers of FLOSS systems get frequently replaced by new developers. Therefore, the dashed line in Figure 3.7 exhibits this possibility of a transition from phaseout stage to evolution stage.

3.5 CHANGE MINI-CYCLE MODEL

Software change is a fundamental ingredient of software evolution and maintenance. Let us revisit the first law of software evolution which is stated as "A program undergoes continuing changes or becomes less useful. The change process continues until it becomes cost-effective to replace the program with a re-created version." The CCS staged model discussed earlier is based on the above fundamental premise. The difficulty of software changes distinguishes the two stages: evolution and servicing. Whereas substantial software changes are allowed in the evolution stage, in the Servicing stage limited changes are permitted. Note that iterative modification is

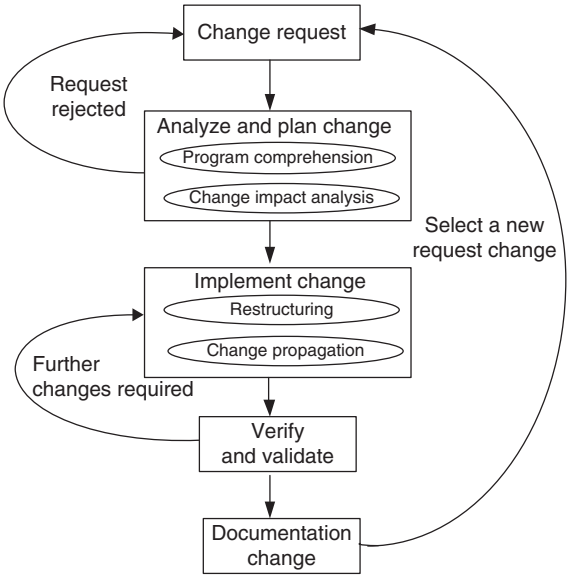

FIGURE 3.8 The change min-cycle. From Reference 12. © 2008 Springer

the primitive building block from which both the evolution and servicing stages are derived.

Software change is a process that may introduce new requirements to the existing system. In addition, there may be a need to alter the software system if the requirements are not correctly implemented. In order to capture this, an evolutionary model, known as *change mini-cycle* (Figure 3.8), was proposed by Yau et al. [10] in the late 1970s and revisited by other researchers, namely, Bennet et al. [11] and Mens [12]. The change mini-cycle model consists of five major phases: CR, analyze and plan change, implement change, verify and validate, and documentation change. In this process model, new significant activities were identified to reflect the fact that software changes are rarely isolated. Examples of those new activities are change impact analysis and change propagation. These activities continue to be the subjects of research.

Change request. A CR generally originates from the management, users of the software, or customers. A CR may take one of the following two forms: defect report and enhancement request.

- A defect report describes the defect and software system actions that are out of line with requirements.
- An enhancement request describes a change to the requirements, functionality, or quality of the system.

The above two items were in the focus of practitioner's concerns that can be traced back to the circa 1972 article "That maintenance 'iceberg'" by Canning [13].

According to the said article, practitioners observe maintenance narrowly as correcting errors and broadly as expanding and extending software functionality. In this book, CR refers to both the aforementioned views. The CR document must capture a minimal set of information about changes to software, hardware, and documentation.

Analyze and plan change. In the second phase, program comprehension and impact analysis are conducted. Program comprehension [14] is essential to understanding which parts of the software will be affected by a CR. Program comprehension is basically a process of acquiring useful information from source code. One such information is the location of the domain-specific concept in the source code [15, 16]. The code implementing the concepts may need to be changed in order to provide a solution to the CR. Concepts are units of human knowledge that can be processed by the human mind in one instance. As an example, let us consider the CR "Add a debit card payment issued by Chautauqua bank to the ATM system." In order to change the implementation, the maintenance engineer must locate those system components that implement the concepts "debit card," "payment," and "issued" embedded in the CR. The idea here is to identify the set of system components that are thought to be initially affected by the CR [17]. The identified system components are called Starting Impact Set (SIS) which is discussed in Chapter 6. A thorough discussion of program comprehension is given in Chapter 8.

Impact analysis is conducted to identify the potential consequences of a change and estimate the resources needed to accomplish the change [18]. By means of impact analysis, a software system is analyzed by maintenance personnel to identify the software components that will be affected by a CR. In this analysis, first decide if the components, which are neighbors of the SIS, also need to be modified due to the *ripple effect* [19]. A neighboring component is added to the set if it needs to be modified. For the newly added component, identify which of its neighboring components will be modified, and add them to the set. The process of identifying new components to be modified is repeated until it is found that a modification will not impact new neighboring components. The resulting set of components estimated to be modified is known as the estimated impact set (EIS). The objectives of impact analysis are as follows:

- to determine the set of system components to be affected, given the SIS identified by program comprehension activity;
- to develop accurate estimates of the resources needed to accomplish the implementation task; and
- to analyze the cost and benefits of the CR and make a decision on whether or not to implement the CR.

Software developers use the information gathered from impact analysis in planning how to implement a CR. Moreover, the goal of impact analysis is to minimize unexpected *side effects* of change. A side effect is an error or an undesirable behavior that occurs as a result of a modification in the software [20]. Chapter 6 discusses impact analysis in greater detail.

Implement change. The CR is implemented after the feasibility of a change is established. However, before the implementation of the CR, *restructuring* or *refactoring* of the software is performed in order to accommodate the requested modification. Refactoring is essentially the object-oriented variant of restructuring [21]. Restructuring is most often required in software maintenance; otherwise, systems lose structure. Restructuring is a means of restoring order to understand and change; the restructured product is less susceptible to error when future changes are made. Refactoring, discussed in Chapter 7, improves the software structure without changing their behavior.

Implementing a change comprises a number of steps, each focusing on one specific software component after the completion of refactoring. If a component is changed, it may cease to be compatible with the components with which it interacts. Therefore, non-essential changes must be made in the interacting components, thereby creating a ripple effect throughout the system. The aforementioned activity, generally called a change propagation activity [22, 23], ensures that a modification performed in one component is completely reflected throughout the entire system. Chapter 6 discusses change propagation in greater detail.

Verify and validate. In this phase the software system is verified and validated in order to assure that the integrity of the system has not been compromised. This activity includes code review, regression testing, and execution of new tests if necessary. Regression testing comprises a subset of the unit-, integration-, and system-level tests [24]. If the results are unsatisfactory, then the actualization of the request is rejected which in turn is investigated and further changes are implemented.

Documentation change. The final phase of the change mini-cycle deals with updating the program documentation. It is time to complete the documentation aspect which may include updating the requirements, functional specifications, and design specifications to be consistent with the code. In addition, user manuals and installation and troubleshooting guides are accordingly updated.

3.6 IEEE/EIA MAINTENANCE PROCESS

The IEEE/EIA 1219 standard [25] explains a process for executing and managing activities for software maintenance. The standard basically explains maintenance as a fundamental life cycle process and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity" (p. 6–1 of Reference 26). The standard focuses on a seven-phase activity model of maintenance as illustrated in Figure 3.9. The seven phases are listed below:

- Identification of problems
- Analysis
- Design
- Implementation

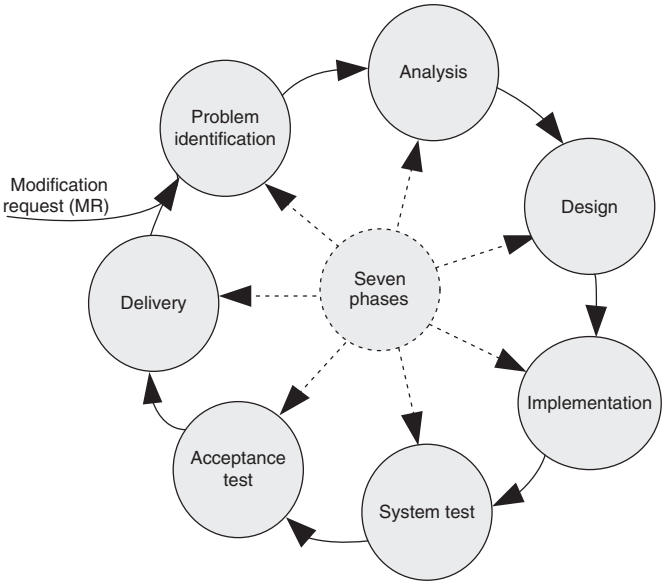

FIGURE 3.9 Seven phases of IEEE maintenance process. From Reference 26. © 2004 IEEE

- System test
- Acceptance test
- Delivery

Each of the seven activities have five associated attributes as follows:

- *Activity definition.* This refers to the implementation process of the activity.
- *Input.* This refers to the items that are required as input to the activity.
- *Output.* This refers to the items that are produced by the activity.
- *Control.* This refers to those items that provide control over the activity.
- *Metrics.* This refers to the items that are measured during the execution of the activity.

Problem identification. A request for change to the software is normally made by the users of the software system or the customers, and it starts the maintenance process. The request for change is submitted in the form of a modification request (MR) for a correction or for an enhancement. It may be noted that MR and CR are interchangeably used in maintenance literature. The maintenance (or sustaining) organization: (i) determines the type of request; (ii) determines the appropriate maintenance category – corrective, adaptive, or perfective; (iii) assigns a priority level; and (iv) assigns a unique identification number. Activities included in this phase are as follows: (i) reject or accept the MR; (ii) identify and estimate the resources needed to change the system; and (iii) put the MR in a batch of changes scheduled

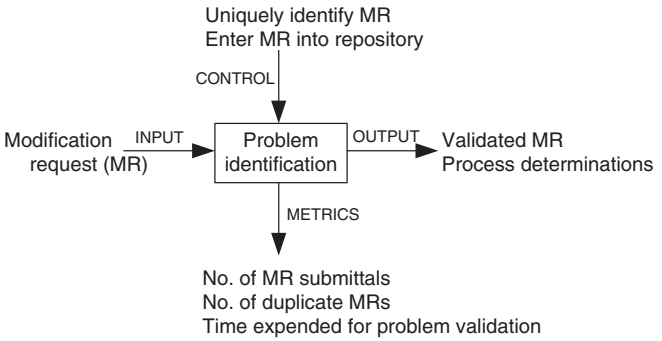

FIGURE 3.10 Problem identification phase

for implementation. The process of collecting and reviewing MRs, such as number of MR submitted and number of MR rejected, begins in this phase. For the problem identification phase, the input, output, control, and metrics have been summarized in Figure 3.10.

Analysis. The inputs to this phase are a validated MR, an initial resource estimation, repository information, and project documentation. Repository is the location in which all software-related artifacts are stored. The process is viewed to have two major components: feasibility analysis and detailed analysis. First, feasibility analysis is performed to (i) determine the impact of the change, (ii) investigate other possible solutions including prototyping, (iii) assess both short-term and long-term costs, and (iv) determine the benefits of making the change. After selecting a specific approach, the second phase of detailed analysis is undertaken. The second phase identifies (i) firm modification requirements, (ii) the software components involved, (iii) an overall test strategy, and (iv) an implementation plan. The standard puts emphasis on at least three levels of tests: unit, integration, and acceptance. In addition, regression tests are associated with each of the three levels of tests. Figure 3.11 summarizes input, control, metrics, and output for the analysis phase.

Upon completion of the analysis phase, a number of actions are taken: (i) risk analysis is performed; (ii) the preliminary resource estimate is updated; and (iii) by involving the customer, it is decided whether or not to proceed on to the next phase. If it is decided to move on to the next phase, the phase deliverables, including a detailed analysis report, are specified. The standard suggests several metrics to be gathered, such as the number of requirement changes, elapsed time, and the error rate generated.

Design. A modification to the system begins in this phase based on the information gathered up to this point. The information includes system and project documentation, the output of the analysis phase, existing software, and repository information. Activities of this phase are as follows: (i) identify the affected software components; (ii) modify the software components; (iii) document the changes; (iv) create a test suite for the new design; and (v) select test cases for regression testing. This phase provides a revised design baseline, revised test plans, an up-to-date detailed analysis report, revised risk analysis, and verified requirements. Figure 3.12 summarizes the input, output, metrics, and control for the design phase.

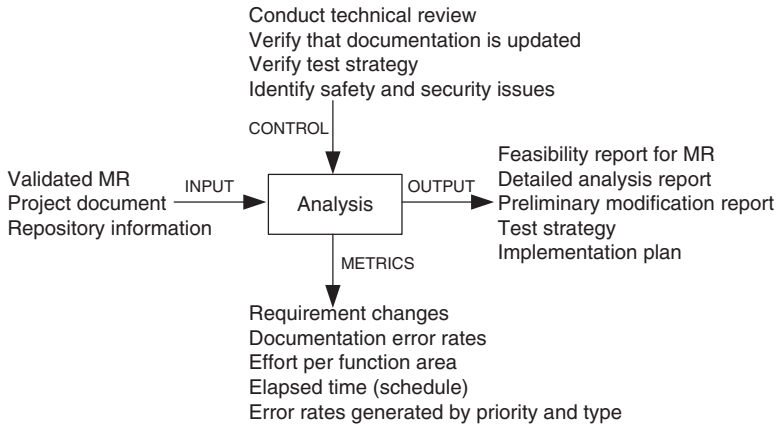

FIGURE 3.11 Analysis phase

Implementation. The design phase produces the primary inputs to this phase. The activities executed in this phase are: writing new code and performing unit testing, integrating changed code, conducting integration and regression testing, performing risk analysis, and reviewing the system for test readiness. To assess whether or not the system is ready for system-level testing, a review is performed in this phase. In this phase, risk analysis and reviews are periodically performed, rather than at the end of the phase. Multiple reviews need to be performed due to the fact that a large percentage of design, performance issues, risks, and cost are exposed while changing the system. All documentations, including the software, design, test, user, and training

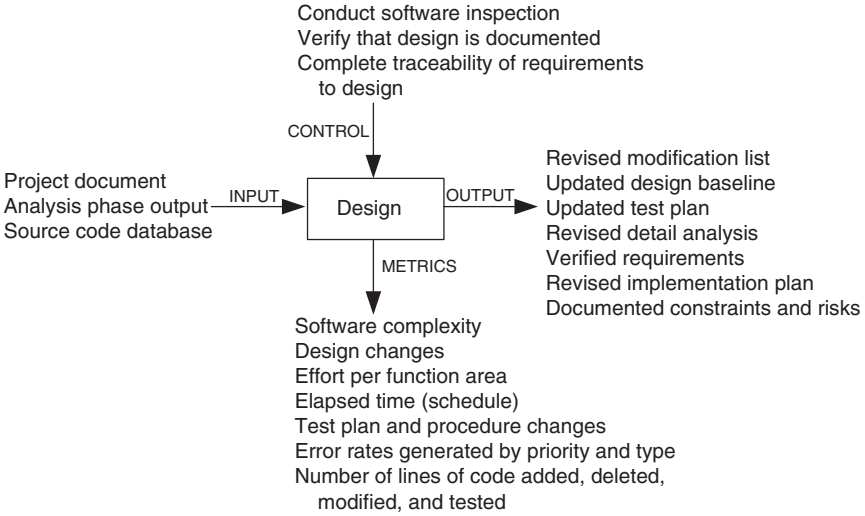

FIGURE 3.12 Design phase

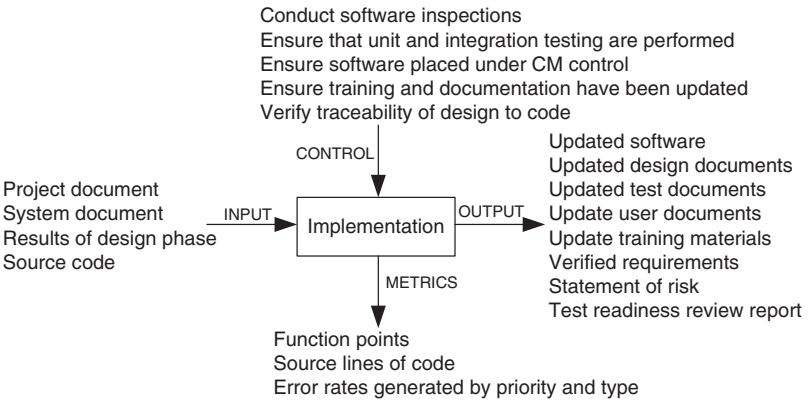

FIGURE 3.13 Implementation phase

information are updated. For the implementation phase, the input, output, metrics, and control are summarized in Figure 3.13.

System test. In this phase, tests are performed on the full system to ensure that the modified system complies with the original requirements as well as the new modifications. System-level testing comprises a broad spectrum of testing activities: functionality testing, robustness testing, stability testing, load testing, performance testing, security testing, and regression testing. Regression testing is conducted to validate that no new faults have been introduced. Quite often, during the maintenance process, the sustaining test engineers execute the system test cases [24]. Finally, the maintenance personnel verify whether or not the system is ready to perform acceptance testing. This phase accepts as its input a system test plan consisting of detailed test cases, test readiness review report, and an updated system. This phase provides a test report, a fully integrated tested system, and test readiness review report. For the system test phase, the input, output, metrics, and control are summarized in Figure 3.14.

Acceptance test. Acceptance testing is performed on a completely integrated system, and it involves customers, users, or their representatives. The main objective of

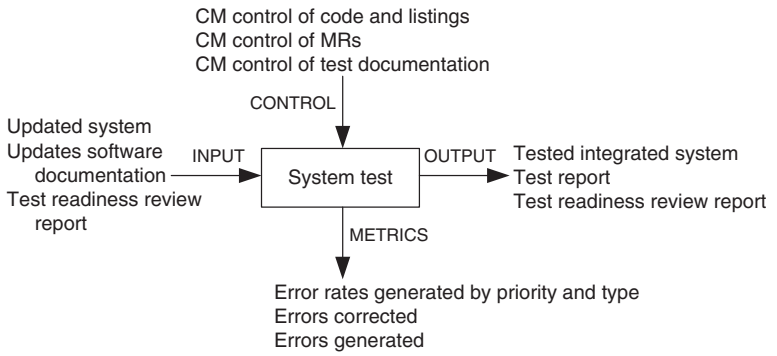

FIGURE 3.14 System test phase

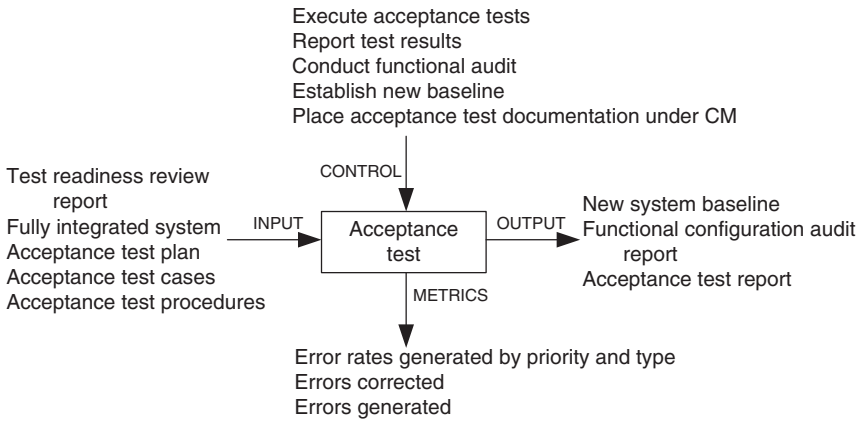

FIGURE 3.15 Acceptance test phase

acceptance testing is to assess the overall quality of the system, rather than actively identify defects. As an aside, on the other hand, the objective of system testing is to search for defects [24]. An important concept in acceptance testing is the customer's expectation from the system. The primary inputs to this phase are the test readiness review report, a fully integrated system, and a test plan with detailed test cases for acceptance testing. At the end of acceptance testing, a test report is generated. The report explains the status of the criteria that was agreed upon for successful completion of acceptance testing. The status report is communicated to the committee responsible for review. The customer chairs the review committee to evaluate the exit criteria and the test report to make sure that the system is ready for a release. For the acceptance test phase, the input, output, metrics, and control are summarized in Figure 3.15.

Delivery. In this phase, the changed system is released to customers for installation and operation. Included in this phase are the following activities: notify the user community, perform installation and training, and develop an archival version of the system for backup. For the delivery phase, the input, output, metrics, and control are summarized in Figure 3.16.

Guidelines on maintenance practices are also recommended by the standard in its appendices. For example, guidelines for maintenance practices include a guideline to make a maintenance plan; Table 3.1 shows the key sections of a maintenance plan.

3.7 ISO/IEC 14764 MAINTENANCE PROCESS

The document ISO/IEC 14764 [27] is an international standard for software maintenance, and it describes maintenance using the same concepts as IEEE/EIA 1219 except that they are depicted slightly differently. An iterative process to execute and manage maintenance activities is described in the document. The basic structure of an ISO process is made up of activities, and an activity is made up of tasks. To change

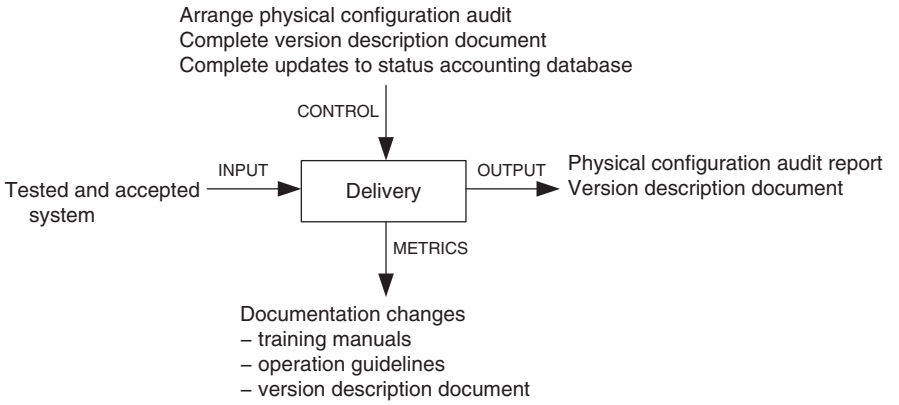

FIGURE 3.16 Delivery phase

an operational software without breaking its integrity, the necessary activities are described in the maintenance process.

Upon an activation of the maintenance process, plans and procedures are developed and resources are allocated to carry out maintenance. In response to a CR, code is modified in conjunction with the relevant documentation. Modification of the running software without losing the system's integrity is considered to be the overall objective of maintenance. The maintenance process enables the software product to migrate from its initial environment at its inception to new environments. The maintenance process is terminated upon the eventual decommissioning of the product, commonly known as being retired. The maintenance process comprises the following high-level activities:

1. Process implementation
2. Problem and modification analysis
3. Modification implementation
4. Maintenance review and acceptance
5. Migration
6. Retirement

The maintenance process activities developed by ISO/IEC are shown in Figure 3.17. Each of these activities is made up of tasks, and each task describes a specific action with inputs and outputs. A task specifies *what to do, but not* how to do [28]. Inputs refer to the items that are used by the maintenance activity to generate outputs. Effective controls are needed to provide useful guidance so that the maintenance activity produces the desired outputs. Outputs are objects generated by the maintenance activity. Support refers to the items that support the maintenance activity.

Process implementation. This activity establishes plans and procedures to be followed. A maintenance plan is made concurrently with the plan for development. Figure 3.18 graphically summarizes the process implementation activity with the

TABLE 3.1 Template of a Maintenance Plan

1. Introduction

This section outlines the goals, purpose, and general scope of the maintenance effort. Also, deviations from the standard are identified.

2. References

The documents that impose constraints on the maintenance effort are identified in this section.

In addition, other documents supporting maintenance activities are identified.

3. Definitions

All terms required to understand the maintenance plan are defined in this section.

If some terms are already defined in other documents, then references are provided to those documents.

4. Software Maintenance Overview

This section briefly describes the following aspects of the maintenance process:

4.1 Organization
4.2 Scheduling Priorities
4.3 Resource Summary
4.4 Responsibilities
4.5 Tools, Techniques, and Methods

5. Software Maintenance Process

This section describes the actions to be executed in each phase of the maintenance process. Each action is described in the form of input, output, process, and control.

5.1 Problem Identification/Classification and Prioritization
5.2 Analysis
5.3 Design
5.4 Implementation
5.5 System Testing
5.6 Acceptance Testing
5.7 Delivery

6. Software Maintenance Reporting Requirements

This section briefly describes the process for gathering information and disseminating it to members of the maintenance organization.

7. Software Maintenance Administrative Requirements

Describes the standard practices and rules for anomaly resolution and reporting.

7.1 Anomaly Resolution and Reporting
7.2 Deviation Policy
7.3 Control Procedures
7.4 Standards, Practices, and Conventions
7.5 Performance Tracking
7.6 Quality Control of Plan

8. Software Maintenance Documentation Requirements

Describes the procedures to be followed in recording and presenting the outputs of the maintenance process.

Source: From Reference 25.

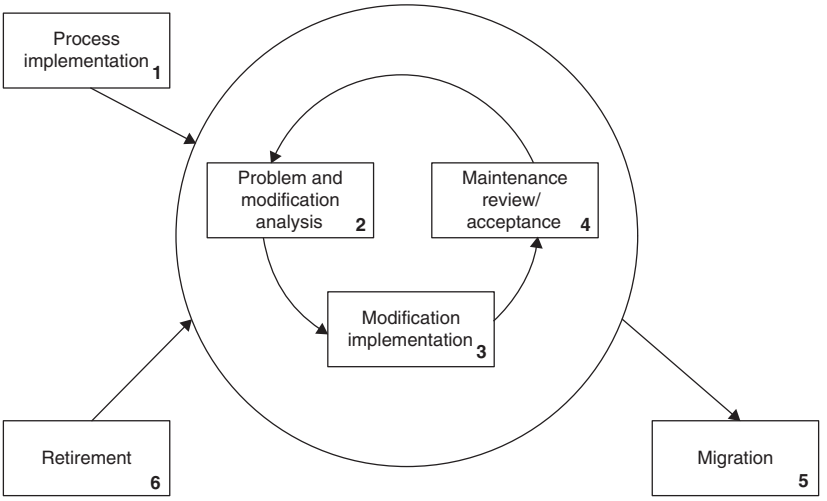

FIGURE 3.17 ISO/IEC 14764 iterative maintenance process. From Reference 26. © 2004 IEEE

input, output, control, and support items. The process implementation activity consists of three major tasks as explained in the following:

Maintenance plan: The maintenance plan describes a strategy to maintain the system, whereas the procedures for maintenance describe in details how to actually accomplish maintenance. The plan also describes how to: (i) organize and staff the maintenance team; (ii) assign responsibilities among team members; and (iii) schedule resources. The main idea is to provide cost-effective support to the maintenance team.

Modification requests: Users submit modification (or change) requests to communicate with the maintainer. The maintainer establishes procedures to receive, record, and track user requests for modifications and giving them feedback. The problem resolution process is initiated whenever an MR is received. MRs are

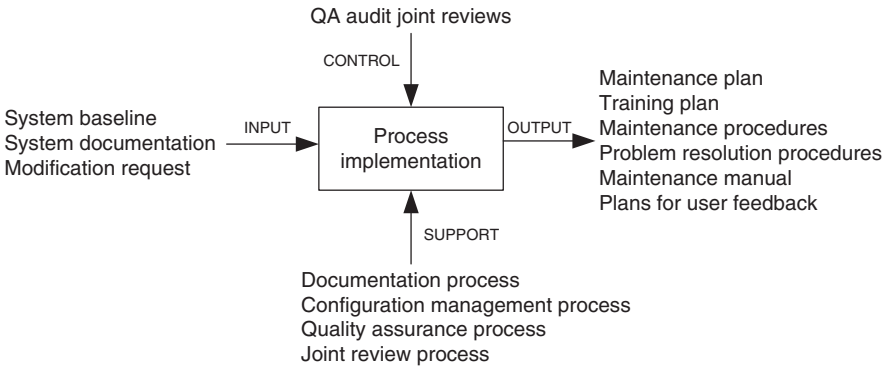

FIGURE 3.18 Process implementation activity

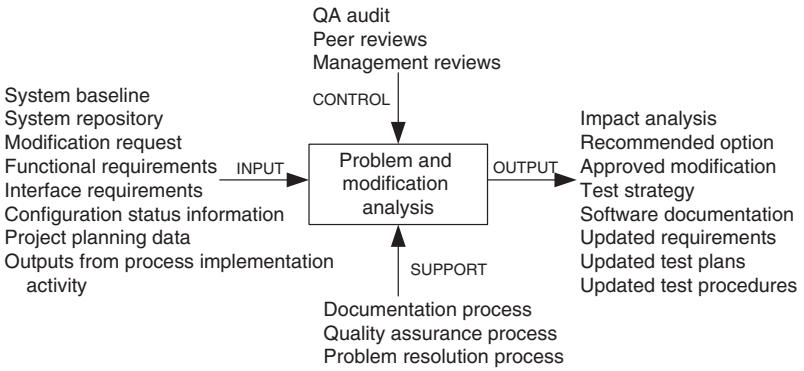

FIGURE 3.19 Problem and modification activity

classified by the maintainer as either problem reports (corrective) or enhancement (adaptive and perfective) requests. The maintenance process prioritizes and tracks these requests individually as different types of maintenance are there. A thorough discussion of MR workflow is given in Section 3.9.

Configuration management (CM): The software product and any changes made to it during its maintenance lifespan need to be controlled. Basically, change control is performed by enforcing and implementing an approved SCM process. The SCM process is implemented by developing and following a configuration management plan (CMP) and the corresponding procedures. It is discussed in detail in Section 3.8.

Problem and modification analysis. This activity is invoked after the software system transitions from development stage to maintenance stage, and it is called iteratively when the need for modification arises, as depicted in Figure 3.17. The maintainer analyzes the MR to identify its impact on the organization, the existing system, and the interfacing systems. Further, the maintainer (i) develops and documents potential solutions and (ii) obtains the approval from the upper management to implement the solutions. Figure 3.19 graphically summarizes the problem and modification analysis activity with the input, output, control, and support items. This activity comprises five tasks as discussed in the following paragraphs.

MR analysis: The maintainer analyzes the MR to determine the impact on the organization, hardware, the existing system, other interfacing systems, documentation, data structures, and humans (operators, maintainers, and users). The overall objective of impact analysis is to determine all the entities that are going to be modified and/or affected if the MR is going to be implemented. The steps of impact analysis are given in Table 3.2.

Verification: The maintainer must reproduce the problem and document the test results in the laboratory environment if the MR is corrective in order to determine the validity of the MR. For adaptive and perfective maintenance tasks, verification is not required. The maintainer designs a test strategy to verify and replicate the problem.

TABLE 3.2 Modification Request Task Steps

1. Decide whether or not the maintainer is adequately staffed to make the proposed changes.
2. Decide whether or not the maintenance program has received adequate budget.
3. Decide whether or not enough resources are available and whether the proposed change will effect some current or future projects.
4. Determine the operational issues to be considered.
5. Determine handling priority.
6. Classify the type of maintenance.
7. Determine the impact to current and future users.
8. Determine safety and security implications.
9. Identify ripple effects.
10. Determine any hardware or software constraints that may result from the proposed changes.
11. Estimate the values of the benefits of making the changes.
12. Determine the impact on existing schedules.
13. Document the risks resulting from the impact analysis.
14. Estimate the evaluation to be performed.
15. Estimate the cost of management to execute the modification.
16. Place developed artifacts under CM.

Options: The maintainer must outline two or more alternative solutions to the MR based on the analysis performed. The alternative solutions report must include the cost, effort, and schedule for implementing different solutions. The maintainer must perform the task steps shown in Table 3.3 to identify alternative solutions to the MR.

Documentation: The maintainer documents the MRs, the analysis results, and the implementation option report after the analysis is complete and the alternate solutions are identified. The maintainer may use the task steps shown in Table 3.4 to write this document.

Approval: The maintainer submits the analysis report to the appropriate authority in the organization to seek their approval for the selected change option. Upon approval, the maintainer updates the requirements if the MR is an enhancement (improvement).

TABLE 3.3 Option Task Steps

1. The MR is assigned a work priority.
2. Explore a work-around for the problem. If a work-around exists, provide it to the user.
3. Identify concrete requirements for the planned modification.
4. Calculate the magnitude and size of the planned modification.
5. Identify a variety of options to execute the planned modification.
6. Estimate the impacts of the options on the users and system hardware.
7. Analyze the risks of each option.
8. Document the outcomes of risk analysis for each of the proposed options.
9. Develop a widely acceptable plan to implement the modification.

TABLE 3.4 Documentation Task Steps

1. Ensure result analyses have been completed and documentations updated. If documentations do not exist, develop new documentation.
2. For accuracy, review the planned strategy to perform tests and review the schedule.
3. Review resource estimates for accuracy.
4. Revise the database for storing accounting status.
5. Describe a procedure to decide whether or not to approve the MR.

Modification implementation. In this activity, maintainers (i) identify the items to be modified and (ii) execute a development process to actually implement the modifications. The maintainer determines the type of documentation, software units, and version of the software that are to be changed. Though development becomes part of the modification activity, it is tailored to eliminate the activities that do not apply to maintenance effort, such as requirement elicitation and architectural design. To ensure that the modified or the newly added requirements are correctly implemented, test plans and procedures are included in the development process. In addition, it is ensured that the requirements that have not been modified are not affected by the new implementation. The inputs to this activity include all the analysis work performed in previous activities, and the output is a new software baseline. Figure 3.20 shows the modification implementation activity.

Maintenance review/acceptance. By means of this activity, it is ensured that (i) the changes made to the software are correct and (ii) changes are made to the software according to accepted standards and methodologies. The activity is augmented with the following processes: (i) a process for quality management; (ii) a process to verify the product; (iii) a process to validate the product; and (iv) a process to review the product. The maintenance plan should have documented how these supporting processes were tailored to address the characteristics of the specific software product. The inputs to this activity include the modified software and the test results. Figure 3.21 summarizes the maintenance/acceptance activity with the input,

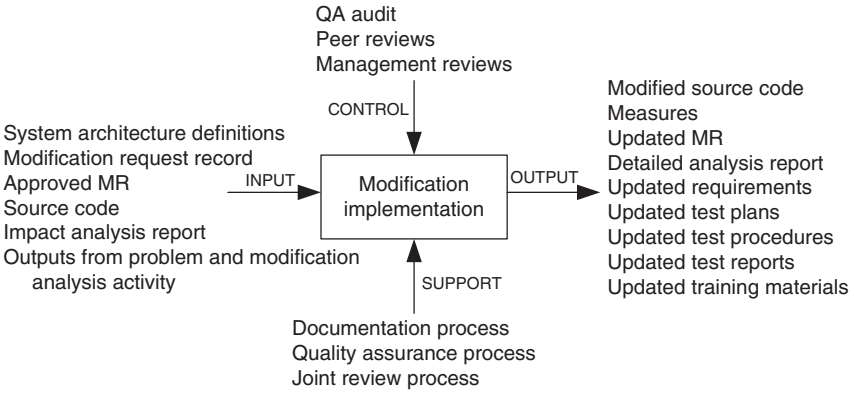

FIGURE 3.20 Modification implementation activity

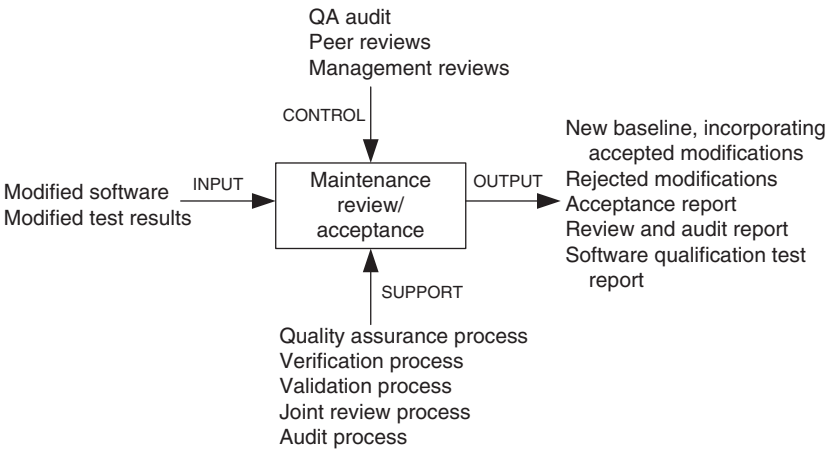

FIGURE 3.21 Maintenance review/acceptance activity

output, control, and support items. The process implementation activity consists of two major tasks: review and approval. The task steps for both review and approval are enumerated in Table 3.5.

Migration. This refers to the process of moving a software system from one technological environment to a different one that is considered to be better. Migration is effected in two broad phases: (i) identify the actions required to achieve migration and (ii) design and document the concrete steps to be executed to effect migration. Figure 3.22 summarizes the migration activity with the input, output, control,

TABLE 3.5 Review and Approval Task Steps

Review Task Steps

1. Track the MRs from requirement specification to coding.
2. Ensure that the code is testable.
3. Ensure that the code conforms to coding standards.
4. Ensure that only the required software components were changed.
5. Ensure that the new code is correctly integrated with the system.
6. Ensure that documentations are accurately updated.
7. CM personnel build software items for testing.
8. Perform testing by an independent test organization.
9. Perform system test on a fully integrated system.
10. Develop test report.

Approval Task Steps

1. Obtain quality assurance approval.
2. Verify that the process has been followed.
3. CM prepares the delivery package.
4. Conduct functional and physical configuration audit.
6. Notify operators.
7. Perform installation and training at the operator's facility.

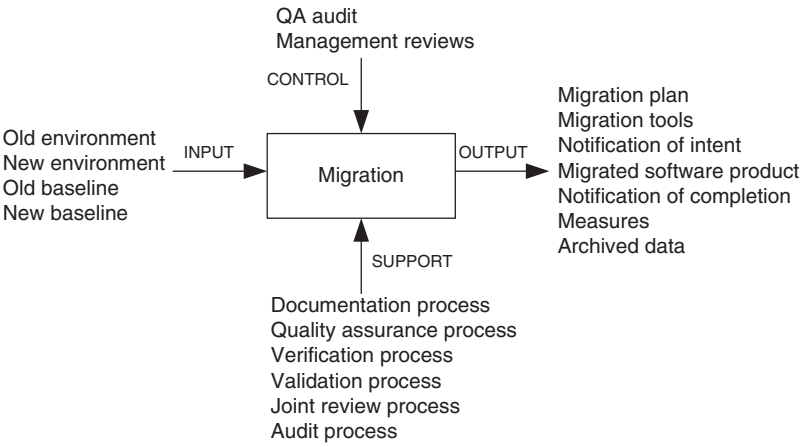

FIGURE 3.22 Migration activity

and support items. This activity comprises seven tasks discussed in the following paragraphs.

Migration standard: During the migration of a software product from an old to a new operational environment, the maintainer must ensure that any additional software products or data produced or modified adhere to standard ISO/IEC 12207 [29]. As a part of the standard tasks, the maintainer (i) identifies all the software elements or data that were changed or added and (ii) ensures that the tasks were performed according to standard ISO/IEC 12207.

Migration plan: For successful migration, a plan must be developed, documented, reviewed, and executed. The maintainer performs the task steps shown in Table 3.6 to write this document. The plan is developed in collaboration with the customers and it addresses the following:

- Requirements analysis and definition of migration
- Development of migration tools
- Conversion of software product and data
- Execution of migration
- Verification of migration
- Support backward compatibility with the old execution environment

Notification of intent: The maintainer explains to the users: (i) why support for the old environment has been discontinued; (ii) the new environment and when it will be supported; and (iii) the availability of other options, if there is any, upon the removal of the old environment.

Implement operations and training: Once a software product has been improved by modification and tested by the maintainer, it is installed in an operational environment to run concurrently with the old system. By running the old and the new system in parallel, users get an opportunity to become familiar with

TABLE 3.6 Migration Plan Task Steps

1. Analyze the requirements for migration.
2. Perform an impact analysis of migrating the software system.
3. Make a schedule to execute migration.
4. Determine all requirements for data collection to perform post-operation review.
5. Identify and record the migration effort.
6. Identify and reduce risks.
7. Identify the required tools to support migration.
8. Determine how the old environment is going to be supported.
9. Acquire and/or design new tools to support migration.
10. Partition software products and data for conversion in an incremental manner.
11. Prioritize the activities involving data conversion and software products.
12. Execute software products and data conversions.
13. Perform migration of software products and data to the new environment.
14. Operate the migrated system and the old system in parallel as much as possible.
15. Perform testing to ensure the success of migration.
16. Should there be a need, continue to provide support for the old environment.

TABLE 3.7 Operation and Training Task Steps

Parallel Operations Task Steps

1. Survey the site.
2. Install hardware equipment.
3. Install the software system.
4. Run basic tests to ensure that hardware and software have been correctly installed.
5. Run both the new and old systems in parallel, under the desired operational load.
6. Gather data from the old and the new systems.
7. Analyze the collected data.

Training Task Steps

1. Identify the requirements for migration training.
2. Schedule the requirements for migration training.
3. Review the migration training.
4. Update the plan to provide training.

the new system, so that transition from the old to the new system becomes smoother. In addition, this will create an environment for the maintainer to compare and understand the input/output relationships of the old and the new system. During this period training should also be provided to the users. The steps listed in Table 3.7 can be performed by the maintainer in this step.

Notification of completion: The maintainer notifies all the sites that the new system will become operational and that the old system can be shut down and uninstalled, after the completion of training and parallel operation of both the new and the old system for an appropriate number of hours. Essentially, the following task steps are performed by the maintainer:

1. Announce the migration.
2. Document the site-specific issues and make a plan to resolve them.
3. Archive the old system, including data and software.
4. Remove the old equipment.

Post-operation review: Following the installation and operation of a changed system, a review is performed to assess the impact of changing the system in the new environment. The review reports are sent to the competent parties for information, guidance, and further actions. The maintainer executes the following steps, as part of the task:

1. Analyze the results of running the two systems concurrently.
2. Identify potential risk areas.
3. Summarize the lessons learned.
4. Produce a report on impact analysis.

Data archival: Data associated with the old environment are made accessible to comply with the contractual requirements for data protection and audit. The maintainer performs the following steps as part of the task:

1. The old data and software are archived.
2. The old data and software are put on multiple media.
3. The media are saved in secure places.

Retirement. A software product is retired when it is viewed to have reached the end of its useful life. An economic-based analysis is performed to retire the product and it is included in the retirement plan. Sometimes the work performed by the product is no longer needed; therefore, the retired product is not replaced. In other cases, a new software product has already been developed to replace the current system. In either case, the software system must be removed from the service in an orderly manner. In addition, considerations are given to accessing data produced by the software to be retired. Figure 3.23

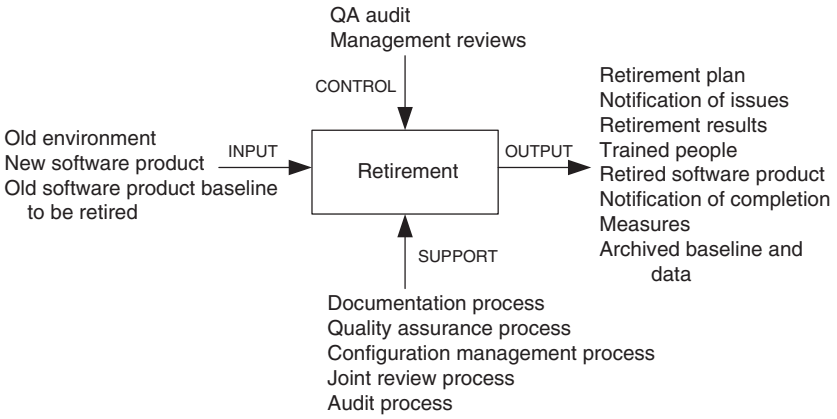

FIGURE 3.23 Retirement activity

TABLE 3.8 Retirement Plan Task Steps

1. Analyze the requirements to retire the systems.
2. Determine what impacts the retiring software will have.
3. Identify a product that will replace the software to be retired.
4. Make a schedule to retire the software.
5. Determine the need for residual support in the future.
6. Identify and describe the retirement effort.

summarizes the retirement activity with the input, output, control, and support items. All artifacts from the retirement activity are controlled with CM. This activity comprises five tasks discussed in the following paragraphs.

Retirement plan: In order to ensure a successful retirement, a retirement plan is developed, documented, reviewed, and executed. The maintainer performs the task steps shown in Table 3.8 to write this document. The plan is developed in collaboration with the customers to address the following:

- Transition to any new software system.
- Withdrawal of partial or full support after a grace period.
- Responsibility for any future contractual support.
- Archiving the software system, including all the documentation.
- Accessibility to archived data.

Notification of intent: The maintainer conveys to the users: (i) the reason for discontinuing support for the product; (ii) a note about the replacement or upgrade for the phased-out system, with an availability date; and (iii) a list of the other options available, if there is any, upon the removal of the old environment.

Implement parallel operations and training: If there is a replacement system for the software product to be retired, it is installed in an operational environment to run concurrently with the old system. By running the new and the old system in parallel, users will get an opportunity to become familiar with the new system so that transition from the old to the new system becomes smoother. In addition, this will create an environment for the maintainer to compare and understand the input/output relationships between the new system and the old system. In addition, training is provided to the users during this period.

Notification of completion: The maintainer notifies all the sites that the new system will become operational and that the old system can be shut down. The old system is generally shut down after the the new system is in operation for a certain length of time. The maintainer performs the following steps as part of the task:

1. Make an announcement about the changes.
2. Identify issues specific to individual sites and describe how those will be resolved.

3. Store the old data and software in an archive.
4. Disconnect and move out the old hardware infrastructure.

Data archival: Data associated or used with the old environment will be made accessible according to contractual requirements involving data protection and audit. The maintainer executes the following steps as part of this task:

1. Archive the old data and software.
2. Make multiple copies of the old data and software.
3. Keep the media in safe places.

3.8 SOFTWARE CONFIGURATION MANAGEMENT

Large, complex systems undergo many more changes than relatively small systems, and management of changes in large systems is nontrivial. Therefore, the concept of CM was developed to manage changes in large systems.

The goal of CM is to manage and control the numerous corrections, extensions, and adaptations that are applied to a system over its lifetime. It handles the control of all product items and changes to those items. On the other hand, SCM is applied to software products. In this case the product items include document, executable software, source code, hardware, and disks. SCM has been defined by Bersoff, Henderson, and Siegel [30] as *the discipline of identifying the configuration of a system at discrete points in time for the purpose of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle.* SCM accrues two kinds of benefits to an organization as follows:

- SCM ensures that development processes are traceable and systematic so that all changes are precisely managed. Consequently, the product is always in a well-defined state [31].
- SCM enhances the quality of the delivered system and the productivity of the maintainers.

CM is an essential part of software development and maintenance environment. It ensures that the released software is not contaminated by uncontrolled or unapproved changes. The objectives of SCM are to:

- Uniquely identify every version of every software at various points in time.
- Retain past versions of documentations and software.
- Provide a trail of audit for all modifications performed.
- Throughout the software life cycle, maintain the traceability and integrity of the system changes.

Projects benefit from effective SCM as follows [32]:

1. Confusion is reduced and order is established.
2. To maintain product integrity, the necessary activities are organized.
3. Correct product configurations are ensured.
4. Quality is ensured and better quality software consumes less maintenance efforts.
5. Productivity is improved, because analysts and programmers know exactly where to find any piece of the software.
6. Liability is reduced by documenting the trail of actions.
7. Life cycle cost is reduced.
8. Conformance with requirements is enabled.
9. A reliable working environment is provided.
10. Compliance with standards is enhanced.
11. Accounting of status is enhanced.

3.8.1 Brief History

A need for CM was originally felt in the aerospace industry in the 1950s with the primary purpose of guaranteeing reproducibility of aircrafts and managing engineering changes (ECs). In the 1970s, large-scale computer software began to pose many of those same change management problems. It became apparent that software maintenance engineers could borrow CM techniques from the aerospace industry to manage software modifications. In the beginning, punch cards with different colors were used to indicate changes. In the late 1960s, to indicate changes to the UNIVAC-1100 EXEC-8 operating system, maintenance personnel used "corrective cards." At that time, development of operating systems benefited from SCM. In the 1970s and the 1980s, SCM emerged as a distinct discipline. With the advancements in user-friendly software development environments, namely, Unix, specialized computer software tools were built. For example, the Unix-based software tool *Make* [33] accepts descriptions of system configurations and can automatically construct the system from its descriptions. The source code control system (SCCS) [34] and the revision control system (RCS) [35] tools permit the maintainer to keep track of all the *textual* alterations made to a file.

Gradually, software products became candidates for configuration control. This resulted in a need to manage user workspaces, which was duly supported by newer SCM systems. In other words, SCM functionalities continued to evolve. Instead of storing the entire versions of software products, in the 1980s, delta algorithms based on text matching were developed to enable SCM tools to store just the differences among versions. In the 1990s, developers felt an increasing need to manage nontextual objects. Consequently, novel algorithms were developed for efficient storage and retrieval of nontextual objects. By the year 2000, due to disk storage becoming inexpensive, CPUs becoming fast, and nontextual objects becoming common, the storage of deltas became unimportant and many new tools simply used compression

such as zip. These days SCM systems support the management of evolution of a broad range of software systems that are being modified by a large number of maintenance personnel working in different countries and utilizing a variety of machines.

3.8.2 SCM Spectrum of Functionality

These days a broad range of high-level functionalities are supported by SCM systems. Estublier et al. [31] classified the functionalities into three broad areas: product, tool, and process. Next, each area is decomposed into a number of technical dimensions as shown in Figure 3.24. Those technical dimensions are briefly explained in the following.

Identification. The items whose configurations need to be managed are identified in this function. The identified items include specification, design, documents, data, drawings, source code, executable code, test plan, test script, hardware components, and components of the software development environment, namely, compilers, debuggers, and emulators. Project plan and customer requirements should also be included. To accurately identify products, including their configuration and version levels, a schema of names and numbers is designed. Finally, for all configuration items and systems, a baseline configuration is established. If there is a need to make changes to the baseline, it is done so with the concurrence of the configuration control organization.

Version control. To avoid confusion during the process of artifact evolution, a new identifier is assigned to the artifact every time the artifact is modified. It is important to note that assigning a new identifier for every modification of the same artifact may hide important relations among the uniquely identified artifacts. As an example, one may be interested in recording a fact that a given artifact fixes a subset of defects

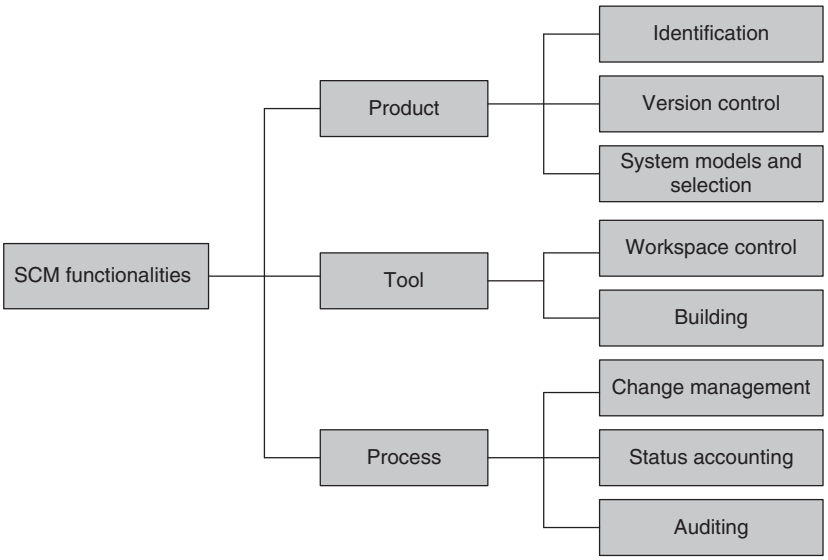

FIGURE 3.24 Technical dimensions of SCM systems

found in an earlier release. The aforementioned kind of relation can be recorded by means of the version control (VC) functionality of SCM by: (i) interpreting software artifacts as configuration items and (ii) identifying the relations, if there is any, among the configuration items.

The basic VC idea is to have two separate files: master copies and working copies. The former is stored in a centralized repository. Software developers check out working copies from the repository, modify the working copies, and, finally, check in the working copies into the repository. Checking in a file means committing to the changes made to the working copies. The VC system creates a new version in the repository every time a file is committed. As time passes, all versions of the file are stored in the repository. Storing multiple copies of a file does not excessively waste space. Therefore, storing many copies of a file does not excessively waste storage space. Conflicts can arise if many software developers want to use the same version of a file. However, conflicts can be resolved by means of two techniques: lock-modify-unlock and copy-modify-merge [36]. The former model requires developers to obtain a lock on the file they want to modify. While a developer is holding the lock on a file, no other developer can modify the file. However, it may be noted that a locked file can be checked out for viewing and compilation. When there is no need to further keep a file locked, the developer commits the modifications and the lock is released. On the other hand, developers are at liberty to change their working copies in the copy-modify-merge model. If different developers make conflicting changes, the VC system flags the conflicts so that the developers can resolve them.

In addition, VC must support parallel development by allowing branching of versions. For example, consider the scenario: (i) an organization is currently developing the next version of their already released application; and (ii) a report about a major defect is received from the end users. Now the development group has the option to retrieve the released version and create a branch, as illustrated in Figure 3.25, to fix the defect. The figure illustrates how a file evolves with two branches, where the main path is called trunk. As shown in the figure, branch changes are incorporated by merging with the trunk.

System models and selection. Files are discrete entities that contain descriptions of well-defined items of a project, namely, requirement specification, test cases, design, code, test results, and defect reports. However, it is neither efficient nor effective to manage a project file-by-file. Consequently, a need to support aggregate artifacts arises so that maintenancex personnel can enforce consistency in large projects by

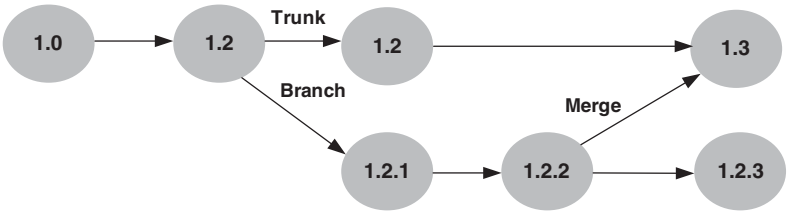

FIGURE 3.25 An evolution of a file with two branches

means of relationships among artifacts and attributes. Relationships among artifacts and attributes are captured by developing models which support the idea of software configurations. Intuitively, a configuration means an aggregate of versionable items. The general idea of configuration raises a need for enabling users to have selective access to parts and versions of such aggregated artifacts. By default, SCCS and RCS keep in the workspace the most recent version of the principal variant. Next, all artifacts that are exceptions to the default placement rule can be explicitly fetched by the user.

Workspace control. Workspaces are implemented by SCM systems to give users an isolated place. In their own workspaces, users perform the usual tasks of editing and managing their artifacts. Such an environment that enables the maintainer to make and test the changes in an isolated manner is called workspace. In an SCM system, software versions are stored in a repository that cannot be directly modified. Rather, when a need to modify some files arises, the files are copied into a workspace. One can realize a workspace in two ways: (i) it can be as simple as the home directory of the programmer who wants to modify the files and (ii) it can be a complex mechanism such as an integrated development environment and a database. In general, three basic functions are performed in a workspace:

- Sandbox: Checked out files are put in a workspace to be freely edited. In addition, it is not necessary to lock the original files in the repository.
- Building: An SCM system generally stores the differences between successive versions to save space. Therefore, the workspace expands the deltas into full-fledged source files. In addition, the workspace stores the derived binaries.
- Isolation: Every developer maintains at least one workspace. Therefore, the developer makes modifications to the source code, compiles the files, performs tests, and debugs code without impacting the works of other developers.

The aforementioned features are generally available in modern software development environments. SCM systems provide a centralized facility to manage these features.

Building. Efficiency is a key requirement of SCM systems so that developers can quickly build an executable file from the versioned source files. A second requirement of SCM systems is that it must enable the building of old versions of the system for recovery, testing, maintenance, or additional release purpose. Third, most SCM systems support building of software. The build process and their products are assessed for quality assurance. Outputs of the build process become quality assurance records, and the records may be needed for future reference. The *make* [33] application on the Unix operating system, originally developed by researchers at Bell Laboratories, is a classical example of a build process. The tool continues to remain popular for system building. For instance, commercial SCM systems such as *ClearCase* [37] continue to rely on variants of *make*.

Change management. SCM systems must: (i) enable users to understand the impact of modifications; (ii) enable users to identify the products to which a specific

modification applies; and (iii) provide maintenance personnel with tools for change management so that all activities from specifying requirements to coding can be traced. In the beginning, CRs were managed in paper form. However, these days CRs are saved in the SCM repository and are linked with the actual modifications, in addition to being automated. This topic is further discussed in Section 3.9.

Status accounting. To be able to quantify the properties of the software being developed and the process being used, it is necessary to gather statistics—and it can be done at the SCM level. The primary purpose of status accounting is to: (i) keep formal records of already existing configurations and (ii) produce periodic reports about the status of the configurations. These records: (i) describe the product correctly; (ii) are used to verify the configuration of the system for testing and delivery; and (iii) maintain a history of CRs, including both the approved ones and the rejected ones. A history of CR includes the answers to the following questions:

- Why are changes made?
- When are the changes made?
- Who makes the changes?
- What changes are made?

Status accounting is useful in communicating important details of the project and configuration items to the stakeholders of the project. For example, maintenance personnel can view what files or fixes were part of what baseline systems. Another example is that project managers can trace completion of problem reports. Status accounting reduces the need to produce extensive reports, which include item *delta* report, transaction log, and modification log. Other common reports include resource usage, change in process, change in deviations, and status of all configuration items [38]. Examples of status accounting include the number of CR per software configuration item and the average time needed to implement a CR.

Auditing. SCM systems need to provide the following features: (i) roll back to earlier stable points and (ii) identify which modifications were performed, why those modifications were performed, and who performed those modifications. In other words, ideally, an SCM system behaves as a searchable archive of all things that happened in the past. By means of auditing, the organization maintains the integrity of the baselines and release configurations for all products. Two kinds of audits are performed before a software is released: audit for functional configuration and audit for physical configuration. The former determines whether or not the software satisfies the user requirement specification and the system requirement specification. On the other hand, the latter verifies if the reference and design documents accurately represent the software. Overall, a configuration audit tries to find answers to the following:

- To what extent are the requirements satisfied by the modified system?
- Does the software release under consideration reflect the MRs?

The activities to perform a configuration audit are as follows:

- Procedures and an audit schedule are defined.
- The personnel to perform the audits are identified.
- Established baselines are audited.
- Audit reports are generated.

3.8.3 SCM Process

There is a large gap between mere understanding of the capabilities of SCM and successfully applying SCM in practice. As it is the case with large software projects, planning is critical to the successful application of SCM. By means of a plan, a configuration baseline is established. Baselining is the process by which a given set of configurable items formally become publicly available at a standard location, such as in a repository, to the people who are authorized to use it. Following the identification of the initial configuration, a configuration control process, as illustrated in Figure 3.26 [32], is invoked. Figure 3.26 shows the three major SCM implementation activities: planning, baseline development, and configuration control.

Planning. Planning is begun with two activities: (i) defining the SCM process and (ii) establishing procedures to control and document changes. A key step during planning is the identification of the stakeholders. All those who influence a system's behavior and all those who are impacted by the system are stakeholders of the system [39]. The stakeholders in a CM are the maintainers, development engineers, sustaining test engineers, quality assurance auditors, users, and the management. The

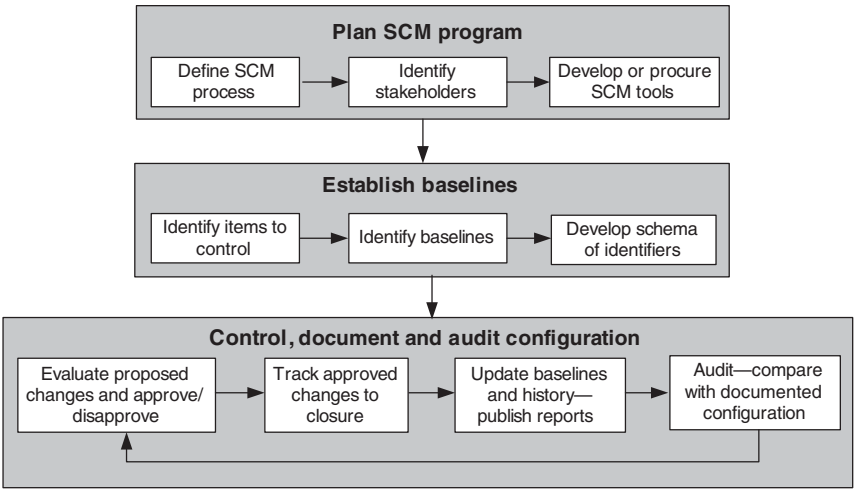

FIGURE 3.26 A process for implementing SCM

stakeholders are also known as configuration control board (CCB) members. Not all changes are reviewed by the board. Rather, small groups review and approve most of the changes. Therefore, those groups need to be identified in the planning phase. Various SCM tools are used to maintain configuration history and facilitate the SCM process flow. Examples of such tools are concurrent version system (CVS) and Clearcase [37].

Establishing baselines. Once an SCM program plan is in place, the next step in implementing effective SCM is to identify the items that are the subject of configuration control. Some examples of those items are code, data, and documents. With the configuration items identified, a software baseline library is established to make the set of configurable items publicly available. The library, called repository, is the heart of the SCM system. The repository is the central place that contains all configurations that have been made public. In other words, the repository has information about all the baselined items. The process of baseline (or re-baseline for a change) involves the following activities:

1. Create a snapshot of the current version of the product and its configuration items and allocate a configuration identifier to the entire configuration.
2. Allocate version numbers to the configuration items and check in the configuration.
3. Store the approved authority information as part of meta data in the repository.
4. Broadcast all the above information to the stakeholders.
5. To accurately identify the configuration version, design a schema of words, numbers, or letters for common types of configuration items. In addition, project requirements may dictate specific nomenclature.

Controlling, documenting, and auditing. After establishing a baseline, it is important to: (i) keep the actual and the documented configuration identical and (ii) ensure that the baseline complies with a project's configuration described in the requirements document. The aforementioned requirements of a baseline are realized by means of a four-step iterative process illustrated in Figure 3.26. The stakeholders specified in the SCM plan review and evaluate all changes to the configuration. After their evaluations, both approvals and disapprovals are documented. Approved changes are tracked until they are verified—and this is discussed in Section 3.9. Next, the appropriate baseline is revised in conjunction with all relevant documents, and reports are generated. At regular intervals, records and products are audited to verify that:

- there is acceptable matching between the documented configuration and the actual configuration;
- the configuration conforms with the requirements of the project; and
- documentations of all change activities are complete and up-to-date.

The three steps in the cycle, namely, controlling, documenting, and auditing, are repeatedly executed throughout the lifetime of the project.

3.9 CR WORKFLOW

A CR, also called an MR, is a vehicle for recording information about a system defect, requested enhancement, or quality improvement. In other words, defect reports or enhancement requests are documented as a CR. CRs are placed under the control of a change management system. Change management systems control changes by an automated system in the form of workflow. The basic objective of change management is to uniquely identify, describe, and track the status of each requested change. It is a methodology for controlling changes to evolving systems. The objectives of change management are as follows [1]:

- Provide a common method for communication among stakeholders.
- Uniquely identify and track the status of each CR. This feature simplifies progress reporting and provides better control over changes.
- Maintain a database about all changes to the system. This information can be used for monitoring and measuring metrics.

A CR describes the desires and needs of users which the system is expected to implement. While describing a CR, two factors need to be taken into account:

- Correctness of CRs: CRs need to be unambiguously described so that it is easy to review them for their correctness. The "form" of a CR is key to effective interactions between the software development organization and the users. The "form" should document essential information about changes to software, hardware, and documentation.
- Clear communication of CRs to the stakeholders: CRs need to be clearly communicated to the stakeholders, including the maintainers, so that those CRs are not interpreted in a different way. The people who might be collecting CRs may not be part of the maintenance group. For example, the marketing people may be collecting CRs. Therefore, there may not be direct communication between the teams actually carrying out the changes to the system and the end users.

It becomes counterproductive for different teams to interpret CRs differently. The results of interpreting a CR in different ways are as follows:

- The team carrying out actual changes to the system and the team performing tests may develop contradicting views about the new system's quality.
- The changed system may not meet the needs and desires of the end users.

CRs need to be represented in an unambiguous manner and made available in a centralized repository. Wide availability of CRs to all the stakeholders is likely to reveal differences in interpretations by different groups.

Next, a formal model is described to represent CRs for analysis and review. The life cycle of a CR has been illustrated in Figure 3.27, by means of a state-transition diagram. Each state represents a distinct stage in the life-cycle of a CR.

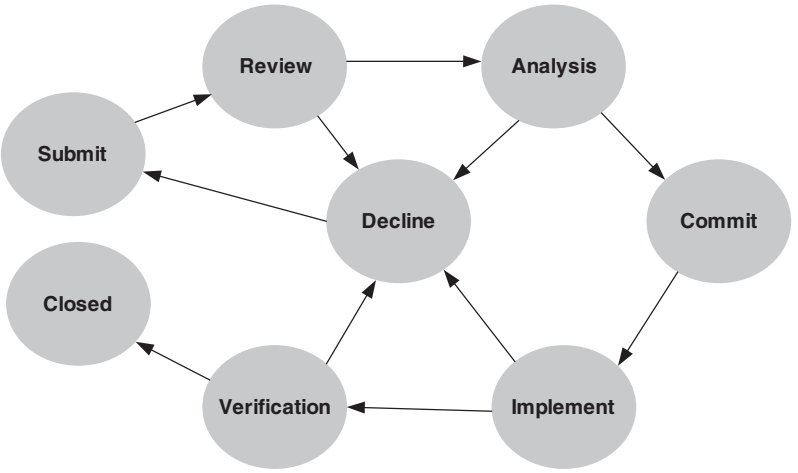

FIGURE 3.27 State transition diagram of a CR

The model shows the evolution of a CR via the following major states: *Submit, Review, Analysis, Commit, Implement, Verification,* and *Closed*. Specific actions are associated with each state, and the state of a CR is updated upon the completion of those actions. For several reasons, the status of a CR is changed to the *Decline* state from *Review, Analysis, Implement,* and *Verification*. For instance, the marketing team may conclude that realization of a CR may not fetch more business. The motivation for describing CRs by means of state diagrams is to enable their easy tracking. For ease of implementation and management of CRs, a general schema, as shown in Table 3.9, can be used. Once such a schema is implemented with a back-end database and a front-end graphical user interface, CRs can be stored in a database. Later, for the purposes of tracking and reporting the status of the CRs, queries are generated.

Submit state. This is the initial state of a newly submitted CR. Usually, end users, customers, and marketing managers are the prime sources of CRs. When a new CR is filed, the following fields, described in Table 3.9, are instantiated:

change_request_id
priority
description
maintenance_type
component
note
product
customer

Based on the priority level of a CR, it is moved from *Submit* to *Review*. Usually, a marketing manager assumes the responsibility of this initial handling of a CR, and he becomes the ”owner” of the CR.

TABLE 3.9 Change Request Schema Field Summary

Field Name	Description
change_request_id	A unique identifier of the CR
title	A concise summary of the CR
description	A short description of the CR
maintenance_type	Classification of the maintenance type in terms of a member of {Corrective, Adaptive, Perfective, Preventive}
product	Product name
component	Component where the change is needed, or where the problem occurred
state	Present state of the CR in terms of a member of {Submit, Review, Analysis, Commit, Implementation, Verification, Closed, Declined}
customer	Name of the customer making the CR
problem_origin	The origin of the problem
impacts	Components that are affected by a change and its ripple effect
resolution	Documentation of what was changed, how, and why
note	Additional information provided by the submitter for subsequent decision making
software_release	The version number of the product release in which the CR is likely to be effective
committed_release	The version number of the product release in which the CR will be effective
priority	Priority of CR, which is an element of a set, namely, {normal, high}
severity	Severity of CR, which is an element of a set, namely, {normal, critical}
marketing_justification	The business justification for the CR to exist
time_to_implement	The time, in person-week, required to effect the change
eng_assigned	The engineering personnel assigned to analyze the CR
functional_spec_title	Title of the specification for functional requirements
functional_spec_name	Name of the file describing the functional requirements
functional_spec_version	This is the most recent version number of the specification of functional requirements
decline_note	The reason for declining the CR
ec_number	The identifier of the EC document
attachment	Attachment to further describe the CR (if any)
tc_id	Identifiers of test cases used in effecting the CR
tc_results	The result of testing: {Untested, Passed, Failed Blocked, Invalid}
verification_method	Record the methods of verification of the CR: analysis, testing, inspection, and/or demonstration
verification_status	The verification state of the CR: passed, failed, or incomplete
compliance	The level of compliance:{Non-compliance, Partial Compliance, or Compliance}

(*continued*)

TABLE 3.9 (*Continued*)

Field Name	Description
testing_note	Reports from the test personnel, possibly describing the demonstration given to the customers, analysis performed on the change, or inspection of the code performed by test personnel
defect_id	Defect identifier If "Failed" is assigned to the *tc_results* field, the defect identifier is associated with the failed test to indicate the defect causing the failure. The defect identifier is obtained from test database.

Review state. Generally, a manager for marketing handles the CR in the Review state by coordinating the following activities:

- It is possible that the newly generated CR is a duplicate of an existing CR. If it is found to be a duplicate of an existing CR, the request is moved to the *Decline* state with a short explanation and a link to the original CR. Should there be any ambiguity in the description of the CR, the submitter is asked to provide more details, which are recorded in the *note* and the *description* fields.
- Accept the assigned priority level of the CR or modify it.
- Re-evaluate the maintenance_type of the CR initially estimated by the submitter, and accept or modify it.
- Determine the level of severity of the CR: normal and critical. If it is critical, then the upper management may want to complete the review immediately. Note that a severity level and a priority level are independently assigned.
- To reflect the CR, determine a software release.
- Give a marketing rationale for the CR.
- For further actions, the CR is moved to the *Analysis* state.

In summary, the following fields are updated in the *Review* state:

priority
severity
maintenance_type
decline_note
software_release
marketing_justification
description and
note

Analysis state. In this stage, impact analysis is conducted to understand the CR and to estimate the time required to implement it. In addition, a high-level functional specification for the CR is prepared. If it is decided that it is not possible or desirable to implement the CR, then *Decline* becomes the next state of the CR. Otherwise, the CR is moved to the *Commit* state. In the *Commit* state, the program manager controls the CR by becoming its owner. While in the *Analysis* state, the owner, who is typically the director of software engineering, updates the following fields:

component
problem_origin
impacts
time_to_implement
attachment
functional_spec_title
functional_spec_name
functional_spec_version
eng_assigned

Commit state. The CR continues to stay in the *Commit* state before it is committed to a specific release of the product. In this state, the program manager is the owner of the CR. All the CRs that are desired to be in a specific software release are reviewed. Some CRs may be re-assigned to a later release after consultations with customers, the marketing division, and the director of software engineering. After committing a CR to a particular release, the CR is moved to the *Implement* state and all the functional specifications are frozen for development and test design purposes. In the *Commit* state: (i) the engineering team begins modifying the software component documentation, namely, data and control flow diagrams and schematics; (ii) test personnel review the CR and the associated functional specification to ensure that the CR is testable; (iii) test personnel write new test cases for the CR; and (iv) test personnel select regression tests. *Committed_release* is the only field updated in the *Commit* state.

Implement state. The *Implement* stage is controlled by the director of software engineering. A number of different scenarios can occur in this stage as follows:

- The CR can be declined if its implementation is infeasible.
- If the CR is infeasible in its current form, the director of software engineering may assign an EC number and provide an explanation, and the EC document is linked with the CR definition. Table 3.10 shows how to organize an EC document.
- If the CR or its modified version is doable, the software engineering group writes code and performs unit tests. The CR is moved from *Implement* to *Verification* after the product is available for system-level testing.

TABLE 3.10 Engineering Change Document Information

EC number	A unique number.
Requirement(s) affected	Identifiers of CRs and their titles.
Description of problem/issue	Brief description of the issue.
Description of change required	Description of changes needed to the original CR description.
Secondary technical impact	Description of the impact the EC will have on the system.
Customer impacts	Description of the impact the EC will have on the end customer.
Change recommended by	Name of the engineer(s).
Change approved by	Name of the approver(s).

Source: From Reference 24. © 2008 John Wiley & Sons.

In the *Implement* state the following fields are updated:

decline_note
ec_number
attachment
resolution

Verification state. In the *Verification* state, activities are largely controlled by the sustaining test manager. To assign a test verdict, verification can be performed by one or more methods: demonstration, analysis, inspection, and testing. If verification is performed by testing, then the software is executed with a set of tests. Inspection means reviewing the code to detect defects. Analysis is performed by means of statistical and/or mathematical tools. Demonstration implies showing the system in a live operation. A status of verification is provided in terms of the degree of compliance of the modified system to the CR: noncompliance, partial compliance, or full compliance. If the testing method is not used, then a note explains the details of the demonstration, the inspection, and/or the analysis performed.

Shortfalls in the realization of the CR, in the form of incomplete and even partly accurate implementation, are specified in an EC document. It is very difficult to correct any deviations or errors discovered at this stage. Therefore, a pragmatic approach to dealing with the deficiency is to produce an EC document, after negotiating with the customer, to revise the CR, and possibly generate a new CR for future considerations. As an extreme decision, the sustaining test manager may decline to accept the modifications made to the code an EC number and an explanation, followed by a state change to *Decline*. On the other hand, after ensuring that the implementation passed the required tests, the sustaining test manager moves the CR to the *Closed* phase. In the *Verification* state, the following fields are instantiated:

decline_note
ec_number
attachment

verification_method
verification_status
compliance
tc_id
tc_results
defect_id
testing_note

Closed state. After successfully verifying that the CR has been incorporated into the software, the CR is moved from *Verification* to the *Closed* state. This is done by the owner of the CR in the *Verification* state who is, in general, the sustaining test manager.

Decline state. The *Decline* state is controlled by the marketing department. Due to one or more of the following causes, a CR happens to be in this state:

- Because of insufficient business impact of the CR, the marketing department decides to reject the CR.
- It is technically infeasible to implement the CR.
- The sustaining test manager concludes that changes made to the software to effect the CR could not be satisfactorily verified. An explanation is provided in the form of an EC number.

The CR may be moved to *Submit* by the marketing group after negotiating with the customer. Negotiations with the customer may lead to a reduction in the scope of the CR. The EC information is used as a basis for negotiation with the customer.

3.10 SUMMARY

This chapter began with three well-known reuse-oriented paradigms: quick fix, iterative enhancement, and full reuse. All the three models assume that a set of documents completely and accurately describe the existing system. The first model makes the necessary changes to the code first, followed by changes to the relevant documentations. The second model modifies the top-level documents impacted by the modifications and then propagates those changes down to the code level. The third model builds a new system from components of the old system and other components available in the repository.

Next, we studied a simple staged model for CSS development, which comprises five major stages: Initial development, Evolution, Servicing, Phaseout, and Closedown. In this view software life cycle, maintenance is actually a series of distinct stages, each with different activities. The software evolution process is the backbone of the model. It continues in an iterative fashion and eventually produces the next version of the software. We discussed its applicability to FLOSS systems model. The

model benefits developers to characterize FLOSS systems in terms of stages, and identify the current stage of a system and the new stage to which it is more likely to move. Three major differences between CSS and FLOSS systems were identified and discussed.

Next, we described an evolutionary model known as change mini-cycle, which consists of five major phases: CR, analyze and plan change, implement change, verify and validate, and document change. In this model, several interesting activities were identified, such as program comprehension, impact analysis, refactoring, and change propagation, which continue to be the subjects of intense research.

With the three evolution models in place, we examined two standards: IEEE/EIA 1219 and ISO/IEC 14764. Both the standards describe the process for managing and executing software maintenance activities. Next, we discussed the management of system evolution by focusing on SCM. Next, a state-transition model was given to monitor individual CRs as those move through the software organization. Certain actions are completed in each state of the model. Finally, a sample schema to manage CRs was presented in detail.

LITERATURE REVIEW

In the classical Waterfall model for software development proposed by Royce in circa 1970 [40], the final phase is Operation and Maintenance. This model considers software maintenance as another task of software development. On the other hand, J. R. McKee, in his article "Maintenance as a function of design," *AFIP National Conference Proceeding*, 53, 1984, pp. 187–193, suggests software maintenance to be 2nd, 3rd, … , *n*th round of development. In this regard it is worth quoting Norman Schneidewind from his article "The state of software maintenance," *IEEE Transactions on Software Engineering*, 13(3), 1987, pp. 303–309: "The traditional view of the software life cycle has done a disservice to maintenance by depicting it solely as a single step at the end of the cycle" (p. 304). Victor Basili [2] argues that software maintenance is continued development, using the same knowledge, methods, and tools used for software development. Based on this view he developed the reuse-oriented software development process. A more complicated idea of using SDLC is proposed by Barry Boehm ("Software engineering," *IEEE Transaction on Computers*, December 1976, pp. 1226–1241) based on his spiral model [41]. Ned Chapin, in his article "Software maintenance life cycle," *Proceedings of Conference on Software Maintenance*, Phoenix, Arizona, IEEE Computer Society Press, Los Alamitos, CA. 1988, pp. 6-13, argued that the SDLC model is not compatible with a software maintenance model. The use of SDLC will generate an inappropriate expectation set of metric requirements, such as effort needed, selection of tools, management support, and complexity of relevant task. Therefore, he prefers software maintenance to have its own SMLC model.

A number of proposals based on the SMLC model have been published with some variations among them. Three common features of SMLC models found in literature

are: (i) understanding the code; (ii) modifying the code; and (iii) revalidating the software. In the following we list up those advocating the SMLC model:

G. Parikh. 1982. 'The world of software maintenance. In: *Techniques of Program and System Maintenance* (Ed. G. Parikh), pp. 9–13. Little, Brown and Company, Boston, MA.

J. Martin and C. L. McClure. 1983. *Software Maintenance: The Problem and Its Solution*. Prentice-Hall, Englewood Cliffs, NJ.

S. Chen, K. G. Heisler, W. T. Tsai, X. Chen, and E. Leung. 1990. A model for assembly program maintenance. *Journal of Software Maintenance: Research and Practice*, March, 3–32.

D. R. Harjani and J. P. Queille. 1992. *A Process Model for the Maintenance of Large Space Systems Software*. Proceedings of Conference on Software Maintenance, November 1992, Orlando, FL. IEEE Computer Society Press, Los Alamitos, CA. pp. 127–136.

W. K. Sharpley. 1977. *Software Maintenance Planning for Embedded Computer Systems*. Proceedings of the IEEE COMPSAC, November 1977, IEEE Computer Society Press, Los Alamitos, CA, pp. 520–526.

S. S. Yau, R. A. Nicholi, J. Tsai, and S. Liu. 1988. An integrated life-cycle model for software maintenance. *IEEE Transaction on Software Engineering*, August, 1128–1144.

S. S. Yau and I. S. Collofello. 1980. Some stability measures for software maintenance. *IEEE Transaction on Software Engineering*, November, 545–552.

L. J. Arthur. 1988. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, New York, NY, 1988.

The model proposed by Sharpley focused on corrective maintenance activities through problem verification, problem diagnosis, reprogramming, and baseline revalidation. On the other hand, Arthur's model is for corrective, adaptive, and perfective maintenance activities and consisted of several phases: (i) change management; (ii) impact analysis; (iii) system release planning; (iv) design change; (v) coding; (vi) testing; and (vii) system release.

Neal Febbraro and V. Rajlich presented an agile model of incremental change and development process that consists of repeated incremental change (*The Role of Incremental Change in Agile Software Processes*. Proceedings Agile, August 2007, Washington, D.C. IEEE Computer Science Press, Los Alamitos. pp. 92–103. At Iona Technologies, XP has been used successfully for maintenance of a Corba-based middleware product called Orbix, as reported in the article ("Using extreme programming in a maintenance environment," *IEEE Software*, November/December, 2001, pp. 42–50) by C. Poole and J. W. Huisman. According to Kent Beck, one of the proponents of XP, "Maintenance is really the normal state of an XP project" (*Extreme Programming Explained*. Addison-Wesley, Reading, MA, 1999).

By following a development process an organization develops new software products, whereas a maintenance process is seen as providing subsequent services. The line between services and products is not a crisp one. On the one hand, adaptive maintenance is viewed as a hybrid of services and product. On the other hand, corrective maintenance is a product-intensive service, and software operation can be

considered a pure service. Basically, an intangible set of activities and/or benefits are bundled and sold as a service by an organization to its customers. Therefore, to deliver high-quality results from maintenance tasks, two quality dimensions must be considered: functional quality and technical quality. Maintenance services are improved by organizations by improving the: (i) technical quality of their work and (ii) functional quality of software maintenance. In this context it is valuable to read and practice the process proposed by Niessink and van Vliet in their article "Software maintenance from a service perspective," published in the *Journal of Software Maintenance: Research and Practice*, March/April 2000, pp. 103–120. Their process can be summarized in the form of four items:

1. Customer expectations are translated into maintenance service agreements.
2. Maintenance activities are planned and implemented by using the service agreements as a basis.
3. Planning and procedures guide the maintenance activities.
4. Manage the communication concerning the delivered services.

In addition, it is highly recommended for information technology (IT) professionals to study the following two standards:

1. The IT Service Capability Maturity Model, Version 1.0RC1, January 28, 2005, http://www.itservicecmm.org
2. The IT Infrastructure Library (ITIL)—An Introduction, Central Computer and Telecommunication Agency (CCTA), HMSO books, Norwich, England, 1993.

The British government developed ITIL through CCTA and it was maintained by the Netherlands IT Examinations Institute (EXIN). ITIL aims at establishing best standards and practices for IT service delivery. To explain the "best practices" in the delivery of IT service, nine sets of infrastructure library booklets have been developed. The nine sets of booklets cover two broad topics: (i) provision of IT services and management of IT infrastructure and (ii) environment. The former has been addressed in the first six sets of booklets, whereas the latter in the remaining three. Cabling, building, and service facilities are part of the environment. The IT service capability maturity model has much similarities with the Software CMM. IT services are delivered by installing, maintaining, managing, or operating the IT needs of a customer. In other words, software maintenance is one element of a whole gamut of the deliverable IT services.

The software maintenance maturity model **SM**^{mm} was designed by Alain April and Alain Abran (*Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley-IEEE Computer Society Press, April 2008) as a customer-focused reference model for either (i) auditing the software maintenance capability of a software maintenance service supplier or outsourcer or (ii) improving internal software maintenance organizations. It includes 4 process domains, 18 key processes

areas (KPAs), 74 roadmaps, and 443 practices. In addition, the reader is recommended to study the corrective maintenance maturity model article by Mira Kajko-Mattsson, "*Motivating the Corrective Maintenance Maturity Model (CM^3)*. Seventh IEEE International Conference on Engineering of Complex Computer Systems, 2001, Skovde, Sweden. IEEE Computer Society Press, Piscataway, NJ. pp. 112–117."

Those who are interested in a much detailed treatment of SCM peruse the article by Estublier, Leblang, Hoek, Conradi, Clemm, Tichy, and Wiborg-Weber ("Impact of software engineering research on the practice of software configuration management," *ACM Transactions on Software Engineering and Methodology*, 14(4), October 2005, pp. 383–430). They discussed the evolution of SCM technology, with emphasis on the impact of industrial and university research.

REFERENCES

[1] L. J. Arthur. 1988. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, New York, NY.

[2] V. R. Basili. 1990. Viewing maintenance as reuse-oriented software development. *IEEE Software*, January, 19–25.

[3] C. Larman and V. R. Basili. 2003. Iterative and incremental development: a brief history. *IEEE Computer*, June, 47–55.

[4] T. Gilb. 1988. *Principles of Software Engineering Management*. Addison-Wesley, Reading, MA.

[5] G. Visaggio. 1999. Assessing the maintenance process through replicated controlled experiments. *The Journal of Systems and Software*, January, 187–197.

[6] V. T. Rajlich and K. H. Bennett. 2000. A staged model for the software life cycle. *IEEE Computer*, July, 2–8.

[7] E. Yourdon. 1995. When good enough software is best. *IEEE Software*, May, 79–81.

[8] N. Chapin, J. E. 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*, January/February, 3–30.

[9] A. Capiluppi, J. M. G. Barahona, I. Herraiz, and G. Robles. 2007. *Adapting the Staged Model for Software Evolution to Free/Libre/Open Source Software*. IWPSE, September 2007, Dubrovnik, Croatia. ACM, New York. pp. 79–82.

[10] S. S. Yau, J. S. Collofello, and T. MacGregor. 1978. *Ripple Effect Analysis of Software Maintenance*. COMPSAC, November 1978, Chicago, Illinois. IEEE Computer Society Press, Piscataway, NJ. pp. 60–65.

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

[12] T. Mens. 2008. Introduction and roadmap: history and challenges of software evolution. In: *Software Evolution* (Eds T. Mens and S. Demeyer). Springer-Verlag, Berlin.

[13] R. G. Canning. 1972. That maintenance 'iceberg'. *EDP Analyzer*, October, 1–14.

[14] A. V. Mayrhauser and A. M. Vans. 1995. Program comprehension during software maintenance and evolution. *IEEE Computer*, August, 44–55.

[15] V. T. Rajlich and N. Wilde. 2002. *The Role of Concepts in Program Comprehension*. IWPC, June 2002, Paris, France. IEEE Computer Society Press, Piscataway, NJ. pp. 271–278.

[16] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. 1994. Program understanding and the concept assignment problem. *Communications of the ACM*, May, 72–82.

[17] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. 2000. *Identifying the Starting Impact Set of a Maintenance Request: A Case Study*. CSMR, February 2000, Zurich, Switzerland. IEEE Computer Society Press, Los Alamitos, CA. pp. 227–230.

[18] S. A. Bohner and R. S. Arnold. 1996. *Software Change Impact Analysis* (Eds S. A. Bohner and R. S. Arnold). IEEE Computer Society Press, Los Alamitos, CA.

[19] W. P. Stevens, G. J. Myers, and L. Constantine. 1974. Structured design. *IBM Systems Journal*, 13(2), 115–139.

[20] D. P. Freedman and G. M. Weinberg. 1981. A checklist for potential side effects of a maintenance change. In: *Techniques of Program and System Maintenance* (Ed. G. Parikh), pp. 93–100. QED Information Sciences Inc., Wellesley, MA, USA.

[21] T. Mens. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering*, February, pp. 126–139.

[22] V. T. Rajlich. 1997. *A Model for Change Propagation Based on Graph Rewriting*. ICSM, October 1997, Bari, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 84–91.

[23] A. E. Hassan and R. C. Holt. 2004. *Predicting Change Propagation in Software System*. ICSM, September 2004, Chicago, Illinois. IEEE Computer Society Press, Los Alamitos, CA. pp. 284–293.

[24] S. Naik and P. Tripathy. 2008. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Hoboken, NJ. pp. 93–100.

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

[26] SWEBOK. 2004. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Scociety Press, Los Alamitos, CA.

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

[28] T. M. Pigoski. 1996. *Practical Software Maintenance*. John Wiley & Sons, New York, NY.

[29] ISO/IEC 12207:2006 and IEEE Std 12207-2006. 2008. *System and Software Engineering - Software Life Cycle Processes*. Geneva, Switzerland.

[30] E. Bersoff, V. Henderson, and S. Siegel. 1980. *Software Configuration Management—An Investment in Product Integrity*. Prentice Hall, Englewood Cliffs, NJ.

[31] J. Estublier, D. Leblang, A. V. Hock, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. 2005. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, October, 383–430.

[32] Software Technology Support Center. 2005. Configuration management fundamentals. *CrossTalk A Journal of Defense Software Engineering*, July, 10–15.

[33] S. Feldman. 1979. Make—a program for maintaining computer programs. *Software-Practice and Experience*, 9, 255–265.

[34] M. Rochkind. 1975. The source code control system. *IEEE Transactions on Software Engineering*, December, 364–370.

[35] W. Tichy. 1985. Rcs—a system for version control. *Software-Practice and Experience*, 15, 637–654.

[36] P. Louridas. 2006. Version control. *IEEE Software*, January–February, 104–107.

[37] ClearCase. 2009. *IBM Rational*, October. Available at http://www.ibm.com/software/awdtools/clearcase/.

[38] M. Ben-Menachem. 1994. *Software Configuration Management Guidebook*. McGraw-Hill, New York, NY.

[39] M. Glinz and R. J. Wieringa. 2007. Stakeholders in requirements engineering. *IEEE Software*, March–April, pp. 18–20.

[40] W. W. Royce. 1970. *Managing the Development of Large Software System: Concepts and Techniques*. Proceedings of IEEE WESCON, August 1970. pp. 1–9; Republished in 1987 by ICSE, Monterey, CA. 1987, pp. 328–338.

[41] B. W. Boehm. 1988. A spiral model of software development and maintenance. *IEEE Computer*, May, 61–72.

EXERCISES

1. Define the terms process, life-cycle, and model.

2. Discuss different ways of changing the following characteristics from one stage to another stage during the CSS life cycle model.

(a) Staff expertise

(b) Software architecture

(c) Software decay

(d) Economics

3. Discuss the similarities and differences between the staged models of CSS systems and FLOSS systems.

4. What is the difference between change propagation and change impact analysis?

5. Discuss the role of concept in program comprehension.

6. What is ripple effect? In what way is it different from side effect?

7. Explain why it is important to conduct program comprehension before impact analysis.

8. What are the advantages of the quick fix model and why is it still used?

9. Explain the differences between the incremental and the iterative development models?

10. What are the drawbacks of the iterative enhancement model?

11. Discuss the differences between the iterative enhancement and the full reuse models.

12. Discuss the major differences between the IEEE 1219 and the ISO/IEC 14764 standards for software maintenance procedure.

13. How would each of the following groups use the information contained in a CR?

(a) Maintainers

(b) Management

(c) Quality assurance auditors

(d) Sustaining test engineers

(e) Customers

14. What are some of the factors that you would think about when reviewing a CR?

15. Why should a maintainer categorize CRs into different groups? What are some of the factors that should be considered when categorizing those CRs?