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.

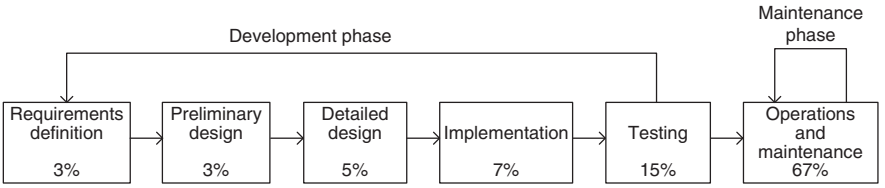

FIGURE 3.1 Traditional SDLC model. From Reference 1. © 1988 John Wiley & Sons

- *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

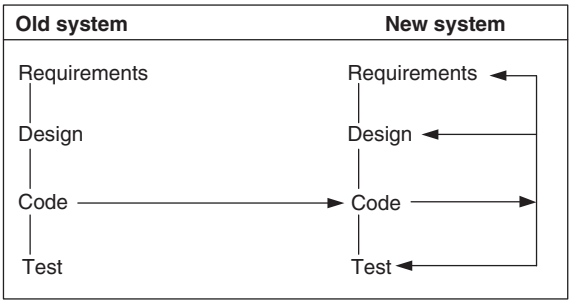

FIGURE 3.2 The quick fix model. From Reference 2. © 1990 IEEE

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

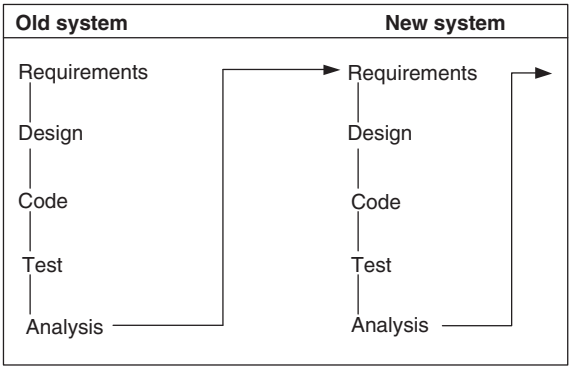

FIGURE 3.3 The iterative enhancement model. From Reference 2. © 1990 IEEE

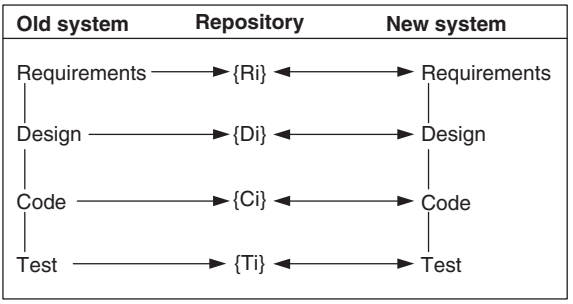

FIGURE 3.4 The full reuse model. From Reference 2. © 1990 IEEE

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.

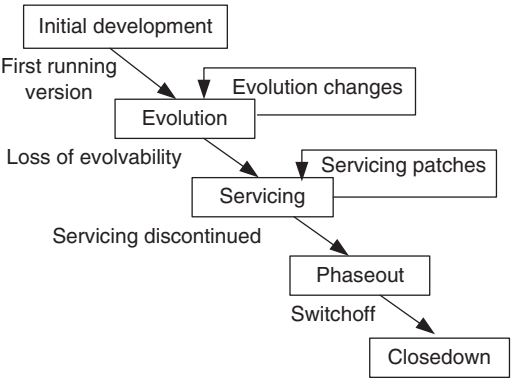

FIGURE 3.5 The simple staged model for the CSS life cycle. From Reference 6. © 2000 IEEE

- *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

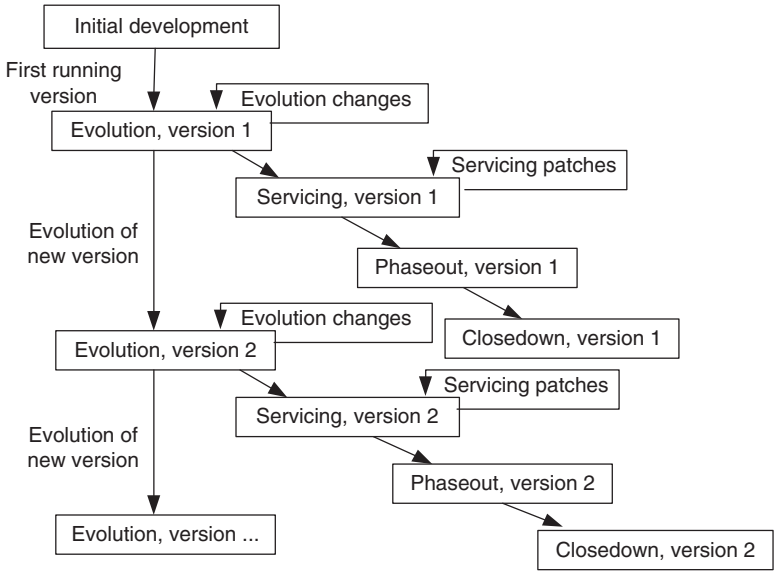

FIGURE 3.6 The versioned staged model for the CSS life cycle. From Reference 6. © 2000 IEEE

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

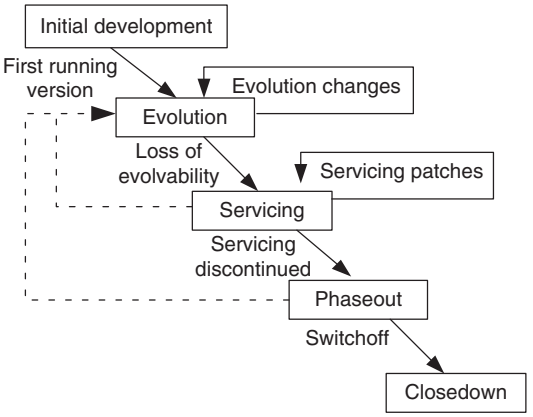

FIGURE 3.7 The staged model for the FLOSS system. From Reference 9. © 2007 ACM

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

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.

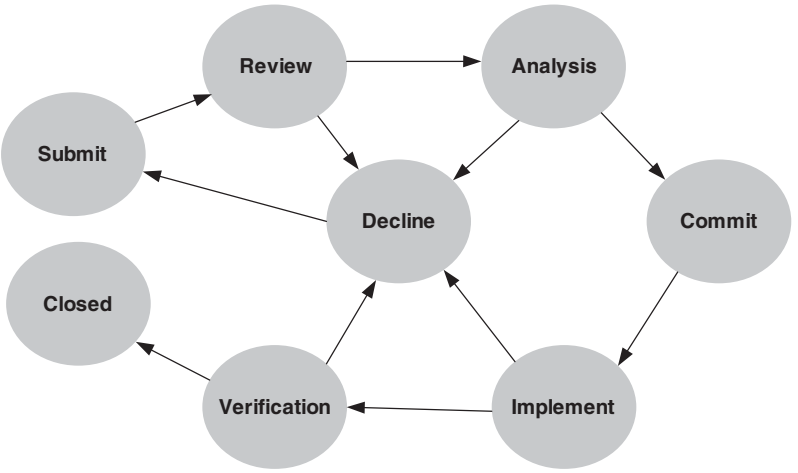

FIGURE 3.27 State transition diagram of a CR

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