CHAPTER 4

Software Process Models

Chapter Objectives

- Introduce the generic concept of software engineering process models.
- Discuss the three traditional process models.
    - Waterfall
    - Incremental
    - Spiral
- Discuss the chief programming team approach.
- Describe the rational unified process along with the significance of entry and exit criteria for all the processes.
- Assess processes in terms of the capability maturity model (CMM) and capability maturity model integrated (CMMI).
- Discuss the need to modify and refine a standard process.

4.1 Software Processes

We have mentioned processes in earlier chapters and have indicated the significant roles they play in software engineering. As shown in Chapter 2, the process of developing and supporting software often requires many distinct tasks to be performed by different people in some related sequences. When software engineers are left to perform tasks based on their own experience, background, and values, they do not necessarily perceive and perform the tasks the same way or in the same order. They sometimes do not even perform the same tasks. This inconsistency causes projects to take a longer time with poor end products and, in worse situations, total project failure. Watts Humphrey has written extensively on software processes and process improvement in general and has also introduced the personal software process at the individual level in his book *Introduction to the Personal Software Process* (1997).

In this chapter we will cover the traditional software processes, and leave the emerging processes, such as the Agile processes, to the next chapter. We will also cover the general evaluation and assessment of processes in this chapter.

4.1.1 Goal of Software Process Models

The goal of a software process model is to provide guidance for systematically coordinating and controlling the tasks that must be performed in order to achieve the end product and the project objectives. Similar to the definition provided in Chapter 2 for software development process, a process model defines the following:

- A set of tasks that need to be performed
- The input to and output from each task
- The preconditions and postconditions for each task
- The sequence and flow of these tasks

We might ask whether a software development process is necessary if there is only one person developing the software. The answer is that it depends. If the software development process is viewed as only a coordinating and controlling agent, then there is no need since there is only one person. However, if the process is viewed as a prescriptive roadmap for generating various intermediate deliverables in addition to the executable code—for

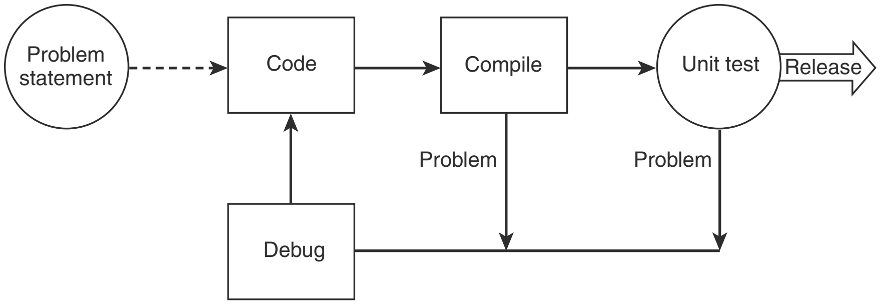

Figure 4.1
A simple process.

example, a design document, a user guide, test cases—then even a one-person software development project may need a process.

4.1.2 The "Simplest" Process Model

When programmers are left alone, they naturally gravitate to what is often perceived as the single most important task, coding. As indicated in Chapter 1, most of the people involved with the information technology field, including the software engineers, start in the profession by learning how to write code in some programming language. Figure 4.1 shows this perhaps simple process. It depicts the tasks involved in the code-compile-unit test cycle. Because coding is usually considered the central task in this process, the model is sometimes known as the code-and-fix model. When there is a problem detected in compilation or in the unit testing, debugging, which is problem analysis and resolution, is performed. The code is then modified to reflect the problem correction and recompiled. Unit testing then follows. When unit testing is completed and all the detected problems resolved, the code is released.

Two areas of Figure 4.1 deserve some attention. The first is the problem statement, the precursor to what we now call requirements specifications in software engineering. The significance of this area was neither recognized nor appreciated in the early days. The second area is testing. Unit testing the code was performed in an informal way by the author of the code. Since the problem statement was often allowed to be incomplete or unclear, the testing of the code to ensure that it met the problem statement was also itself often incomplete. The testing effort often reflected what the programmer understood the problem to be.

Even with all the shortcomings, this simple process model served many early projects. As software projects increased in complexity, more tasks, such as design and integration, were introduced. As more people participated in a software project, better coordination was introduced. The tasks in the process, the relationship among them, and the flow of these tasks become better defined.

As software engineers gained more experience, different software development models were introduced to solve different concerns. Today there is an understanding that there is no one process model that will fit all the software projects. In this chapter, some of the earlier process models and associated topics will be introduced. The more recently developed process models will be discussed in Chapter 5.

4.2 Traditional Process Models

In this section, several of the earlier software development models will be presented. Each of these models has also been adapted and modified to fit different situations. We will present the models only in their generic form.

4.2.1 Waterfall Model

The waterfall software development process model is probably the oldest publicized model. It is sometimes referred to as the classic software life cycle model. Although many organizations utilized this model, Royce (1970) is one of the earliest people to write about this model. The name of the waterfall model is derived from the process it represents: tasks occur sequentially one after another, with the output from one task dropping into the next task, as shown in Figure 4.2.

Resembling a multilayered waterfall, the model provided many advantages, especially to the software project managers in the early 1970s. It served as a tool for managing software projects and represented the software life cycle as the software went through different and distinct stages of development. It gave the project managers a way to describe the status more precisely than just saying the software is "almost complete." Although we now recognize many shortcomings to this process, the waterfall model also has many positive aspects:

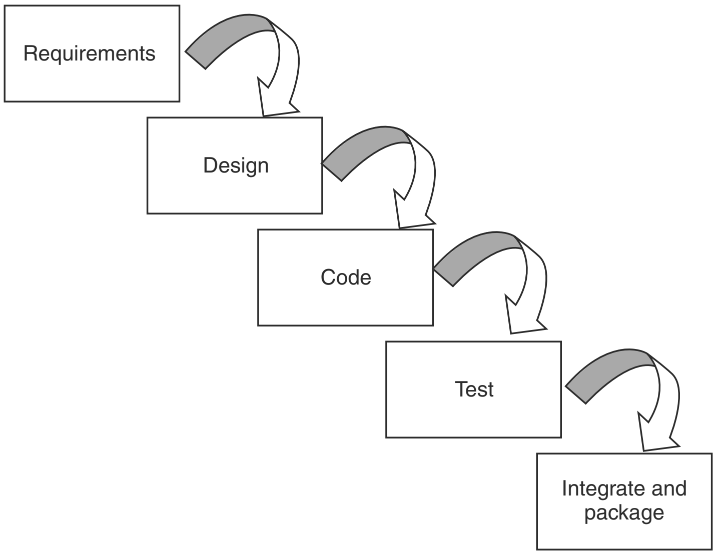

Figure 4.2
A waterfall model.

- Requirements must be specified in the first step.
- Four main tasks must be completed before the software can be packaged for release: requirements, design, code, and test.
- The output from each stage is fed into the next stage in sequence.
- The software project may be tracked as it moves sequentially through specific and identifiable stages.

Because of the heavy amount of documents that were generated with requirements, design, and testing, the waterfall model also became known as the document-driven approach.

Many modifications to the basic waterfall model have been applied throughout the years since its early definition, each addressing some of its shortcomings. For example, the model was usually viewed as a single iteration model that provided very little task overlapping. Thus backward arrows were introduced in the diagram to depict the addition of iterative activities. The waterfall model has also been criticized for its limited interaction with users at only the requirements phase and at the delivery of the software. The implementers of the waterfall model included the users and the customers in the design phase with techniques such as joint application development (JAD) and in the testing phase.

The single most important contribution of the waterfall model is probably that it gave software engineering a process upon which software development could focus its attention. As a result of this focus on process, the waterfall model as well as the software quality problems in general, started to be resolved through the years.

4.2.2 Chief Programmer Team Approach

The chief programmer team approach is a type of coordination and management methodology rather than a software process. The concept was a popular organizational idea in the mid 1970s.

In his book, *The Mythical Man Month* (1975), Fred Brooks described a small-team approach to coordinate the activities of software development. He attributed the original proposal to Harlan Mills of IBM. The proposed approach mimics a surgical team organization where there is a chief surgeon and other specialists to support the chief surgeon. Instead of a large number of people all working on smaller pieces of the problem, there is a chief programmer who plans, divides, and assigns the work to the different specialists. The chief programmer acts just like a chief surgeon in a surgical team and directs all the work activities. The team size should be about 7 to 10 people, composed of specialists such as designer, programmers, tester, documentation editors, and the chief programmer. This approach made sense and is a precursor to dividing a large problem into multiple components, then having the small chief programming teams develop the components.

4.2.3 Incremental Model

The incremental model may be viewed as a modification to the waterfall model. As software projects increased in size, it was recognized that it is much easier, and sometimes necessary, to develop the software if the large projects are subdivided into smaller components, which may thus be developed incrementally and iteratively. In the early days, each component followed a waterfall process model, passing through each step iteratively. In the incremental model the components were developed in an overlapping fashion, as shown in Figure 4.3. The components all had to be integrated and then tested as a whole in a final system test. The incremental model provided a certain amount of risk containment. If any one component ran into trouble, the other components were able to still continue to be devel-

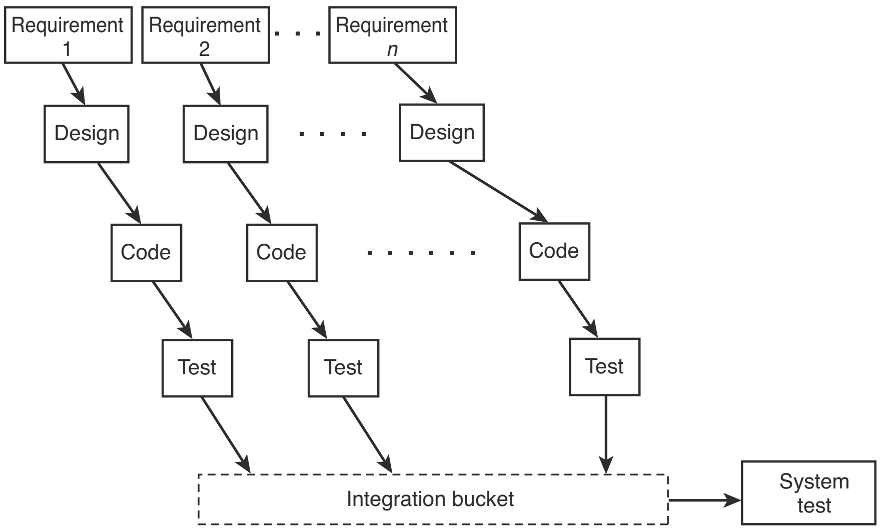

Figure 4.3
A multiple components incremental model.

oped independently. Unless the problem was a universal one, such as the underlying technology being faulty, one problem would not hold up the entire development process.

Another perspective in utilizing the incremental model is to first develop the "core" software that contains most of the required functionality. The first increment may be delivered to users and customers as Release 1. Additional functionality and supplemental features are then developed and delivered separately as they are completed, becoming Release 2, Release 3, and so on. Utilizing the incremental model in this fashion provides an approach that is more akin to an evolutionary software product development. When utilized in this development mode, the model in Figure 4.3 would not have the integration bucket.

The incremental model in Figure 4.3 would have individual releases. For example, Requirement 1 would be the core functionality release. Other requirements would each depict different deliveries. Figure 4.4 depicts the incremental, multiple release scenario where the first release, Release 1, is the core function, followed by subsequent releases that may include fixes of bugs from previous releases along with new functional features. The multiple release incremental model also makes it possible to evolve the first release, which may have flaws, into an ideal solution through subsequent

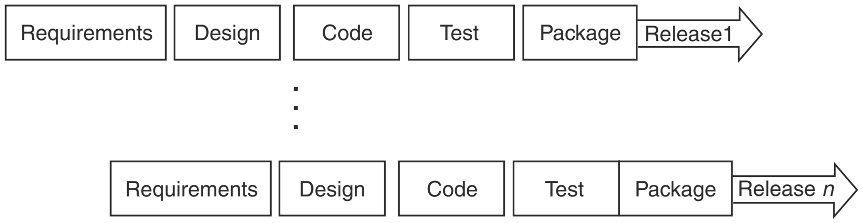

Figure 4.4
A multiple release incremental model.

releases. Thus it facilitates evolutionary software development and management, a model that has been advocated by many, especially by Tom Gilb who has written recently about the "evo" process (2004). The number of releases for a software project will depend on the nature and goals of the project. Although each release is independently built, there is a link between releases because the existing design and code of the previous release is the basis upon which future releases are built.

Both incremental models utilize the "divide and conquer" methodology where a large, complex problem is decomposed into parts. The difficulty with this model is that such problems are also intertwined, making the decoupling of the parts into independently implementable components difficult. It will require a deep understanding of the problem, the solution, and the usage environment. Overlapping the different increments is another area of difficulty in that there may be some amount of sequential dependency of information among the components. How much overlapping can take place depends on how much prerequisite information is required.

4.2.4 Spiral Model

Another evolutionary approach to software development is the spiral model, proposed by Barry Boehm at a time when there were concerns with the waterfall model's document-driven approach. The early spiral model is based on experiences with various large government software projects at TRW. An important aspect of the spiral model is its emphasis in the reduction of risks in software development. The model is thus a risk-driven approach to software process. It provides a cyclic approach to incrementally develop the software system while reducing the project risk as the project goes through cycles of development, as illustrated in Figure 4.5.

The spiral model has four quadrants, and the software project traverses through the quadrants as it is incrementally developed. As shown in the fig-

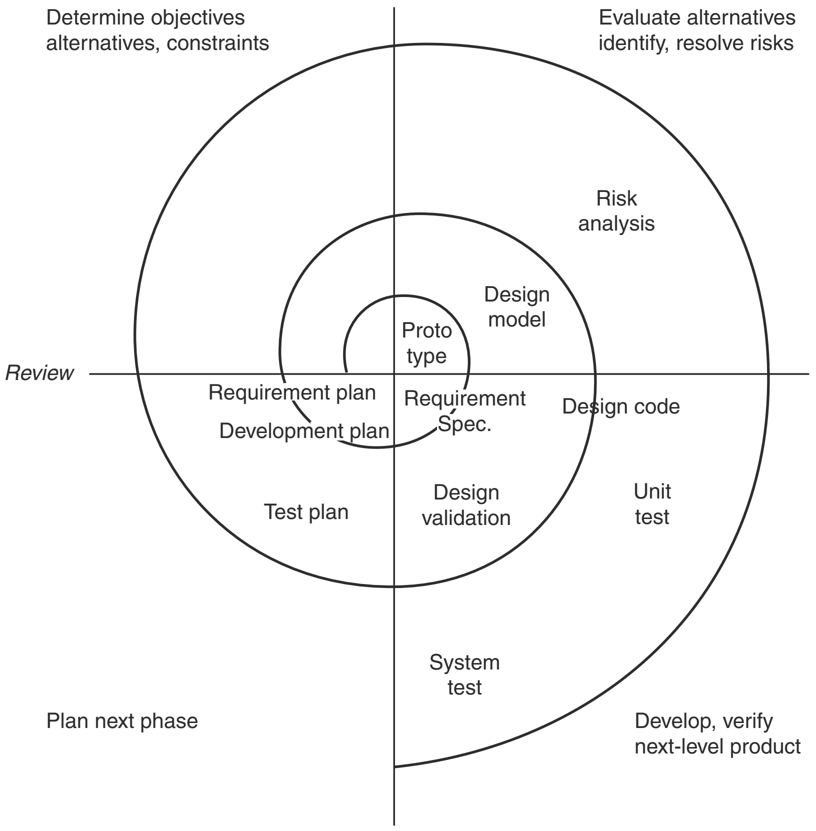

Figure 4.5
A spiral model.

ure, the spiral path may not be very smooth. Each cycle involves the same sequence of steps for each of the concerns, components, or artifacts.

Equally applicable to software development and software enhancement projects, the spiral model is based on some objective. The spiral process then involves the continuous "testing" or iterations of this objective or requirement until either the end result is achieved or shown to be unachievable. A typical traversal through the four quadrants is as follows:

1. Identify the objectives, alternatives, or constraints for each cycle of the spiral.
2. Evaluate the alternatives relative to the objectives and constraints. In performing this step, many of the risks are identified and evaluated.

3. Depending on the amount of and type of identified risks, develop a prototype, more detailed evaluation, an evolutionary development, or some other step to further reduce the risk of achieving the identified objective. On the other hand, if the risk is substantially reduced, the next step may just be a task such as requirements, design, or code.
4. Validate the achievement of the objective and plan for the next cycle.

An integral part of the cycle is the review of all the activities and products completed in the cycle by all the major stakeholders involved in the project. The review's major objective here is to ensure that all the parties are continuously committed to the project and concur with the approach for the next phase of the project.

Because the spiral model is based on risk reduction of the project through iterations, several convenient features are built into it.

- The model incorporates prototyping and modeling as an integral part of the process.
- It allows iterative and evolutionary approaches to all activities based on the amount of risks involved.
- The model does not preclude the rework of an earlier activity if a better alternative or a new risk is identified.

The ironic part of the spiral model is that one of its risks is the reliance on risk assessment expertise. Not all software engineers are trained or experienced in risk identification and risk analysis.

4.3 A More Modern Process

In recent years, many newer processes have been introduced. A fairly recent and popular process, which was initially developed by Rational Software Corporation, is described in this section.

4.3.1 General Foundations of Rational Unified Process Framework

The Rational Unified Process (RUP) is a software process framework, rather than a single process, developed by Rational Software Corporation, which was acquired by IBM. The origin of RUP is rooted in the original 1987 Objectory Process and the 1997 Rational Objectory Process as well as the **Unified Modeling Language** (**UML**). Fowler and Scott (1999) provide extensive coverage of UML in their book, *UML Distilled*. In many ways, RUP has incorporated many of the earlier experiences from the incremental and iterative process model and the spiral model. This process framework is driven by three major concepts:

DEFINITION

Unified Modeling Language (UML)

An object-oriented modeling language that provided the elements and relationships to model software requirements and design.

- Use case and requirements driven
- Architecture centric
- Iterative and incremental

Use cases have been used mainly to capture requirements, but they may be used to describe any interaction between the software system and anything external such as a user of the system. This approach is different from the traditional functional specification approach where the functionality of the system is described but the complete interaction between the system and its users is not. The emphasis is on the users and the values to the users. Use case driven means that the development process is initiated by the use case, that the designs are developed from the use cases, and that the test cases are derived from the use cases. Thus use cases drive this software development process.

Architecture plays a significant role in RUP, describing the static and the dynamic aspects of the overall system, with the more important aspects highlighted and the less important details left out. In RUP, the architecture initially provides what Jacobson, Booch, and Rumbaugh (1999) call the "form" of the system, which is use case independent. It describes the high-level design, such as the user interface standard or error processing, which transcends all the use cases. From this baseline, the architecture is refined to accommodate the important major use cases. Each of the important use cases represents a key component of the software system, providing more details to the design. As more details of the use cases are considered, the architecture also evolves into a more mature and stable design. The use cases drive the architecture and the architecture influences the choices of use cases.

RUP is also iterative and incremental in that it promotes large software to be developed in smaller pieces or increments. In developing the chosen increment, RUP promotes the iterative approach. The first iteration would include all the use cases or requirements representing that increment or slice of the product. The second iteration would handle all the most important risks in the chosen increment. Successive iterations would then build upon the results of the previous iterations.

These three concepts of use case, architecture, and iterative and incremental form the basis of RUP. For more comprehensive studies on RUP, refer to the books in the Suggested Readings at the end of this chapter.

4.3.2 The Phases of RUP

The phases in RUP are not named after the activities such as design, testing, or coding; an iteration may include many activities in varying degrees. There are four phases in RUP:

- Inception
- Elaboration
- Construction
- Transition

As an increment of the product is developed, it may go through several iterations within each phase. The degree of the activities such as requirements specifications, testing, or coding that takes place within each phase is also different.

The inception phase may be viewed as the beginning stage when the product increment is still in an early stage of uncertainty. An initial idea is being developed during this phase. During the elaboration phase, the detailed use cases are being formulated, and the architecture and design are getting firmed up. The product increment is built, coded, and tested during the construction phase. Finally, the product increment is released to a small restricted group of users during the transition phase for further testing and correction. It is then released to the general public. Figure 4.6 provides a view of the four phases of the rational unified process and how the development activities relate to the phases. The software development activities on the left in the figure all flow through the four phases. Each activity will be in "peak mode" in different phases. The extent of each activity is represented by the thickness of the bar with relative approximations showing where the activities will peak. Although not explicitly shown in this figure, any activity such as design may also iterate several times within a phase. RUP not only provides incremental development but also includes iterative

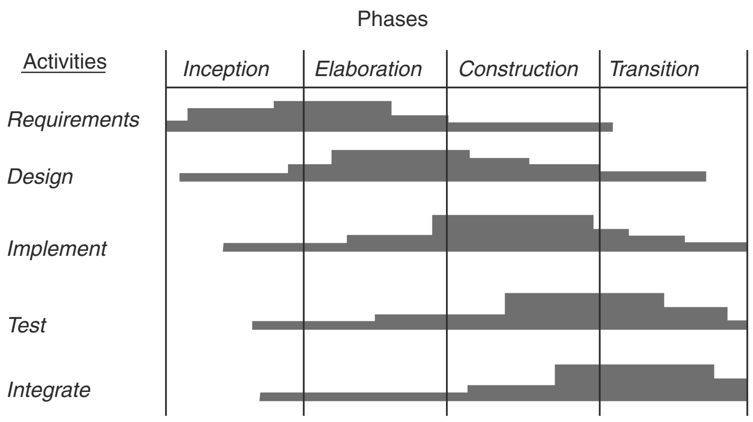

Figure 4.6
The rational unified process (RUP).

development. The four phases provide a mechanism to track project milestones.

Inception Phase

Inception is a planning phase that includes the following primary objectives:

- Establish the scope and clarify the goals of the software project.
- Establish the critical use cases and the major scenarios that will drive the architecture and design.
- Establish some architecture and early design alternatives.
- Estimate the schedule and required resources.
- Plan the implementation, testing, integration, and configuration methodologies.
- Estimate the potential risks.

In order to accomplish these primary objectives, the requirements activities must be building up to a full crescendo. The architecture of the software system is narrowed down and various design alternatives must be considered during this phase. Implementation, testing, integration methodologies, tools, etc. are being planned during the inception phase. The overall project schedule, needed resources, and potential risks are estimated based on the major requirements and early architecture. The project goals and measurement are established. The stakeholders should all concur with the estimates and the plan for the project.

Elaboration Phase

Elaboration may be the most critical phase of the Rational Unified Process. At the end of this phase, most of the "unknowns" should be resolved. The primary objectives of this phase include the following:

- Establish all the major and critical requirements for the system.
- Establish and demonstrate the baseline design.
- Establish the implementation, test, and integration platforms and methodologies.
- Establish the major test scenarios.
- Establish the measurement and metrics for the agreed-upon goals.
- Organize and set up all the needed resources for implementation, testing, and integration.

In order to achieve these objectives, all the requirements must be gathered, analyzed, understood, documented, and agreed to by all parties during the elaboration phase. Any prototyping of requirements must be completed, as well as the architecture and most of the design. Any design feasibility questions must be prototyped and answered. Major test scenarios are identified during this phase. Plans for implementation, testing, and integration are completed. Resources needed for implementation, testing, and integration are acquired and organized. Education for any new methodology or tools for implementation, testing, or integration is completed. A clear metric and measurement system is accepted and resources for measurement are acquired. That is, the project control for the rest of the phases is set in place. At the end of the elaboration phase, the software project is ready to go into full implementation and testing mode.

Construction Phase

The construction phase is equal to the production phase in manufacturing. At the end of this phase, the code for the software should be complete and all the major requirements tested. The following objectives are the key points of this phase:

- Complete the implementation in a timely manner within estimated cost.

- Achieve the version of the code that is releasable to a restricted set of Alpha test sites.
- Establish the remaining activities that need to be completed to achieve the goals of the project.

In order to meet these objectives, the coding of the design must be completed in the construction phase. All the planned test cases must be executed and most of the discovered problems are fixed in this phase. The software must meet most of the established goals of the project and the measurements taken must validate that. Assessment must be made of how much and what remaining activities are needed to achieve the planned goals. For example, an assessment of whether the software product quality goal is met needs to be performed. Any necessary activities to follow up on this goal, such as additional testing and fixes, must be set up.

Transition Phase

The transition phase is the last phase prior to the release of the software to general users. All the fixes and components are integrated. The noncode artifacts, such as manuals and educational materials, are also integrated into the complete product. The key objectives of this phase are the following:

- Establish the final software product for general release.
- Establish user readiness and acceptance of the software.
- Establish support readiness.
- Gain concurrence for release and deployment.

All Alpha and Beta tests with a restricted number of users must be completed and the fixes to the discovered problems are integrated into the final release during this phase. Users must be trained. All transitional activities, such as data migration and usage process modifications, are completed prior to the end of this phase. The software support group is trained and must stand ready to service users. If this is a software product for external sale, the sales organization must be educated, and the marketing material must be created and be available for distribution. A final assessment of the software, in terms of its goals, is performed and a decision on release is made.

4.4 Entry and Exit Criteria

The processes discussed so far have emphasized the sequencing and coordination of activities. The Rational Unified Process does, however, go further and provide some guidelines on what artifacts need to be developed by whom. Still, there are very few guidelines on how much of each activity must be performed. That is, what are the exit criteria for each activity and the entry criteria for the next follow-on activity?

Figure 4.7 shows that the entry criteria for an activity must be met before the activity can start. The exit criteria must be met before the activity can be considered complete and before the next activity may start. The difficulty comes in when the activities overlap in a concurrent manner. The entry and exit criteria must then be defined with much more granularity.

4.4.1 Entry Criteria

Prior to performing any of the activities portrayed in the process diagram, we must ask for the condition that allows the performer of that activity to start. The conditions for initiating the activity define the entry criteria. These include a listing and a description of the following resources:

- Required artifacts
- Required people
- Required tools
- Required definition of the activity to be performed

There must be a specified list of artifacts. Just listing them alone is not enough. These artifacts must be in a condition that they are usable by the activity. As an example, consider the design task that needs the requirement

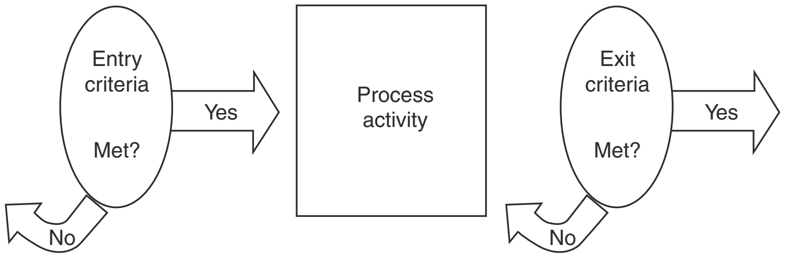

Figure 4.7
Entry and exit criteria.

specifications. The state of each specification must be defined as "completed," which means the following:

- All specifications have been reviewed by the customers and other stakeholders.
- All exceptions found during the review are changed.
- The modified specifications are accepted by all parties.

When the requirement specifications have attained these conditions, they are considered to be complete and to have met the entry criteria for the design task. Note that if the desired process is incrementally driven, the "completed" state may apply to only the incremental requirement that is needed for the next activity of design.

The people required to perform the task must also be specified. They must be in a "ready" state, meaning that they are available and can be applied to the task prior to the commencement of the task.

Any tools that are required or that may be later used to perform a task are specified. Again, just listing the tools is not enough. The rationale and the expectations of using any tool for the task must be spelled out. The people who are pegged to use the tools have to be identified and trained prior to the beginning of performing the task.

The most obvious requirement, yet one that is often left out, is the definition and explanation of the task itself. If there is not a clear understanding of the task, different individuals may perform the task differently, which can cause erratic results.

The definition of the entry criteria for each of the steps or activities described in a process will bring the high-level definition of process down to an executable level. It also allows each part of the organization to tailor the process by specifying slightly different entry criteria for each of the tasks in a process.

4.4.2 Exit Criteria

Before an activity is declared complete, the exit criteria for such a declaration need to be specified ahead of time. Only when those criteria are met can the activity be considered complete. Again, in the case of incremental and overlapping activities, the exit criteria must then be declared at a much finer level.

The main purpose of the exit criteria is to describe the artifacts that must be available for the next activity. A clear description of what must be included in each completed artifact must be defined. Furthermore, it is important to clearly spell out any conditions such as the following:

- All the artifacts are reviewed.
- All or some prespecified percentage of the errors are corrected.
- People in the downstream activities have concurred and accepted the artifacts.

There are other conditions that we may include as part of the exit criteria—for example, that the person who is to participate in the next downstream activity is freed from the current activity. The important thing is that the exit criteria should be clearly specified ahead of time.

4.5 Process Assessment Models

Software engineering development and support processes continue to be modified, improved, and invented through countless studies, experiments, and implementations, some achieving great success and some utter failure (see Cusumano, et al. 2003; MacCormack 2001). The software industry has embraced the importance of software development processes for years. One of the key organizations that has contributed, advanced, and advocated the software development processes is the Software Engineering Institute (SEI), a research and development center funded by the U.S. Department of Defense and located on the Carnegie Mellon campus. Its stated core purpose is to "help others make measured improvements on their software engineering capabilities." (See the Suggested Readings for the SEI web address.)

Another organization that has contributed to software engineering is the International Standards Organization (ISO). Its ISO 9000 series of software quality standards includes the ISO/IEC 90003:2004 document, which provides guidance for organizations to apply ISO9001:2000 to the computer software activities. Specifically, there are four documents—ISO/IES 9126–1 through ISO/IES 9126–4 that address various aspects of software quality.

Also, the ISO/IEC 12207 Standard for Information Technology document discusses and provides a framework for software life cycle processes. These documents can be purchased from the ISO website listed in the Suggested Readings. Both the SEI and ISO contributed greatly to assessing the maturity of the organization in their software development and support.

4.5.1 SEI's Capability Maturity Model

The Capability Maturity Model (CMM), initially proposed by SEI, is a framework that is used to help a software organization define its level of maturity in software development. (See the Suggested Readings for information on the original document on CMM in 1993.) The model presents five levels of maturity and is based on the concept of continuous improvements. The level of maturity of a software organization is determined by its practice of different sets of key software development process activities. The levels are sequential and accumulative in that an organization assessed at a Level x is expected to have elevated from Level $(x - 1)$. There is a list of "officially" trained CMM assessors, which may be obtained from SEI, who perform the appraisal of an organization and provide the feedback on the strength and weakness of the organizations' key processes activities and commitments. The five levels of CMM are represented in Figure 4.8.

At the initial level (Level 1 in the figure) an organization has no process, and any success is probably attributed to a strong and experienced leader. The probability of repeating this success is low. As an organization defines, practices, and continuously improves on the different processes, it moves up the scale of maturity.

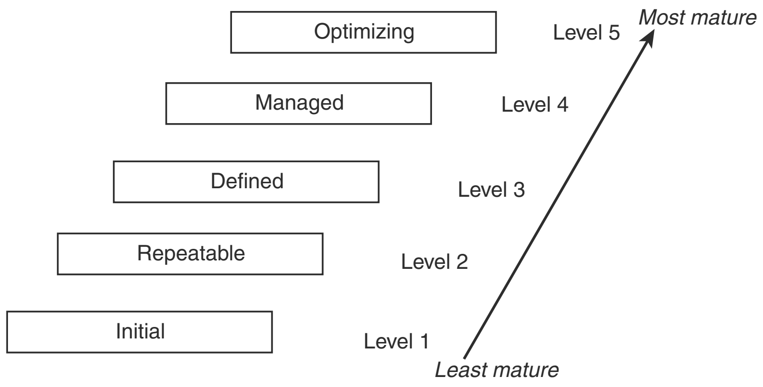

Figure 4.8
The five levels of the original Capability Maturity Model.

At Level 2, there are six key processes that an organization must master:

- Requirements management
- Software project tracking and oversight
- Software quality assurance
- Software project planning
- Subcontract management
- Software configuration management

An organization at Level 2 (the repeatable level) has mastered these key project management related processes and is expected to be able to repeat its success when given a similar project.

In order for an organization to elevate from Level 2 to Level 3 (the defined level), it must master seven more key processes:

- Organization process focus
- Training program
- Software product engineering
- Peer reviews
- Organization process definition
- Integrated software management
- Intergroup coordination

At Level 3, the organization has mastered the major processes related to construction of software along with additional project management related processes.

An organization moves up to Level 4 (the managed level) when it focuses its effort on quantitative and quality management in addition to all the key processes of Levels 2 and 3. As such, two more key processes are added:

- Quantitative process management
- Software quality management

Metrics and measurements of the process and of the software artifacts are introduced. Quantitative management of attributes such as quality, productivity, or efficiency is part of the organization at this level. With the captured measurements, the feedback from prior activities becomes visible, which allows future improvements to both the processes and the product.

The highest level of CMM is Level 5 (the optimizing level). The emphasis here is on continuous improvement. In order to facilitate such improvement, three key processes must be included:

- Defect prevention
- Technology change management
- Process change management

All the key processes at this ultimate level contribute to an organization poised for changes and improvements.

SEI's original CMM has been used by thousands of software organizations across multiple countries. Today, large and small companies around the world—from Wipro in India to Neusoft in China—have attained Level 5. Occasionally, several organizations within the same company may be assessed at different levels. For example, Lockheed Martin, the U.S. technological giant in the aerospace industry, is an example of a company that has several organizations within it that have attained CMM Level 5. The United States leads the world in the number of CMM assessed organizations. Some organizations, however, just utilize the CMM framework for self-improvement and never request any formal assessment. Others have used the assessed CMM level as a marketing tool for their organizations. This is especially evident in the software service sector.

The time required for ascending from one level to the next higher level is usually on the order of one or two years, rarely in months or days.

4.5.2 SEI's Capability Maturity Model Integrated

In 2001, the CMM was upgraded to the Capability Maturity Model Integrated (CMMI). Again, the important factor to remember is that CMMI's purpose is to provide guidance for improving the processes of an organization and its ability to develop, manage, and support the software product and services. While there are multiple aspects of the CMMI (e.g., systems engineering, software engineering, integrated product and process development, and supplier sourcing), the one we are interested in and will be discussing here is the CMMI-SW, the software engineering model.

The CMMI-SW model has two representations:

- Continuous
- Staged

The continuous representation model is more applicable to the assessment and improvement of processes. The staged representation model is, like the CMM, better applied to assessing the maturity of an organization. In the next three sections we will first discuss the three key concepts common to both the continuous and the staged representations, and we will then delineate the differences between the two representations.

The Process Areas of CMMI

The first key concept related to both the continuous and staged representations in CMMI is that there are 25 major process areas covering four major categories of processes: (1) process management, (2) project management, (3) engineering, and (4) support.

The following five process areas fall under process management:

- Organizational process focus
- Organizational process definition
- Organizational training
- Organizational process performance
- Organizational innovation and deployment

The following eight process areas fall under project management:

- Project planning
- Project monitoring and control
- Supplier agreement management
- Integrated project management
- Risk management
- Integrated teaming
- Integrated supplier management
- Quantitative project management

The following six process areas fall under engineering:

- Requirements development
- Requirements management
- Technical solution
- Product integration
- Verification
- Validation

The last six process areas fall under support:

- Configuration management
- Process and product quality assurance
- Measurement and analysis
- Organizational environment for integration
- Decision analysis and resolution
- Causal analysis and resolution

These 25 process areas form the basis for process evaluation in CMMI.

Levels in CMMI

Both the continuous and staged representations utilize levels for assessment. In the case of continuous representation, there are 6 (0–5) capability levels for assessing the process areas. The staged representation has 5 (1–5) maturity levels for assessing the organization. Figure 4.9 compares the

Level 5	Optimizing	Optimizing
Level 4	Quantitatively managed	Quantitatively managed
Level 3	Defined	Defined
Level 2	Managed	Managed
Level 1	Performed	Initial
Level 0	Incomplete	------
	Continuous (Capability Levels)	Staged (Maturity Levels)

Figure 4.9
Different levels in CMMI.

capability and staged levels. The utilization of levels for designating assessment results is the second key concept in CMMI. Note that the names for Levels 2 through 5 are the same for both the continuous capability levels and the staged maturity levels. However, as will be explained in a later section of this chapter, these levels are different in their structures.

Goals and Practices in CMMI

A third key concept that is common to both the continuous and the staged representations in CMMI is the notion of goals and practices. Within each of the 25 process areas that were mentioned earlier is a designated set of specific goals that uniquely describe the specific practices that must be implemented to satisfy that process area. Furthermore, the specific practices associated with each of the goals are also unique to each goal. Thus the specific practices are all different, as illustrated in Figure 4.10.

As an example, consider one of the 25 process areas, organizational process focus. For this process area, there are two specific goals with their respective specific practices. Specific goal 1 has three specific practices, and specific goal 2 has four specific practices.

- Specific goal 1: Strengths, weaknesses, and improvement opportunities for the organization's processes are identified periodically and as needed.
  - Specific practice 1.1: Establish organizational process needs.
  - Specific practice 1.2: Appraise the organization's processes.
  - Specific practice 1.3: Identify improvements to the processes.
- Specific goal 2: Improvements are planned and implemented, organizational process assets are deployed, and process-related experiences are incorporated into the organization's process assets.
  - Specific practice 2.1: Establish and maintain process action plans.
  - Specific practice 2.2: Implement process action plans.
  - Specific practice 2.3: Deploy organizational process assets.
  - Specific practice 2.4: Incorporate process-related work products, measures, and improvement information into organizational process assets.

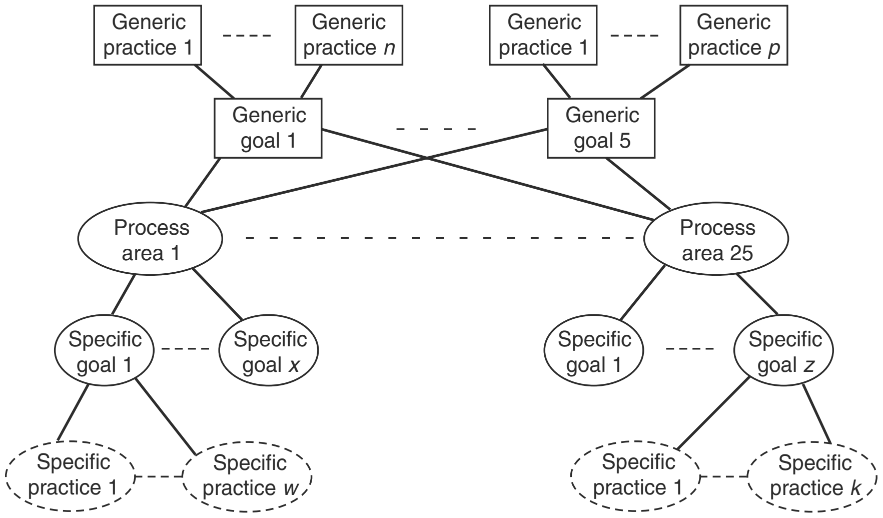

Figure 4.10
Goals and practices.

Consult the CMMI document, CMU/SEI–2002–TR–028, in the Suggested Readings for a complete list of specific goals and their respective specific practices for each of the 25 process areas.

In contrast to the specific goals, which are different for each process area, there are five generic goals that are applicable to all the 25 process areas (see Figure 4.10). The five generic goals are as follows:

- Generic goal 1: Achieve specific goals of the process area.
- Generic goal 2: Institutionalize managed process.
- Generic goal 3: Institutionalize defined process.
- Generic goal 4: Institutionalize quantitatively managed process.
- Generic goal 5: Institutionalize optimizing process.

These generic goals also map into the continuous representation's capability levels 1 through 5, respectively. Capability level 0, which is the incomplete level, has no generic goal associated with it.

Associated with each of the five generic goals are sets of generic practices. Since the goals are applicable to all the process areas, the set of generic practices are also applicable to the 25 process areas. There is one generic practice associated with generic goal 1. There are 10 generic practices associated with generic goal 2. Generic goal 3 has two generic practices. Two generic practices are associated with generic goal 4, and generic goal 5 has two generic practices. Since the generic goals are applicable to all the process areas, their respective generic practices are also applicable to all the process areas.

Continuous Representation Model

The continuous representation model, in which each process area is appraised at its own capability level, uses both the specific goals and the generic goals for assessing the process areas. An example of a profile of an organization's capability level by process areas is depicted in Figure 4.11. This profile not only provides an assessment but also serves as guidance for an organization to improve on the process areas that need improvements. The continuous representation model has several functions in an organization:

- Allows an organization to select the order of improvements that best meets that organization's needs and structure.
- Allows comparisons across different organizations on a process area by process area basis.
- Allows easier migration from and comparison to Electronic Institute Alliance International Standard (EIA/IS) 731 and International Organization for Standardization and International Electro-technical Commission (ISO/IEC) 15504.

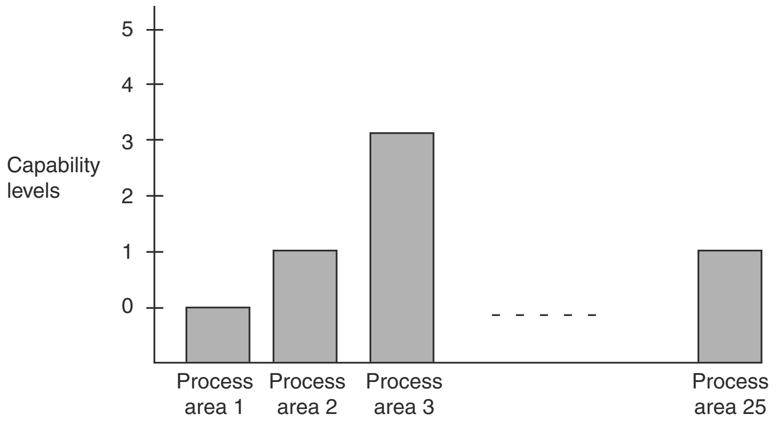

Figure 4.11
Capability level by process areas for continuous representation.

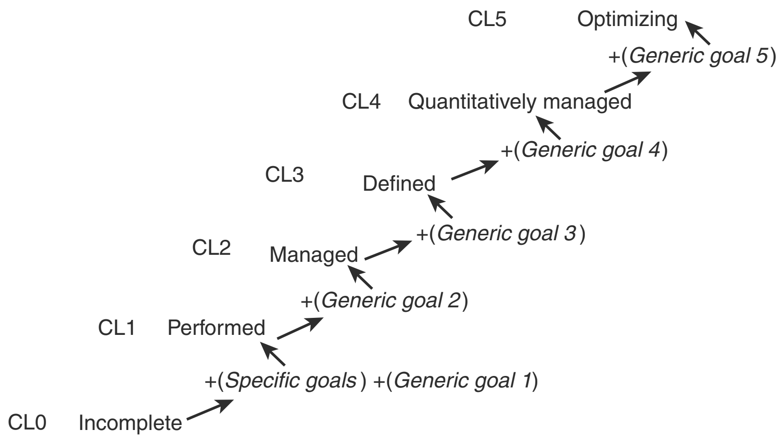

Figure 4.12
Achieving the capability levels by process area in the continuous representation model.

Each process area initially starts at capability level 0 (CL0), or the incomplete level. For any process area to move up from CL0 to the next level, performed level or CL1, two sets of activities must be completed.

- The specific goals for that process area must be achieved through completing all the associated specific practices for those specific goals.
- Generic goal 1 must be achieved through completion of its associated generic practices.

Once a process area reaches capability level 1, the performed level, subsequent levels are achieved by satisfying the subsequent generic goals and their respective generic practices (see Figure 4.12). The figure shows that in order for a process area to improve from CL*n* to CL*n*+1, generic goal *n*+1 must be satisfied.

Staged Representation Model

In a staged representation model, there are five maturity levels (ML's). The same 25 process areas are grouped into four of the five maturity levels. Maturity level 1, the initial level, has no process area associated with it. Essentially, ML1 is similar to CL0 of the continuous representation model. The organization achieves a maturity level by satisfying the set of process areas that are grouped under that maturity level. The groupings of process areas for the staged representation model's maturity levels are as follows:

ML5

- Organizational innovation and deployment
- Causal analysis and resolution

ML4

- Organizational process performance
- Quantitative project management

ML3

- Requirements development
- Technical solution
- Product integration
- Verification
- Validation
- Organizational process focus
- Organizational process definition
- Organizational training
- Integrated project management
- Risk management
- Integrated teaming
- Integrated supplier management
- Decision analysis and resolution
- Organizational environment for integration

ML2

- Requirements management
- Project planning
- Project monitoring and control

- Supplier agreement management
- Measurement and analysis
- Process and product quality assurance
- Configuration management

ML1

- None

The maturity levels are sequential, with any maturity level *n* being built upon maturity level $n - 1$. The staged representation model provides an organization a single maturity level appraisal based on the set of process areas satisfied. For example, an organization is assessed as maturity level 2 if all seven process areas grouped under ML2 are satisfied. The staged representation provides the following for an organization:

- A sequence of improvements of process areas by maturity levels
- The capacity to compare across organizations by maturity levels
- Easy migration from the earlier software CMM model

The rule for a process area to be considered satisfied in a staged representation model is similar to that of the continuous representation model. There are, however, some subtle differences. A process area that is grouped at ML2, managed level, would need to satisfy all its specific goals and associated specific practices along with generic goal 2 and its associated generic practices. For process areas that are grouped in ML3, defined level, those process areas need to satisfy all the specific goals and specific practices along with generic goal 2 and the associated generic practices. Each process area listed under ML4, quantitatively managed level, would need to satisfy all their specific goals and specific practices along with generic goal 3 and associated generic practices. Similar to ML4, each process area in ML5, optimizing level, would need to satisfy all its specific goals and specific practices along with generic goal 3 and generic practices. Note that in satisfying the staged maturity levels, generic goals 1, 4, and 5 do not play a part in the scheme.

4.6 Process Definition and Communication

We have discussed several traditional software development processes. While they serve as good models, it is very likely that they will need some modification to fit a specific organization. Depending on the goal(s) of the software project, a slightly different set of activities may be needed or emphasized. As Osterweil (1987) observed, the software development process is just a vehicle for carrying out those activities. Thus, specifying the process model is similar to constructing a software system itself. A process model or specification is an abstract representation of the actual process. It is important that this “modified” process be well defined and communicated to all participants so that the project can be carried out smoothly.

A software process specification is composed of two basic parts:

- The activities to be included in the software project
- The order in which these activities should be performed

These two main components are further expanded and refined to include the following set of items:

- Activities: Detailed descriptions of each of the activities included in the process
- Control: Necessary entry and exit criteria for each activity, in addition to the order in which each should be performed
- Artifacts: The resulting output from each of the activities
- Resources: The people who perform the activities
- Tools: The tools that may be used to enhance the performance of the activity

A software process definition for development and support projects needs to include all of the preceding information in differing degrees of detail. The modified process definition for a specific organization needs to describe the activities to be performed, specify the controlled conditions of the entry and exit criteria, and define the order in which these activities must be performed. It is necessary to identify and define the resulting artifacts, including null situations, from each of the activities. A software project is usually carried out by several people—each having different skills and experience. The number of people needed, their individual skill levels, and the experience level of each, must be specified. Finally, any tools that can enhance the performance of the activities should be specified.

It is difficult and tedious to define all of the preceding activities and related items. Thus, a team may decide to place an emphasis on the parts that are most relevant to each project. On the other hand, in order to provide some flexibility for a very experienced team that has worked on similar projects before, they may choose to define all five parts at a high level for the purpose of management overview. Note that specifying the software process to the most detailed level would be nearly equivalent to performing the detailed design and programming of the software process itself.

4.7 Summary

In earlier chapters we alluded to the importance of having a process or a set of processes to guide the software developers in large development and support projects. In this chapter we traced through three traditional process models:

- Waterfall
- Incremental
- Spiral

A more modern process model, the Rational Unified Process (RUP), was introduced. The emphasis here is on the need to have well-defined criteria for both entrance to and exit from activities in a process model.

The Software Engineering Institute at Carnegie Mellon has been a driving force in the process modeling and process assessment arena. Its first software process model, the Capability Maturity Model (CMM) is now well known among software industry practitioners. In recent years, the improved model, Capability Maturity Model Integrated (CMMI), is gaining momentum. CMMI's continuous representation model allows an organization to assess the capability level of its process areas separately while the staged model allows an organization to assess the maturity level of the complete organization, much like the CMM model. It is very likely that a standard process needs to be modified and refined before it can be utilized by a software project.

In the next chapter, we will introduce the more recent processes and methodologies such as Agile and Extreme Programming.

4.8 Review Questions

1. Discuss one advantage and one disadvantage of the waterfall process.
2. What is the goal of a software process model?
3. What are the four quadrants in a spiral model? Trace the requirements set of activities through each quadrant.
4. What are the entry and exit criteria to a process?
5. What motivated software engineers to move from the waterfall model to the incremental or spiral model?
6. What are the major concepts that drove the Rational Unified Process framework?
7. What are the four phases of Rational Unified Process?
8. List all of the key processes addressed by SEI's CMM model. Which ones are required for maturity level 2?
9. How many process areas, in total, are included in SEI's Software CMMI? List those that fall into the engineering category and the support category.

4.9 Exercises

1. Look again at the simple process model in Figure 4.1. What development activity would you choose to add first to that process and why?
2. What is the difference between the multiple component incremental model and the multiple release incremental model?
3. Discuss the four phases of Rational Unified Process and their relationship to the development activities such as requirements analysis, design, and testing.
4. Give two entry criteria examples and discuss their importance.
5. Give two exit criteria examples and discuss their importance.
6. Get on the Internet at www.sei.cmu.edu and search for SEI's vision and mission. Do you believe we need such an organization and why?
7. List the process areas that are required for staged maturity level 2 of CMMI. How do these differ from those of maturity level 2 in CMM?

8. Discuss the two representation models in CMMI. What do these two models assess?
9. In the continuous representation model, discuss how a process area moves up (or improves) from CL2 to CL3.

4.10 Suggested Readings

D. M. Ahern, A. Closure, and R. Turner, *CMMI Distilled—A Practical Introduction to Integrated Process Improvement*, 2nd ed. (Reading, MA: Addison-Wesley, 2004).

B. Boehm, "A Spiral Model for Software Development and Enhancement," *Computer* 21, no. 5 (May 1988): 61–72.

F. P. Brooks, *The Mythical Man Month* (Reading, MA: Addison-Wesley, 1975).

Capability Maturity Model Integration (CMMI) Version 1.1, CMMI for Software Engineering, CMU/SEI–2002–TR–028, August 2002.

M. Cusumano, A. MacCormack, C. F. Kemerer, and Crandall, B., "Software Development Worldwide: The State of the Practices," *IEEE Software* 20, no. 6 (November–December 2003): 28–34.

K. E. Emam and N. H. Madhavji, *Elements of Software Process Assessment and Improvement* (Los Alamitos, CA: IEEE Computer Society, 1999).

M. Fowler and K. Scott, *UML Distilled*, 2nd ed. (Reading, MA: Addison-Wesley, 1999).

T. Gilb, *Principles of Software Engineering Management* (Reading, MA: Addison-Wesley Longman, 1989).

—"Rule-Based Design Reviews," *Software Quality Professional* 7, no. 1 (December 2004): 4–13.

T. Glib and K. Gilb, *Evolutionary Project Management and Product Development*. Unfinished book manuscript at http://www.result-planning.com (October 2004).

F. Guerrero and Y. Eterovic, Adapting the SW-CMM in a Small IT Organization," *IEEE Software* (July/August 2004): 29–35.

W. S. Humphrey, *Managing the Software Process* (Reading, MA: Addison-Wesley, 1989).

—*A Discipline for Software Engineering* (Reading, MA: Addison-Wesley, 1995).

—*Introduction to the Personal Software Process* (Reading, MA: Addison-Wesley, 1997).

International Standards Organization (ISO), www.iso.org, 2004.

I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process* (Reading, MA: Addison-Wesley Longman, 1999).

P. Kruchten, *The Rational Unified Process*, 3rd ed. (Reading, MA: Addison-Wesley, 2003).

A. MacCormack, "Product-Development Practices That Work: How Internet Companies Build Software," *MIT Sloan Management Review* (Winter 2001): 75–83.

L. Osterweil, "Software Processes Are Software Too," Proceedings of 9th International Conference on Software Engineering, (April 1987): 2–13.

M. C. Paulk, et al., "Capability Maturity Model for Software, Version 1.1," Software Engineering Institute, CMU/SEI–93-TR-24, DTIC Number ADA263404 (February 1993).

R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. (New York: McGraw-Hill, 2005).

W. W. Royce, "Managing the Development of Large-Scale Software Systems," *Proceedings of IEEE WESCON*, August 1970.

J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual* (Reading, MA: Addison-Wesley, 1998).

Software Engineering Institute (SEI), www.sei.cmu.edu, 2004

J. Wood and D. Silver, *Joint Application Development*, 2nd ed. (New York: John Wiley, 1995).