One

THE SOFTWARE PROCESS

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the process that provides a framework for software engineering practice. These questions are addressed in the chapters that follow:

- What is a software process?
- What are the generic framework activities that are present in every software process?
- How are processes modeled and what are process patterns?
- What are prescriptive process models and what are their strengths and weaknesses?
- What characteristics of incremental models make them amenable to modern software projects?
- What is the unified process?
- Why is "agility" a watchword in modern software engineering work?
- What is agile software development and how does it differ from more traditional process models?

Once these questions are answered you'll be better prepared to understand the context in which software engineering practice is applied.

CHAPTER

2 A GENERIC VIEW OF PROCESS

KEY CONCEPTS

CMMI
ISO 9001: 2000
process assessment
process framework
process patterns
process technology
PSP
task set
TSP
umbrella activities

In a fascinating book that provides an economist's view of software and software engineering, Howard Baetjer, Jr. [BAE98] comments on the software process:

> Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.

Indeed, building computer software is an iterative learning process, and the outcome, something that Baetjer would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

QUICK LOOK

What is it? When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a software process.

Who does it? Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and documentation that are appropriate for the project team and the product that is to be produced.

What are the steps? At a detailed level, the process that you adopt depends on the software that you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a Web site.

What is the work product? From the point of view of a software engineer, the work products are the programs, documents, and data that are produced as a consequence of the activities and tasks defined by the process.

How do I ensure that I've done it right? There are a number of software process assessment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.

But what exactly is a software process from a technical point of view? Within the context of this book, we define a *software process* as a framework for the tasks that are required to build high-quality software. Is *process* synonymous with software engineering? The answer is yes and no. A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

More important, software engineering is performed by creative, knowledgeable people who should adapt a mature software process that is appropriate for the products they build and the demands of their marketplace.

2.1 Software Engineering—A Layered Technology

Although hundreds of authors have developed personal definitions of *software engineering,* a definition proposed by Fritz Bauer [NAU69] at the seminal conference on the subject still serves as a basis for discussion:

> [Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of an effective process. And yet, Bauer's definition provides us with a baseline. What are the "sound engineering principles" that can be applied to computer software development? How do we "economically" build software so that it is "reliable"? What is required to create computer programs that work "efficiently" on not one but many different "real machines"? These are the questions that continue to challenge software engineers.

> **"More than a discipline or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem."**
>
> **Scott Whitmire**

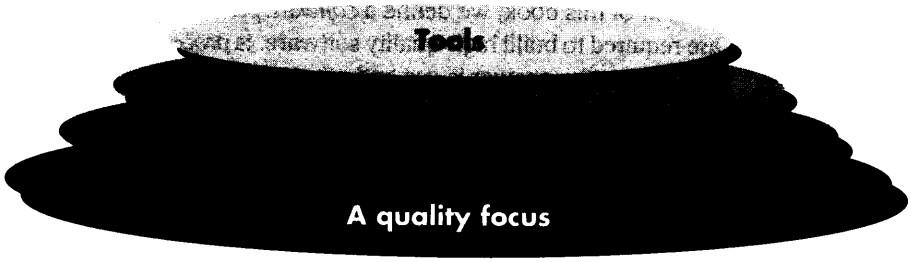
The IEEE [IEE93] has developed a more comprehensive definition when it states:

How do we define *software engineering?*

> Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

And yet, what is "systematic, disciplined" and "quantifiable" to one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

FIGURE 2.1 Software engineering layers



Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total Quality Management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a *quality focus*.

KEY POINT Software engineering encompasses a process, methods, and tools.

The foundation for software engineering is the *process* layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework [PAU93] that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

WebRef *Cross Talk* is a journal that provides pragmatic information on process, methods, and tools. It can be found at www.stsc.hill.af.mil.

Software engineering *tools* provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering,* is established.

2.2 A PROCESS FRAMEWORK

A *process framework* establishes the foundation for a complete software process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process.

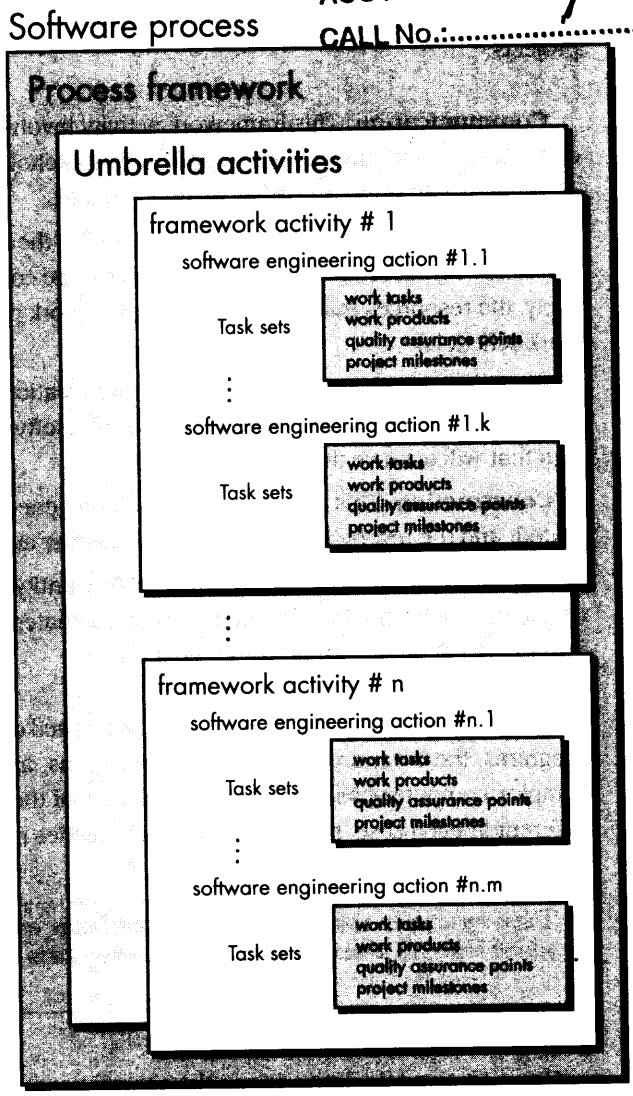


FIGURE 2.2 A software process framework

Referring to Figure 2.2, each framework activity is populated by a set of *software engineering actions*—a collection of related tasks that produces a major software engineering work product (e.g., *design* is a software engineering action). Each action is populated with individual *work tasks* that accomplish some part of the work implied by the action.

"A process defines who is doing *what*, *when*, and *how* to reach a certain goal."
Ivar Jacobson, Grady Booch, and James Rumbaugh

The following *generic process framework* (used as a basis for the description of process models in subsequent chapters) is applicable to the vast majority of software projects:

What are the five generic process framework activities?

Communication. This framework activity involves heavy communication and collaboration with the customer (and other stakeholders[1]) and encompasses requirements gathering and other related activities.

Planning. This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. This activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

Construction. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

> "Einstein argued that there must be a simplified explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity."
>
> **Fred Brooks**

Using an example derived from the generic process framework, the *modeling* activity is composed of two software engineering actions—*analysis* and *design*. Analysis[2] encompasses a set of work tasks (e.g., requirements gathering, elaboration, negotiation, specification, and validation) that lead to the creation of the analysis model (and/or requirements specification). Design encompasses work tasks (data

1 A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end-users, software engineers, support people, and so forth. Rob Thomsett jokes that "a stakeholder is a person holding a large and sharp stake. . . . If you don't look after your stakeholders, you know where the stake will end up."

2 Analysis is discussed at length in Chapters 7 and 8.

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics.

design, architectural design, interface design, and component-level design) that create a design model (and/or a design specification).[3]

Referring again to Figure 2.2, each software engineering action is represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. The task set that best accommodates the needs of the project and the characteristics of the team is chosen. This implies that a software engineering action (e.g., design) can be adapted to the specific needs of the software project and the characteristics of the project team.

INFO

Task Set

A *task set* defines the actual work to be done to accomplish the objectives of a software engineering action. For example, "requirements gathering" is an important software engineering action that occurs during the **communication** activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated requirements gathering meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve requirements gathering, but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each process activity and software engineering action and still maintain quality and agility.

The framework described in the generic view of software engineering is complemented by a number of *umbrella activities*. Typical activities in this category include:

Software project tracking and control—allows the software team to assess progress against the project plan and take necessary action to maintain schedule.

---

3 It should be noted that "modeling" must be interpreted somewhat differently when the maintenance of existing software is conducted. In some cases, analysis and design modeling do occur, but in other maintenance situations, modeling may be used to help understand the legacy software as well as to represent additions or modifications to it.

KEY POINT
Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

Risk management—assesses risks that may effect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Formal technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets customers' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Umbrella activities are applied throughout the software process and are discussed in detail later in this book.

KEY POINT
Software process adaptation is essential for project success.

All process models can be characterized within the process framework shown in Figure 2.2. Intelligent application of any software process model must recognize that adaptation (to the problem, project, team, and organizational culture) is essential for success. But process models do differ fundamentally in:

How do process models differ from one another?

- The overall flow of activities and tasks and the interdependencies among activities and tasks.
- The degree to which work tasks are defined within each framework activity.
- The degree to which work products are identified and required.
- The manner which quality assurance activities are applied.
- The manner in which project tracking and control activities are applied.
- The overall degree of detail and rigor with which the process is described.
- The degree to which customer and other stakeholders are involved with the project.
- The level of autonomy given to the software project team.
- The degree to which team organization and roles are prescribed.

"[And a recipe is only a theme which an intelligent cook can play each time with a variation."
[illegible]

Process models that stress detailed definition, identification, and application of process activities and tasks have been applied within the software engineering community for

the past 30 years. When these *prescriptive process models* are applied, the intent is to improve system quality, to make projects more manageable, to make delivery dates and costs more predictable, and to guide teams of software engineers as they perform the work required to build a system. Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy associated with building computer-based systems and inadvertently create difficulty for developers and customers.

What characterizes an "agile" process?

Process models that emphasize project agility and follow a set of principles[4] that lead to a more informal (but, proponents argue, no less effective) approach to software process have been proposed in recent years. These *agile process models* emphasize maneuverability and adaptability. They are appropriate for many types of projects and are particularly useful when Web applications are engineered.

Which software process philosophy is best? This question has spawned emotional debate among software engineers and will be addressed in Chapter 4. For now, it is important to note that these two process philosophies have a common goal—to create high-quality software that meets the customer's needs—but different approaches.

2.3 The Capability Maturity Model Integration (CMMI)

The Software Engineering Institute (SEI) has developed a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity. To achieve these capabilities, the SEI contends that an organization should develop a process model (Figure 2.2) that conforms to *The Capability Maturity Model Integration* (CMMI) guidelines [CMM02].

WebRef
Complete information on the CMMI can be obtained at http://www.sei.cmu.edu/cmmi/.

The CMMI represents a process meta-model in two different ways: (1) as a *continuous* model and (2) as a *staged* model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure 2.3. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

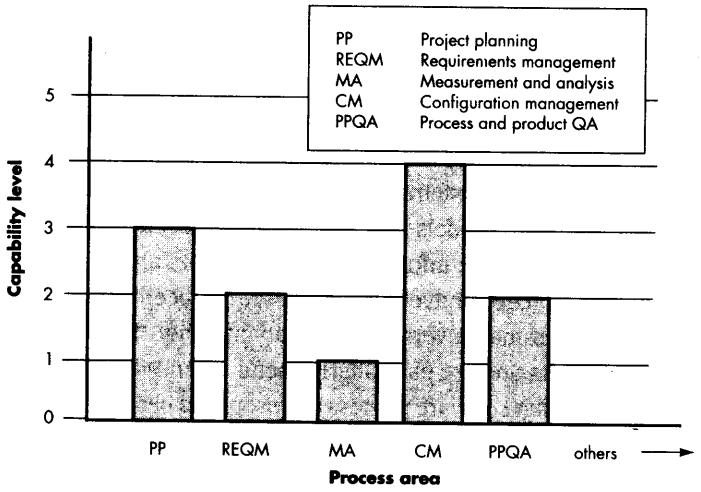
Level 0: Incomplete. The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done;

4 Agile models and the principles that guide them are discussed in Chapter 4.

FIGURE 2.3
CMMI process area capability profile [PHI02]



stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM02].

Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" [CMM02].

Level 4: Quantitatively managed. All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM02].

Level 5: Optimized. All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration" [CMM02].

ADVICE

Every organization should strive to achieve the intent of the CMMI. However, implementing every aspect of the model may be overkill in some situations.

> **"Much of the software crisis is self-inflicted, as when a CIO says, "I'd rather have it wrong than have it late. We can always fix it later."**
>
> **Mark Paulk**

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

For example, **project planning** is one of eight process areas defined by the CMMI for the "project management" category.[5] The specific goals (SG) and the associated specific practices (SP) defined for project planning are [CMM02]:

SG 1 Establish estimates

SP 1.1-1 Estimate the scope of the project

SP 1.2-1 Establish estimates of work product and task attributes

SP 1.3-1 Define project life cycle

SP 1.4-1 Determine estimates of effort and cost

SG 2 Develop a Project Plan

WebRef
Complete information as well as a downloadable version of the CMMI can be obtained at www.sei.cmu.edu/cmmi/.

SP 2.1-1 Establish the budget and schedule

SP 2.2-1 Identify project risks

SP 2.3-1 Plan for data management

SP 2.4-1 Plan for project resources

SP 2.5-1 Plan for needed knowledge and skills

SP 2.6-1 Plan stakeholder involvement

SP 2.7-1 Establish the project plan

SG 3 Obtain commitment to the plan

SP 3.1-1 Review plans that affect the project

SP 3.2-1 Reconcile work and resource levels

SP 3.3-1 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, the generic goals (GG) and practices (GP) for the project planning process area are [CMM02]:

GG 1 Achieve specific goals

GP 1.1 Perform base practices

GG 2 Institutionalize a managed process

GP 2.1 Establish an organizational policy

GP 2.2 Plan the process

GP 2.3 Provide resources

---

5 Other process areas defined for "project management" include: project monitoring and control, supplier agreement management, integrated project management for IPPD, risk management, integrated teaming, integrated supplier management, and quantitative project management.

GP 2.4 Assign responsibility

GP 2.5 Train people

GP 2.6 Manage configurations

GP 2.7 Identify and involve relevant stakeholders

GP 2.8 Monitor and control the process

GP 2.9 Objectively evaluate adherence

GP 2.10 Review status with higher level management

GG 3 Institutionalize a defined process

GP 3.1 Establish a defined process

GP 3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

GP 4.1 Establish quantitative objectives for the process

GP 4.2 Stabilize subprocess performance

GG 5 Institutionalize an optimizing process

GP 5.1 Ensure continuous process improvement

GP 5.2 Correct root causes of problems

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity levels and process areas is shown in Figure 2.4.

INFO

The CMMI—Should We or Shouldn't We?

The CMMI is a process meta-model. It defines (in over 700 pages) the process characteristics that should exist if an organization wants to establish a software process that is complete. The question that has been debated for well over a decade is: Is the CMMI overkill? Like most things in life (and in software), the answer is not a simple yes or no.

The spirit of the CMMI should always be adopted. At the risk of oversimplification, it argues that software development must be taken seriously—it must be planned thoroughly; it must be controlled uniformly; it must be tracked accurately; and it must be conducted professionally. It must focus on the needs of project stakeholders, the skills of the software engineers, and the quality of the end product. No one would argue with these ideas.

The detailed requirements of the CMMI should be seriously considered if an organization builds large complex systems that involve dozens or hundreds of people over many months or years. It may be that the CMMI is just right in such situations, if the organizational culture is amenable to standard process models and management is committed to making it a success. However, in other situations, the CMMI may simply be too much for an organization to successfully assimilate. Does this mean that the CMMI is bad or overly bureaucratic or old fashioned? No, it does not. It simply means that what

is right for one company culture may not be right for another.

The CMMI is a significant achievement in software engineering. It provides a comprehensive discussion of the activities and actions that should be present when an organization builds computer software. Even if a software organization chooses not to adopt its details, every software team should embrace its spirit and gain insight from its discussion of software engineering process and practice.

FIGURE 2.4
Process areas required to achieve a maturity level

Level	Focus	Process Areas
Optimizing	***Continuous process improvement***	**Organizational Innovation and Deployment** **Causal Analysis and Resolution**
Quantitatively managed	***Quantitative management***	**Organizational Process Performance** **Quantitative Project Management**
Defined	***Process standardization***	**Requirements Development** **Technical Solution** **Product Integration** **Verification** **Validation** **Organizational Process Focus** **Organizational Process Definition** **Organizational Training** **Integrated Project Management** **Integrated Supplier Management** **Risk Management** **Decision Analysis and Resolution** **Organizational Environment for Integration** **Integrated Teaming**
Managed	***Basic project management***	**Requirements Management** **Project Planning** **Project Monitoring and Control** **Supplier Agreement Management** **Measurement and Analysis** **Process and Product Quality Assurance** **Configuration Management**
Performed		

2.4 PROCESS PATTERNS

What is a process pattern?

The software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and/or related behaviors [AMB98] required to develop computer software. Stated in more general terms, a *process pattern* provides us with a template—a consistent method for describing an important characteristic of the software process. By combining patterns, a software team can construct a process that best meets the needs of a project.

> "The repetition of patterns is quite a different thing than the repetition of parts. Indeed, the different parts will be unique because the patterns are the same."
>
> **Christopher Alexander**

Patterns can be defined at any level of abstraction.[6] In some cases, a pattern might be used to describe a complete process (e.g., prototyping). In other situations, patterns can be used to describe an important framework activity (e.g., planning) or a task within a framework activity (e.g., project-estimating).

Ambler [AMB98] has proposed the following template for describing a process pattern:

A pattern template provides a consistent means for describing a pattern.

Pattern Name. The pattern is given a meaningful name that describes its function within the software process (e.g., **customer-communication**).

Intent. The objective of the pattern is described briefly. For example, the intent of **customer-communication** is "to establish a collaborative relationship with the customer in an effort to define project scope, business requirements, and other project constraints." The intent might be further expanded with additional explanatory text and appropriate diagrams if required.

Type. The pattern type is specified. Ambler [AMB98] suggests three types:

- *Task patterns* define a software engineering action or work task that is part of the process and relevant to successful software engineering practice (e.g., **requirements gathering** is a task pattern).
- *Stage patterns* define a framework activity for the process. Since a framework activity encompasses multiple work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage (framework activity). An example of a stage pattern might be **communication**. This pattern would incorporate the task pattern **requirements gathering** and others.
- *Phase patterns* define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be a **spiral model or prototyping**.[7]

Initial Context. The conditions under which the pattern applies are described. Prior to the initiation of the pattern, we ask (1) what organizational or team-related activities have already occurred, (2) what is the entry state for the process, and (3) what software engineering information or project information already exists.

---

6 Patterns are applicable to many software engineering activities. Analysis, design, and testing patterns are discussed in Chapters 7, 9, 10, 12, and 14. Patterns and "antipatterns" for project management activities are discussed in Part 4 of this book.

7 These phase patterns are discussed in Chapter 3.

For example, the **planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns (specified) for the **customer-communication** pattern has occurred; and (3) project scope, basic business requirements, and project constraints are known.

Problem. The problem to be solved by the pattern is described. For example, the problem to be solved by **customer-communication** might be described in the following manner: *Communication between the developer and the customer is often inadequate because an effective format for eliciting information is not established, a useful mechanism for recording it is not created, and meaningful review is not conducted.*

Solution. The implementation of the pattern is described. This section discusses how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. The conditions that will result once the pattern has been successfully implemented are described. Upon completion of the pattern we ask (1) what organizational or team-related activities must have occurred, (2) what is the exit state for the process, and (3) what software engineering information or project information has been developed.

Related Patterns. A list of all process patterns that are directly related to this one are provided—as a hierarchy or in some other diagrammatic form. For example, the stage pattern **communication** encompasses the task patterns **project-team assembly, collaborative-guideline definition, scope-isolation, requirements gathering, constraint-description, and mini-spec/model creation.**

Known Uses/Examples. The specific instances in which the pattern is applicable are indicated. For example, **communication** is mandatory at the beginning of every software project; it is recommended throughout the software project; and it is mandatory once the **deployment** activity is underway.

WebRef

Comprehensive resources on process patterns can be found at www.ambysoft.com/processPatternsPage.html.

Process patterns provide an effective mechanism for describing any software process. The patterns enable a software engineering organization to develop a hierarchical process description that begins at a high-level of abstraction (a phase pattern). The description is refined into a set of stage patterns that describe framework activities and then further refined in hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

INFO

An Example Process Pattern

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done, but are unsure of specific software requirements.

Pattern name. Prototyping.

Intent. The objective of the pattern is to build a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

Type. Phase pattern.

Initial context. The following conditions must be met prior to the initiation of this pattern: (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding problem to be solved has been identified by stakeholders; (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem. Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solution. A description of the prototyping process is presented here. See Chapter 3 for details.

Resulting context. A software prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

Related patterns. The following patterns are related to this pattern: **customer-communication; iterative design; iterative development, customer assessment; requirement extraction.**

Known uses/examples. Prototyping is recommended when requirements are uncertain.

2.5 PROCESS ASSESSMENT

KEY POINT

Assessment attempts to understand the current state of the software process with the intent of improving it.

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapter 26). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.[8] The relationship between the software process and the methods applied for assessment and improvement is shown in Figure 2.5. A number of different approaches to *software process assessment* have been proposed over the past few decades:

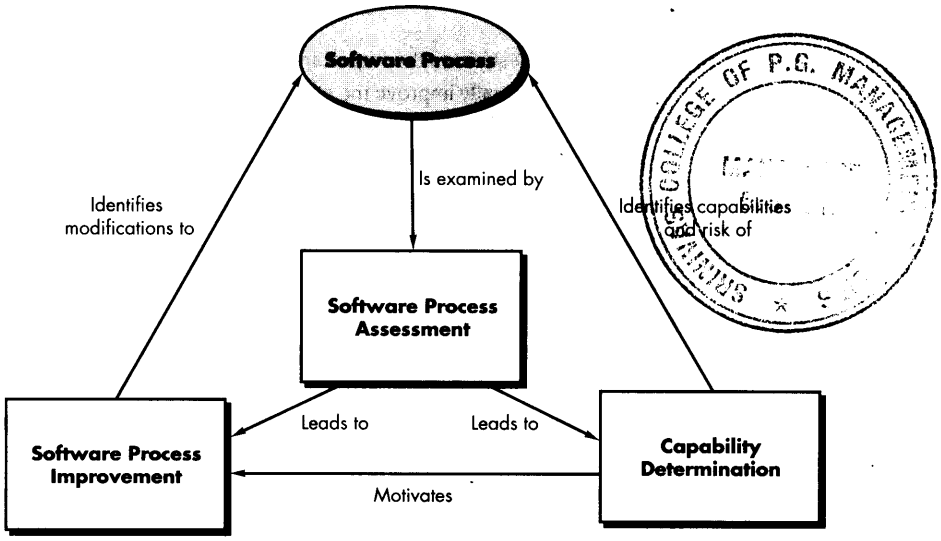
What formal techniques are available for assessing the software process?

Standard CMMI Assessment Method for Process Improvement (SCAMPI) provides a five-step process assessment model that incorporates initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI (Section 2.3) as the basis for assessment [SEI00].

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) provides a diagnostic technique for assessing the relative maturity of a software or-

8 The SEI's CMMI [CMM02] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

FIGURE 2.5



ganization, using the SEI CMM (a precursor to the CMMI discussed in Section 2.3) as the basis for the assessment [DUN01].

SPICE (ISO/IEC15504) standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [SPI99].

ISO 9001:2000 for Software is a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

Because ISO 9001:2000 is widely used on an international scale, we examine it briefly in the paragraphs that follow.

"Software organizations have exhibited significant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

The International Organization for Standardization (ISO) has developed the ISO 9001:2000 standard [ISO00] to define the requirements for a quality management system (Chapter 26) that will serve to produce higher quality products and thereby improve customer satisfaction.[9]

9 Software quality assurance (SQA), an important element of quality management, has been defined as a umbrella activity that is applied across the entire process framework. It is discussed in detail in Chapter 26.

The underlying strategy suggested by ISO 9001:2000 is described in the following manner [ISO01]:

> ISO 9001:2000 stresses the importance for an organization to identify, implement, manage, and continually improve the effectiveness of the processes that are necessary for the quality management system, and to manage the interactions of these processes in order to achieve the organization's objectives . . .

WebRef
An excellent summary of ISO 9001:2000 can be found at http://praxiom.com/iso-9001.htm.

ISO 9001:2000 has adopted a "plan-do-check-act" cycle that is applied to the quality management elements of a software project. Within a software context, "plan" establishes the process objectives, activities, and tasks necessary to achieve high-quality software and resultant customer satisfaction. "Do" implements the software process (including both framework and umbrella activities). "Check" monitors and measures the process to ensure that all requirements established for quality management have been achieved. "Act" initiates software process improvement activities that continually work to improve the process.

For a detailed discussion of ISO 9001:2000 interested readers should see the ISO standards themselves or [CIA01], [KET01], or [MON01] for comprehensive information.

2.6 Personal and Team Process Models

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization. Alternatively, the team itself would create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey ([HUM97] and [HUM00]) argues that it is possible to create a "personal software process" and/or a "team software process." Both require hard work, training and coordination, but both are achievable.[10]

> "A person who is successful has simply formed the habit of doing things that unsuccessful people will not do."
> Dexter Yager

WebRef
A wide array of resources for PSP can be found at www.ipd.uka.de/PSP/.

2.6.1 Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc, may change on a daily basis, may not be efficient, effective or even successful, but a process does exist. Watts Humphrey [HUM97] suggests that in

10 It's worth noting that the proponents of agile software development (Chapter 4) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The *personal software process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition, PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.

The PSP process model defines five framework activities: planning, high-level design, high-level design review, development, and postmortem.

What framework activities are used during PSP?

Planning. This activity isolates requirements and, based on these, develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods (Chapter 26) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

KEY POINT

PSP emphasizes the need to record and analyze the types of errors you make, so you can develop strategies to eliminate them.

PSP stresses the need for each software engineer to identify errors early and, as important, to understand the types of errors that he is likely to make. This is accomplished through a rigorous assessment activity performed on all work products produced by the software engineer.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [HUM96], the resulting improvement in software engineering productivity and software quality are significant [FER97]. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people.

Can PSP be used as an effective software process at a personal level? The answer is an unequivocal yes. But even if PSP is not adopted in its entirety, many of the personal process improvement concepts that it introduces are well worth learning.

2.6.2 Team Software Process (TSP)

WebRef
Information on building high-performance teams using TSP and PSP can be obtained at www.sei.cmu.edu/tsp/.

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a *team software process* (TSP). The goal of TSP is to build a "self-directed" project team that organizes itself to produce high-quality software. Humphrey [HUM98] defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

To form a self-directed team, you must collaborate well internally and communicate well externally.

A self-directed team has a consistent understanding of its overall goals and objectives. It defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

> "Finding good players is easy. Getting them to play as a team is another story."
>
> Casey Stengel

TSP scripts define elements of the team process and activities that occur within the process.

TSP defines the following framework activities: launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. *Scripts* define specific process activities (i.e., project launch, design, implementation, integration and testing, and postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, and unit test) that are part of the team process. To illustrate, consider the initial process activity—*project launch.*

Each project is "launched" using a sequence of tasks (defined as a script) that enables the team to establish a solid basis for starting the project: The following *launch script* (outline only) is recommended [HUM00]:

WebRef
Information on the software process dashboard—a PSP and TSP support tool—can be found at **processdash.sourceforge.net.**

- Review project objectives with management and agree on and document team goals.
- Establish team roles.
- Define the team's development process.
- Make a quality plan and set quality targets.
- Plan for the needed support facilities.
- Produce an overall development strategy.
- Make a development plan for the entire project.
- Make detailed plans for each engineer for the next phase.
- Merge the individual plans into a team plan.
- Rebalance team workload to achieve a minimum overall schedule.
- Assess project risks and assign tracking responsibility for each key risk.

It should be noted that the launch activity can be applied prior to each TSP framework activity noted earlier. This accommodates the iterative nature of many projects and allows the team to adapt to changing customer needs and lessons learned from previous activities.

TSP recognizes that the best software teams are self-directed. Team members set project objectives, adapt the process to meet their needs, have control over schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

2.7 Process Technology

The generic process models discussed in the preceding sections must be adapted for use by a software project team. To accomplish this, *process technology tools* have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality [NEG99].

Process technology tools allow a software organization to build an automated model of the common process framework, task sets, and umbrella activities discussed in Section 2.2. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other computer-aided software engineering tools that are appropriate for a particular work task.

SOFTWARE TOOLS

Process Modeling Tools

Objective: If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* or *process management tools*) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the actions and work tasks that are required to perform it. Process modeling tools provide links to other tools that provide support to defined process activities.

Mechanics: Tools in this category allow a team to define the elements of a unique process model (actions, tasks, work products, QA points), provide detailed guidance on the content or description of each process element, and then manage the process as it is conducted. In some cases, the process technology tools incorporate standard project management tasks such as estimating, scheduling, tracking and control.

Representative Tools:[11]

Igrafx Process Tools, distributed by Corel Corporation (www.igrafx.com/products/process), is a set of tools that enable a team to map, measure, and model the software process

Objexis Team Portal, developed by Objexis Corporation (www.objexis.com), provides full process workflow definition and control.

2.8 PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer. But an obsessive over-reliance on process is also dangerous. In a brief essay, Margaret Davis [DAV95] comments on the duality of product and process:

> About every ten years give or take five, the software community redefines "the problem" by shifting its focus from product issues to process issues. Thus, we have embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process) [followed by object-oriented methods, followed by agile software development].
>
> While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means

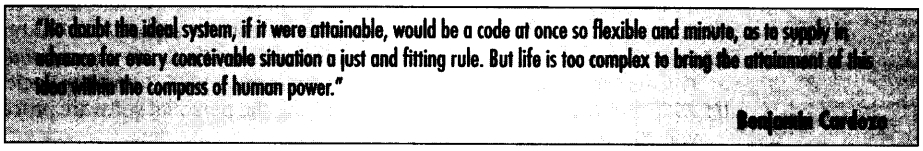
11 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

to perform the job, let alone perform it well. The swings also do not solve "the problem," for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.

There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920s when Louis de Broglie proposed it. I believe that the observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product. . . .

All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.



People derive as much (or more) satisfaction from the creative process as they do from the end-product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. A creative software professional should also derive as much satisfaction from the process as the end-product.

The work of software people will change in the years ahead. The duality of product and process is one important element in keeping creative people engaged as the transition from programming to software engineering is finalized.

2.9 SUMMARY

Software engineering is a discipline that integrates process, methods, and tools for the development of computer software. A number of different process models for software engineering have been proposed, but all define a set of framework activities, a collection of tasks that are conducted to accomplish each activity, work products produced

as a consequence of the tasks, and a set of umbrella activities that span the entire process. Process patterns can be used to define the characteristics of a process.

The Capability Maturity Model Integration (CMMI) is a comprehensive process meta-model that describes the specific goals, practices, and capabilities that should be present in a mature software process. SPICE and other standards define the requirements for conducting an assessment of software process, and the ISO 9001: 2000 standard examines quality management within a process.

Personal and team models for the software process have been proposed. Both emphasize measurement, planning, and self-direction as key ingredients for a successful software process.

The principles, concepts, and methods that enable us to perform the process that we call *software engineering* are considered throughout the remainder of this book.

REFERENCES

[AMB98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology,* Cambridge University Press/SIGS Books, 1998.

[BAE98] Baetjer, Jr., H., *Software as Capital,* IEEE Computer Society Press, 1998, p. 85.

[CIA01] Cianfrani, C., et al., *ISO 9001: 2000 Explained,* American Society of Quality, 2001.

[CMM02] *Capability Maturity Model Integration (CMMI),* Version 1.1, Software Engineering Institute, March 2002, available at http://www.sei.cmu.edu/cmmi/.

[DAV95] Davis, M., "Process and Product: Dichotomy or Duality," *Software Engineering Notes,* ACM Press, vol. 20, no. 2, April, 1995, pp. 17–18.

[DUN01] Dunaway, D., and S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1.2 Method Description,* Software Engineering Institute, 2001, can be downloaded at http://www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.

[ELE98] El Emam, K., J. Drouin, and W. Melo (eds.), *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination,* IEEE Computer Society Press, 1998.

[FER97] Ferguson, P., et al., "Results of applying the personal software process," *IEEE Computer,* vol. 30 , no. 5, May 1997, pp. 24–31.

[HUM96] Humphrey, W., "Using a Defined and Measured Personal Software Process," *IEEE Software,* vol. 13, no. 3, May/June 1996, pp. 77–88.

[HUM97] Humphrey, W., *Introduction to the Personal Software Process,* Addison-Wesley, 1997.

[HUM98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process," *Crosstalk,* April 1998. Available at http://www.stsc.hill.af.mil/ crosstalk/1998/apr/dimensions.asp

[HUM00] Humphrey, W., *Introduction to the Team Software Process,* Addison-Wesley, 2000.

[IEE93] *IEEE Standards Collection: Software Engineering,* IEEE Standard 610.12-1990, IEEE, 1993.

[ISO00] *ISO 9001:2000 Document Set,* International Organization for Standards, 2000, http://www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.

[ISO01] "Guidance on the Process Approach to Quality Management Systems," Document ISO/TC 176/SC 2/N544R, International Organization for Standards, May 2001.

[KET01] Ketola, J., and K. Roberts, *ISO 9001: 2000 in a Nutshell,* 2 ed., Paton Press, 2001.

[MON01] Monnich, H., Jr., and H. Monnich, ISO 9001: 2000 for Small- and Medium-Sized Businesses, American Society of Quality, 2001.

[NAU69] Naur, P., and B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee,* NATO, 1969.

[NEG99] Negele, H., "Modeling of Integrated Product Development Processes," *Proc. 9th Annual Symposium of INCOSE,* United Kingdom, 1999.

[PAU93] Paulk, M., et al., *Capability Maturity Model for Software,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[PHI02] Phillips, M., "CMMI V1.1 Tutorial," April 2002, available at http://www.sei. cmu.edu/cmmi/.

[SEI00] *SCAMPI, V1.0 Standard CMMI ® Assessment Method for Process Improvement: Method Description,* Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, downloadable from http://www.sei.cmu.edu/publications/documents/00.reports/00tr009. html.

[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction," Version 1.0, ISO/IEC JTC1, 1999.

Problems and Points to Ponder

2.1. Consider the framework activity *communication.* Develop a complete process pattern (this would be a stage pattern) using the template presented in Section 2.4.

2.2. What is the purpose of process assessment? Why has SPICE been developed as a standard for process assessment?

2.3. Download the CMMI documentation from the SEI Web site and select a process area other than project planning. Make a list of specific goals (SG) and the associated specific practices (SP) defined for the area you have chosen.

2.4. Try to develop a task set for the *communication* activity.

2.5. Research the CMMI in a bit more detail and discuss the pros and cons of both the continuous and staged CMMI models.

2.6. Describe a process framework in your own words. When we say that framework activities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.

2.7. Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities?

2.8. Is there ever a case when the generic activities of the software engineering process don't apply? If so, describe it.

2.9. In the introduction to this chapter, Baetjer notes: "The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]." List five questions that (a) designers should ask users; (b) users should ask designers; (c) users should ask themselves about the software product that is to be built; and (d) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

2.10. Figure 2.1 places the three software engineering layers on top of a layer entitled "a quality focus." This implies an organization-wide quality program such as Total Quality Management. Do a bit of research and develop an outline of the key tenets of a Total Quality Management program.

2.11. The use of "scripts" (a required mechanism in TSP) is not universally praised within the software community. Make a list of pros and cons regarding scripts and suggest at least two situations in which they would be useful and another two situations where they might provide less benefit.

2.12. Do some research on PSP and present a brief presentation that indicates the quantitative benefits of the process.

Further Readings and Information Sources

The current state of software engineering and the software process can best be determined from monthly publications such as *IEEE Software, Computer,* and *IEEE Transactions on Software Engineering.* Industry periodicals such as *Application Development Trends* and *Cutter IT Journal* often

contain articles on software engineering topics. The discipline is "summarized" every year in the *Proceeding of the International Conference on Software Engineering,* sponsored by the IEEE and ACM, and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes,* and *Annals of Software Engineering.* Thousands of Web pages are dedicated to software engineering and the software process.

Many books addressing the software process and software engineering have been published in recent years. Some present an overview of the entire process, while others delve into a few important topics to the exclusion of others. Among the more popular offerings (in addition to this book!) are:

Abran, A., and J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge,* IEEE, 2002.

Ahern, D., et al., *CMMI Distilled,* Addison-Wesley, 2001.

Chrisis, B., et al., *CMMI: Guidelines for Process Integration and Product Improvement,* Addison-Wesley 2003.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Hunter, R., and R. Thayer (eds), *Software Process Improvement,* IEEE-CS Press (Wiley), 2001.

Persse, J., *Implementing the Capability Maturity Model,* Wiley, 2001.

Pfleeger, S., *Software Engineering: Theory and Practice,* 2nd ed., Prentice-Hall, 2001.

Potter, N., and M. Sakry, *Making Process Improvement Work*, Addison-Wesley, 2002.

Sommerville, I., *Software Engineering,* 6th ed., Addison-Wesley, 2000.

On the lighter side, a book by Robert Glass (*Software Conflict,* Yourdon Press, 1991) presents amusing and controversial essays on software and the software engineering process. Yourdon (*Death March Projects,* Prentice-Hall, 1997) discusses what goes wrong when major software projects fail and how to avoid these mistakes.

Garmus (*Measuring the Software Process,* Prentice-Hall, 1995) and Florac and Carlton (*Measuring the Software Process,* Addison-Wesley, 1999) discuss the use of mea-surement as a means for statistically assessing the efficacy of any software process.

A wide variety of software engineering standards and procedures have been published over the past decade. The IEEE *Software Engineering Standards* contains many different standards that cover almost every important aspect of the technology. The ISO 9001:2000 document set provides guidance for software organizations that want to improve their quality management activities. Other software engineering standards can be obtained from the Department of Defense, the FAA, and other government and nonprofit agencies. Fairclough (*Software Engineering Guides,* Prentice-Hall, 1996) provides a detailed reference to software engineering standards produced by the European Space Agency (ESA).

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

3 PROCESS MODELS

KEY CONCEPTS

AOSD model
CBD model
concurrent development
evolutionaryprocess
formal methods
incremental process
prescriptive models
prototyping
RAD model
spiral model
Unified Process
waterfall model

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these conventional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective roadmap for software teams. However, software engineering work and the product that it produces remain on "the edge of chaos" [NOG00].

In an intriguing paper on the strange relationship between order and chaos in the software world, Nogueira and his colleagues [NOG00] state:

> The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise" [KAU95]. The edge of chaos can be visualized as an unstable, partially structured state. . . . It is unstable because it is constantly attracted to chaos or to absolute order.
>
> We have the tendency to think that order is the ideal state of nature. This could be a mistake. Research . . . supports the theory that operation away from equilibrium generates creativity, self-organized processes, and increasing returns [ROO96]. Absolute order means the absence of variability, which could be an advantage under unpredictable environments. Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder.

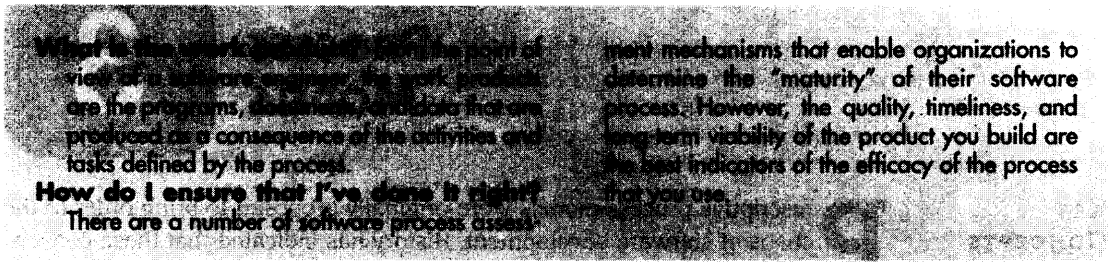
QUICK LOOK

What is it? Prescriptive process models define a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

Who does it? Software engineers and their managers adapt a prescriptive process model to their needs and then follow it. In addition, the people who have requested the software have a role to play as the process model is followed.

Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. Some have referred to prescriptive process models as "rigorous process models" because they often encompass the capabilities suggested by the CMMI (Chapter 2). However, every process model must be adapted so that it is used effectively for a specific software project.

What are the steps? The process guides a software team through a set of framework activities that are organized into a process flow that may be linear, incremental, or evolutionary. The terminology and details of each process model differ, but the generic framework activities remain reasonably consistent.



The philosophical implications of this argument are significant for software engineering. If prescriptive process models[1] strive for structure and order, are they inappropriate for a software world that thrives on change? Yet, if we reject conventional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In this chapter we examine the prescriptive process approach in which order and project consistency are dominant issues. In Chapter 4 we examine the agile process approach in which self-organization, collaboration, communication, and adaptability dominate the process philosophy.

3.1 Prescriptive Models

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

Every software engineering organization should describe a unique set of framework activities (Chapter 2) for the software process(es) it adopts. It should populate each framework activity with a set of software engineering actions, and define each action in terms of a task set that identifies the work (and work products) to be accomplished to meet the development goals. It should then adapt the resultant process model to accommodate the specific nature of each project, the people who will do the work, and the environment in which the work will be conducted. Regardless of the process model that is selected, software engineers have traditionally chosen a generic process framework that encompasses the following framework activities: communication, planning, modeling, construction, and deployment.

> "There are many ways of going forward, but only one way of standing still."
>
> **Franklin D. Roosevelt**

In the sections that follow, we examine a number of prescriptive software process models. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality as-

1 Prescriptive process models are often referred to as "conventional" process models.

Even though a process is prescriptive, don't assume that it is static. Prescriptive models should be adapted to the people, the problem, and the project.

surance, and change control mechanisms for each project. Each process model also prescribes a *workflow*—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities that have been described in Chapter 2, but each applies a different emphasis to these activities and defines a workflow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

3.2 The Waterfall Model

There are times when the requirements of a problem are reasonably well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well-defined and reasonably stable.

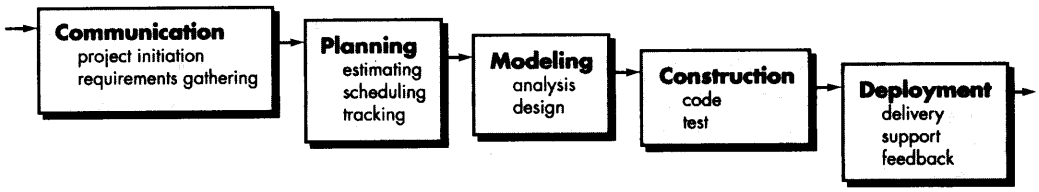
The *waterfall model,* sometimes called the *classic life cycle,* suggests a systematic, sequential approach[2] to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in on-going support of the completed software.

The waterfall model is the oldest paradigm for software engineering. However, over the past two decades, criticism of this process model has caused even ardent supporters to question its efficacy [HAN95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

Why does the waterfall model sometimes fail?

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

FIGURE 3.1 The waterfall model



2 Although the original waterfall model proposed by Winston Royce [ROY70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects, Bradac [BRA94] found that the linear nature of the waterfall model leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

3.3 Incremental Process Models

There are many situations in which initial software requirements are reasonably well-defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, a process model that is designed to produce the software in increments is chosen.

> "Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."
>
> Author unknown

3.3.1 The Incremental Model

KEY POINT
The incremental model delivers a series of releases, called *increments*, that provide progressively more functionality for the customer as each increment is delivered.

The *incremental model* combines elements of the waterfall model applied in an iterative fashion. Referring to Figure 3.2, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [MCD93]. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment may incorporate the prototyping paradigm discussed in Section 3.4.1.

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed, but many supplementary features (some

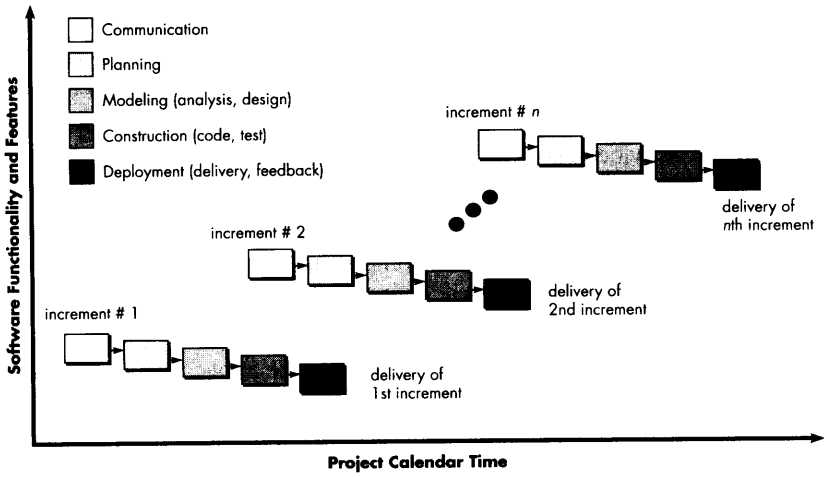


FIGURE 3.2 The incremental model

known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

If your customer demands delivery by a date that is impossible to meet, suggest delivering one or more increments by that date and the rest of the software (additional increments) later.

The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are "stripped down" versions of the final product, but they do provide capability that serves the user and also provides a platform for evaluation by the user.[3]

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

3.3.2 The RAD Model

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaptation

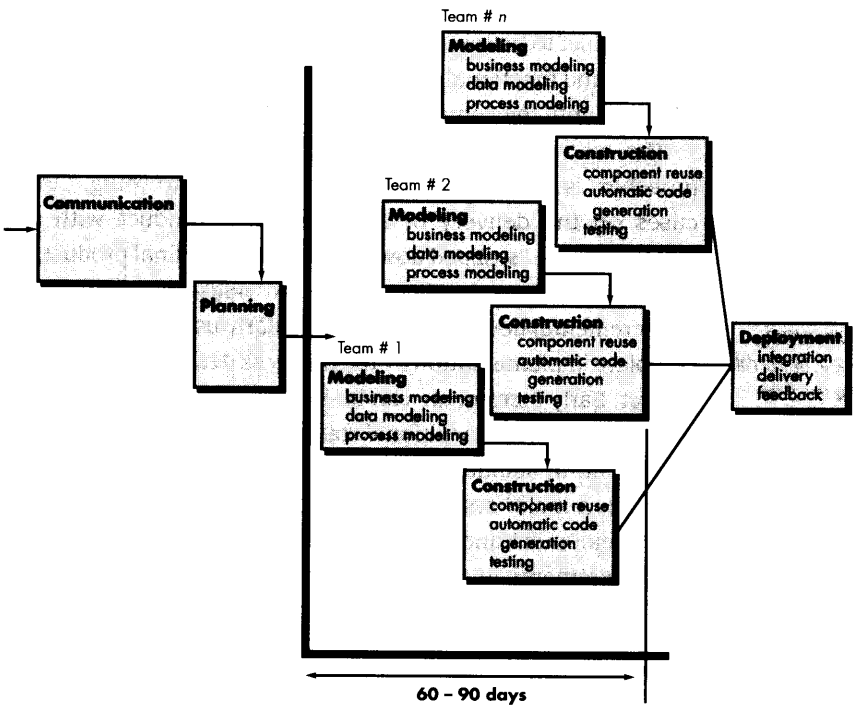
3 It is important to note that an incremental philosophy is also used for all "agile" process models discussed in Chapter 4.

of the waterfall model, in which rapid development is achieved by using a component-based construction approach. If requirements are well understood and project scope is constrained,[4] the RAD process enables a development team to create a "fully functional system" within a very short time period (e.g., 60 to 90 days) [MAR91].

Like other process models, the RAD approach maps into the generic framework activities presented earlier. *Communication* works to understand the business problem and the information characteristics that the software must accommodate. *Planning* is essential because multiple software teams work in parallel on different system functions. *Modeling* encompasses three major phases—business modeling, data modeling and process modeling—and establishes design representations that serve as the basis for RAD's construction activity. *Construction* emphasizes the use of pre-existing software components and the application of automatic code generation. Finally, *deployment* establishes a basis for subsequent iterations, if required [KER94].

The RAD process model is illustrated in Figure 3.3. Obviously, the time constraints imposed on a RAD project demand "scalable scope" [KER94]. If a business application can be modularized in a way that enables each major function to be completed

FIGURE 3.3
The RAD model



4 These conditions are by no means guaranteed. In fact, many software projects have poorly defined requirements at the start. In such cases prototyping or evolutionary approaches (Section 3.4) are much better process options. See [REI95].

in less than three months (using the approach described above), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

What are the drawbacks of the RAD model?

Like all process models, the RAD approach has drawbacks [BUT94]: (1) for large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams; (2) if developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail; (3) if a system cannot be properly modularized, building the components necessary for RAD will be problematic; (4) if high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and (5) RAD may not be appropriate when technical risks are high (e.g., when a new application makes heavy use of new technology).

3.4 Evolutionary Process Models

KEY POINT

Evolutionary process models produce an increasingly more complete version of the software with each iteration.

Software, like all complex systems, evolves over a period of time [GIL88]. Business and product requirements often change as development proceeds, making a straight-line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

3.4.1 Prototyping

Often, a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

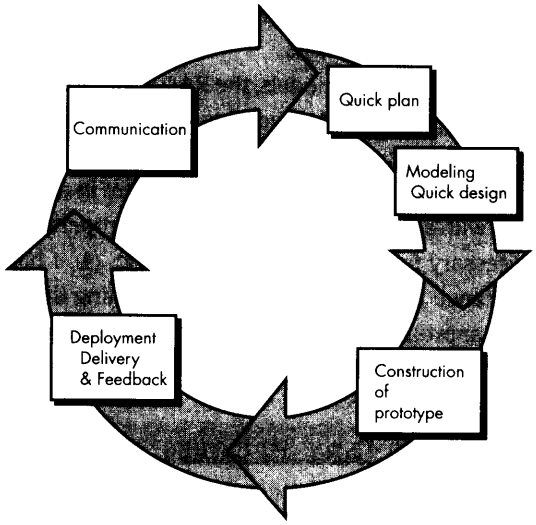
ADVICE

When your customer has a legitimate need but is clueless about the details, develop a prototype as a first step.

Although prototyping can be used as a standalone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 3.4) begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is

FIGURE 3.4
The prototyping model



mandatory. A prototyping iteration is planned quickly and modeling (in the form of a "quick design") occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/end-user (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user. Feedback is used to refine requirements for the software. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools (e.g., report generators, window managers, etc.) that enable working programs to be generated quickly.

But what do we do with the prototype when it has served the purpose described above? Brooks [BRO75] provides one answer:

> In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. . . . When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers.

The prototype can serve as "the first system," the one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system, and

developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire," unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Resist pressure to extend a rough prototype into a production product. Quality almost always suffers as a result.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

SAFEHOME

Selecting a Process Model, Part 1

The scene: Meeting room for the software engineering group at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

The conversation:

Lee: So let's recapitulate. I've spent some time discussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply define the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

Doug: Seems like we've been pretty disorganized in our approach to software in the past.

Ed: I don't know, Doug. We always got product out the door.

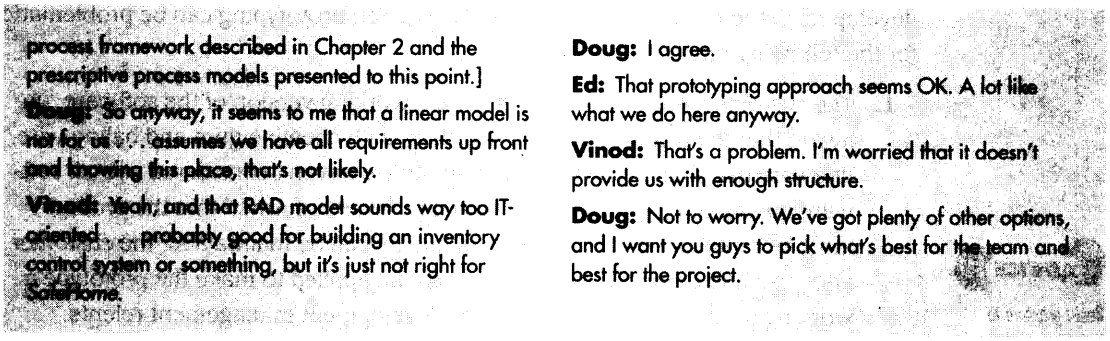
Doug: True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

Jamie: Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particularly if we have a very tight timeline.

Doug (smiling): I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

Jamie (with a frown): My job is to build computer programs, not push paper around.

Doug: Give it a chance before you go negative on me. Here's what I mean. [Doug proceeds to describe the



3.4.2 The Spiral Model

The *spiral model,* originally proposed by Boehm [BOE88], is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm [BOE01] describes the model in the following manner:

> The spiral development model is a *risk-driven process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

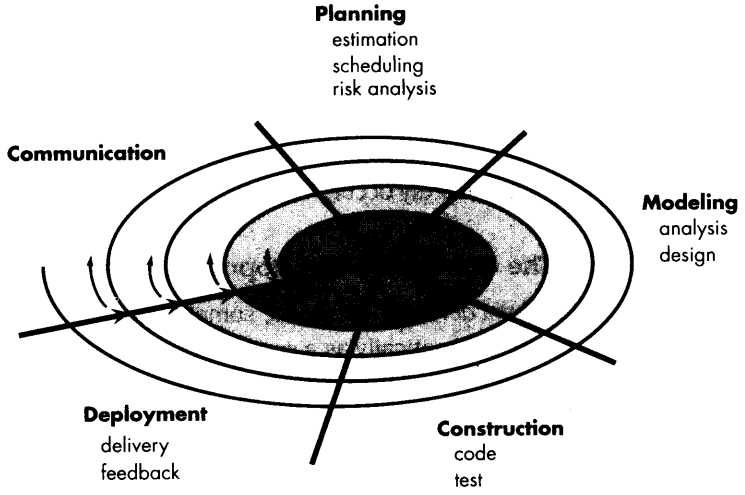
The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.[5] Each of the framework activities represent one segment of the spiral path illustrated in Figure 3.5. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 25) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass

5 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [BOE88]. More recent discussion of Boehm's spiral model can be found in [BOE98].

FIGURE 3.5
A typical spiral model



through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

WebRef
Useful information about the spiral model can be obtained at **www.sei.cmu.edu/cbs/spiral2000/.**

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" which starts at the core of the spiral and continues for multiple iterations[6] until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

If your management demands fixed-budget development (generally a bad idea), the spiral can be a problem: as each circuit is completed, project cost is revisited and revised.

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more importantly, enables the developer to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical

6 The arrows pointing inward along the axis separating the *deployment* region from the *communication* region indicate a potential for local iteration along the same spiral path.

risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

3.4.3 The Concurrent Development Model

The *concurrent development model,* sometimes called *concurrent engineering,* can be represented schematically as a series of framework activities, software engineering actions and tasks, and their associated states. For example, the *modeling* activity defined for the spiral model is accomplished by invoking the following actions: prototyping and/or analysis modeling and specification and design.[7]

The concurrent model is often more appropriate for system engineering projects (Chapter 6) where different engineering teams are involved.

Figure 3.6 provides a schematic representation of one software engineering task within the modeling activity for the concurrent process model. The activity—*modeling*—may be in any one of the states[8] noted at any given time. Similarly, other activities or tasks (e.g., communication or construction) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the *communication* activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The *modeling* activity which existed in the **none** state while initial communication was completed, now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the *modeling* activity moves from the **under development** state into the **awaiting changes** state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a software engineering action that occurs during the modeling activity), an inconsistency in the analysis model is uncovered. This generates the event *analysis model correction* which will trigger the analysis action from the **done** state into the **awaiting changes** state.

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a network of activities. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

---

7 It should be noted that analysis and design are complex actions that require substantial discussion. Part 2 of this book considers these topics in detail.

8 A *state* is some externally observable mode of behavior.

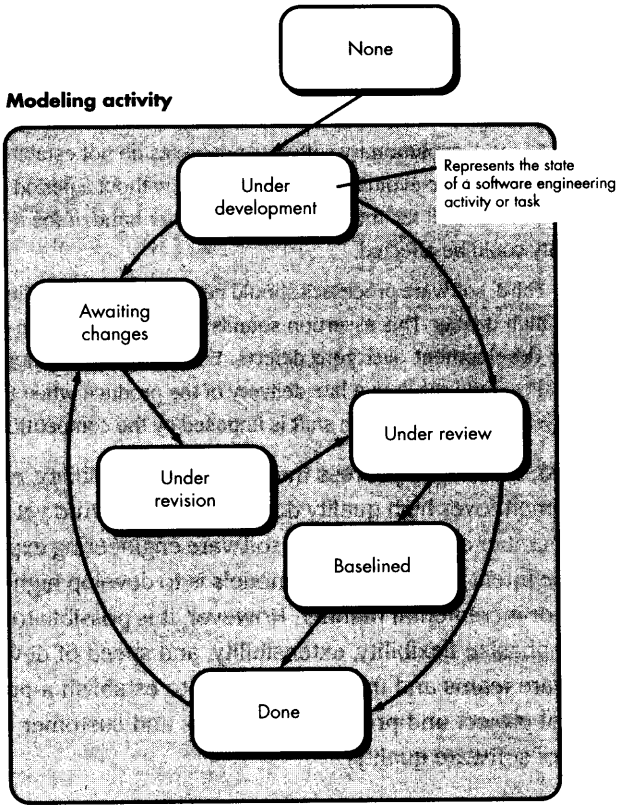


FIGURE 3.6 One element of the concurrent process model

3.4.4 A Final Comment on Evolutionary Processes

We have already noted that modern computer software is characterized by continual change, by very tight timelines, and by an emphatic need for customer/user satisfaction. In many cases, time-to-market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.[9]

> "We only live for and only tomorrow leads my way."
>
> Dave [illegible]

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. These are summarized by Nogueira and his colleagues [NOG00]:

9 It is important to note, however, that being the first to reach a market is no guarantee of success. In fact, many very successful software products have been second or even third to reach the market (learning from the mistakes of their predecessors).

> Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.
>
> Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand, if the speed is too slow then productivity could be affected. . . .
>
> Third, software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts (e.g., [YOU95], [BAC97]).

The intent of evolutionary models is to develop high-quality software[10] in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

SAFEHOME

Selecting a Process Model, Part 2

The scene: Meeting room for the software engineering group at CPI Corporation, a company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Ed and Vinod, members of the software engineering team.

The conversation:

(Doug describes evolutionary process options.)

Ed: Now I see something I like. An incremental approach makes sense and I really like the flow of that spiral model thing. That's keepin' it real.

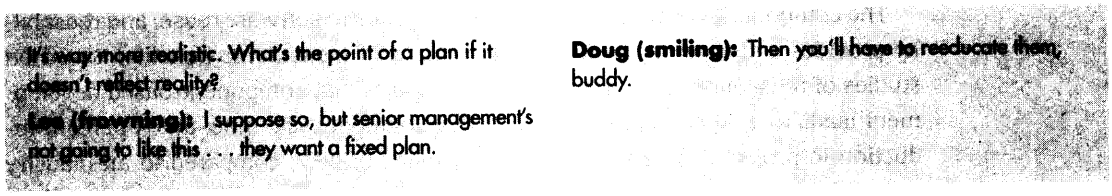
Vinod: I agree. We deliver an increment, learn from customer feedback, replan, and then deliver another increment. It also fits into the nature of the product. We can have something on the market fast and then add functionality with each version, er, increment.

Lee: Wait a minute, did you say that we regenerate the plan with each tour around the spiral, Doug? That's not so great, we need one plan, one schedule, and we've got to stick to it.

Doug: That's old school thinking, Lee. Like Ed said, we've got to keep it real. I submit that it's better to tweak the plan as we learn more and as changes are requested.

---

10 In this context, software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Chapter 26.



3.5 Specialized Process Models

Special process models take on many of the characteristics of one or more of the conventional models presented in the preceding sections. However, specialized models tend to be applied when a narrowly defined software engineering approach is chosen.[11]

3.5.1 Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, can be used when software is to be built. These components provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software.

WebRef
Useful information on component-based development can be obtained at www.cbd-hq.com.

The *component-based development* model (Chapter 30) incorporates many of the characteristics of the spiral model. It is evolutionary in nature [NIE92], demanding an iterative approach to the creation of software. However, the model composes applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages[12] of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

- Available component-based products are researched and evaluated for the application domain in question.
- Component integration issues are considered.
- A software architecture (Chapter 10) is designed to accommodate the components.
- Components (Chapter 11) are integrated into the architecture.
- Comprehensive testing (Chapters 13 and 14) is conducted to ensure proper functionality.

---

11 In some cases, these specialized process models might better be characterized as a collection of techniques or a methodology for accomplishing a specific software development goal. However, they do imply a process.

12 Object-oriented technology is discussed through Part 2 of this book. In this context, a class encapsulates a set of data and the procedures that process the data. A package of classes is a collection of related classes that work together to achieve some end result.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability, QSM Associates, Inc. reports that component-based development leads to a 70 percent reduction in development cycle time; an 84 percent reduction in project cost; and a productivity index of 26.2, compared to an industry norm of 16.9 [YOU94]. Although these results are a function of the robustness of the component library, there is little question that the component-based development model provides significant advantages for software engineers.

3.5.2 The Formal Methods Model

The *formal methods* model (Chapter 28) encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [MIL87, DYE92], is currently applied by some software development organizations and is discussed in Chapter 29.

> **"It is easier to write an incorrect program than understand a correct one."**
>
> **Alan Perlis**

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through *ad hoc* review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

If formal methods can demonstrate software correctness, why is it they are not widely used?

- The development of formal models is currently quite time-consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers who would suffer severe economic hardship should software errors occur.

WebRef

A wide array of resources and information on AOP can be found at **aosd.net.**

3.5.3 Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object-oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain "concerns"—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*. *Aspectual requirements* define those crosscutting concerns that have impact across the software architecture. *Aspect-oriented software development* (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern" [ELR01].

AOSD defines "aspects" that express customer concerns that cut across multiple system functions, features, and information.

Grundy [GRU02] provides further discussion of aspects in the context of what he calls *aspect-oriented component engineering* (AOCE):

> AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both the spiral and concurrent process models (Sections 3.4.2 and 3.4.3). The evolutionary nature of the spiral is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

A detailed discussion of aspect-oriented software development is best left to books dedicated to the subject. The interested reader should see [GRA03], [KIS02], or [ELR01].

SOFTWARE TOOLS

Process Management

Objective: To assist in the definition, execution, and management of prescriptive process models.

Mechanics: Process management tools allow a software organization or team to define a complete software process model (framework activities, actions, tasks, QA checkpoints, milestones, and work products). In addition, the tools provide a roadmap as software engineers do technical work and a template for managers who must track and control the software process.

Representative Tools[13]

GDPA, a research process definition tool suite, developed at Bremen University in Germany (www.informatik.uni-bremen.de/uniform/gdpa/home.htm), provides a wide array of process modeling and management functions.

SpeeDev, developed by SpeeDev Corporation (www.speedev.com), encompasses a suite of tools for process definition, requirements management, issue resolution, project planning, and tracking.

Step Gate Process, developed by Objexis (www.objexis.com), encompasses many tools that assist in workflow automation.

A worthwhile discussion of the methods and notation that can be used to define and describe a complete process model can be found at http://205.252.62.38/English/D-ProcessNotation.htm.

3.6 THE UNIFIED PROCESS

In their seminal book on the *Unified Process,* Ivar Jacobson, Grady Booch, and James Rumbaugh [JAC99] discuss the need for a "use-case driven, architecture-centric, iterative and incremental" software process when they state:

> Today, the trend in software is toward bigger, more complex systems. That is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of the Internet for exchanging all kinds of information. . . . Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

In some ways the Unified Process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development (Chapter 4). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (i.e., the use-case[14]). It emphasizes the important role of software architec-

13 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

14 A *use-case* (Chapters 7 and 8) is a text narrative or template that describes a system function or feature from the user's point of view. A use-case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

ture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [JAC99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

In this section we present an overview of the key elements of the Unified Process. In Part 2 of this book, we discuss the methods that populate the process and the complementary UML[15] modeling techniques and notation that are required as the Unified Process is applied in actual software engineering work.

3.6.1 A Brief History

During the 1980s and into the early 1990s, object-oriented (OO) methods and programming languages[16] gained a widespread audience throughout the software engineering community. A wide variety of object-oriented analysis (OOA) and design (OOD) methods were proposed during the same time period, and a general purpose object-oriented process model (similar to the evolutionary models presented in this chapter) was introduced. Like most "new" paradigms for software engineering, adherents of each of the OOA and OOD methods argued about which was best, but no individual method or language dominated the software engineering landscape.

During the early 1990s James Rumbaugh [RUM91], Grady Booch [BOO94], and Ivar Jacobson [JAC92] began working on a "unified method" that would combine the best features of each of their individual methods and adopt additional features proposed by other experts (e.g., [WIR90]) in the OO field. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of OO systems. By 1997, UML became an industry standard for object-oriented software development. At the same time, the Rational Corporation and other vendors developed automated tools to support UML methods.

UML provides the necessary technology to support object-oriented software engineering practice, but it does not provide the process framework to guide project teams in their application of the technology. Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process,* a framework for object-oriented software engineering using UML. Today, the Unified Process and UML are widely used on OO projects of all kinds, The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

An array of work products (e.g., models and documents) can be produced as a consequence of applying UML. However, these are often pared down by software engineers to make development more agile and more responsive to change.

---

15 UML (the Unified Modeling Language) has become the most widely used notation for analysis and design modeling. It represents a marriage of three important object-oriented notations.

16 If you are unfamiliar with object-oriented methods, a brief overview is presented in Chapters 8 and 9. For a more detailed presentation see [REE02], [STI01], or [FOW99].

3.6.2 Phases of the Unified Process[17]

WebRef
Useful white papers on the UP can be found at www.rational.com/products/rup/whitepapers.jsp.

We have discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process is no exception. Figure 3.7 depicts the "phases" of the Unified Process (UP) and relates them to the generic activities that have been discussed in Chapter 2.

The *inception* phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, business requirements for the software are identified, a rough architecture for the system is proposed, and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use-cases that describe what features and functions are desired by each major class of users. In general, a use-case describes a sequence of actions that are performed by an *actor* (e.g., a person, a machine, another system) as the actor interacts with the software. Use-cases help to identify the scope of the project and provide a basis for project planning.

KEY POINT
UP *phases* are similar in intent to the generic framework activities defined in this book.

Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The *elaboration* phase encompasses the customer communication and modeling activities of the generic process model (Figure 3.7). Elaboration refines and expands the preliminary use-cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use-case model, the analysis model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [ARL02] that represents a "first cut" executable system.[18] The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan may be made at this time.

WebRef
Illuminating discussion and commentary on the UP can be found at www.unifiedprocess.org.

The *construction* phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each

17 The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Corporation, a primary contributor to the development and refinement of the process and a builder of complete environments (tools and technology) that support the process.

18 It is important to note that the architectural baseline is not a prototype (Section 3.4.1) in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

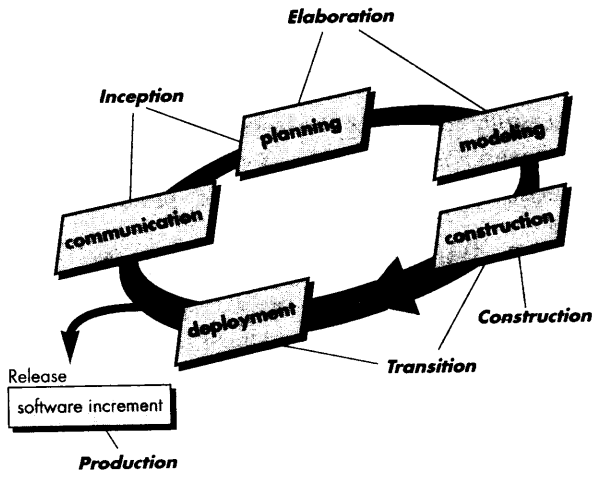


FIGURE 3.7 The Unified Process

use-case operational for end-users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions of the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use-cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

The *transition* phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment activity. Software is given to end-users for beta testing[19], and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, trouble-shooting guides, and installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production* phase of the UP coincides with the deployment activity of the generic process. During this phase, the on-going use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

19 *Beta testing* is a controlled testing action (Chapter 13) in which the software is used by actual end-users with the intent of uncovering defects and deficiencies. A formal defect/deficiency reporting scheme is established, and the software team assesses feedback.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (defined in Chapter 2). That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

3.6.3 Unified Process Work Products

Figure 3.8 illustrates the key *work products* produced as a consequence of the four technical UP phases. During the inception phase, the intent is to establish an overall "vision" for the project, identify a set of business requirements, make a business case for the software, and define project and business risks that may represent a threat to success. From the software engineer's point of view, the most important work product produced during the inception is the *use-case model*—a collection of use-cases that describe how outside actors (human and nonhuman "users" of the software) interact with the system and gain value from it. In essence, the use-case model is a collection of usage scenarios described with standardized templates that imply software features and functions by describing a set of preconditions, a flow of events or a scenario, and a set of post-conditions for the interaction that is depicted. Initially, use-cases describe requirements at the business domain level (i.e., the level of abstraction is high). However, the use-case model is refined and elaborated as each UP phase is conducted and serves as an important input for the creation of subsequent work products. During the inception phase only 10 to 20 percent of the use-case model is completed. After elaboration, between 80 and 90 percent of the model has been created.

The elaboration phase produces a set of work products that elaborate requirements (including nonfunctional[20] requirements) and produce an architectural description and a preliminary design. As the software engineer begins object-oriented analysis, the primary objective is to define a set of analysis classes that adequately describe the behavior of the system. The UP *analysis model* is the work product that is developed as a consequence of this activity. The classes and analysis packages (collections of classes) defined as part of the analysis model are refined further into a *design model* which identifies design classes, subsystems, and the interfaces between subsystems. Both the analysis and design models expand and refine an evolving representation of software architecture. In addition, the elaboration phase revisits risks and the project plan to ensure that each remains valid.

The construction phase produces an *implementation* model that translates design classes into software components that will be built to realize the system, and a *deployment* model maps components into the physical computing environment. Finally,

20 Requirements that cannot be discerned from the use-case model.

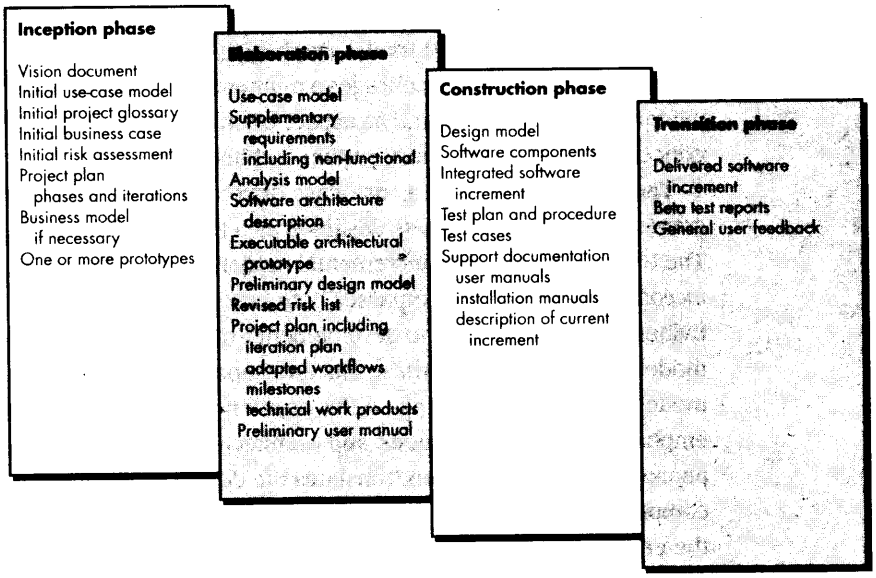


FIGURE 3.8 Major work products produced for each UP phase

a *test* model describes tests that are used to ensure that use-cases are properly reflected in the software that has been constructed.

The transition phase delivers the software increment and assesses work products that are produced as end-users work with the software. Feedback from beta testing and qualitative requests for change are produced at this time.

3.7 SUMMARY

Prescriptive software process models have been applied for many years in an effort to bring order and structure to software development. Each of these conventional models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

The waterfall model suggests a linear progression of framework activities that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight timelines) in the software world. It does, however, have applicability in situations where requirements are well-defined and stable.

Incremental software process models produce software as a series of increment releases. The RAD model is designed for larger projects that must be delivered in tight time frames.

Evolutionary process models recognize the iterative nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply

across all software engineering activities—from concept development to long-term system maintenance.

The component-based model emphasizes component reuse and assembly. The formal methods model encourages a mathematically based approach to software development and verification. The aspect-oriented model accommodates cross-cutting concerns that span the entire system architecture.

The Unified Process is a "use-case driven, architecture-centric, iterative and incremental" software process designed as a framework for UML methods and tools. The Unified Process is an incremental model in which five phases are defined: (1) an *inception* phase that encompasses both customer communication and planning activities and emphasizes the development and refinement of use-cases as a primary model; (2) an *elaboration* phase that encompasses the customer communication and modeling activities focusing on the creation of analysis and design models with an emphasis on class definitions and architectural representations; (3) a *construction* phase that refines and then translates the design model into implemented software components; (4) a *transition* phase that transfers the software from the developer to the end-user for beta testing and acceptance; and (5) a *production* phase in which on-going monitoring and support are conducted.

REFERENCES

[AMB02] Ambler, S., and L. Constantine, *The Unified Process Inception Phase,* CMP Books, 2002.

[ARL02] Arlow, J., and I. Neustadt, *UML and the Unified Process,* Addison-Wesley, 2002.

[BAC97] Bach, J., "Good Enough Quality: Beyond the Buzzword," *IEEE Computer,* vol. 30, no. 8, August 1997, pp. 96–98.

[BOE88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer,* vol. 21, no. 5, May 1988, pp. 61–72.

[BOE98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer,* vol. 31, no. 7, July 1998, pp. 33–44.

[BOE01] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition," *CrossTalk,* May 2001, available at http://www.stsc.hill.af.mil/crosstalk/ 2001/05/boehm.html.

[BOO94] Booch, G., *Object-Oriented Analysis and Design,* 2nd ed., Benjamin Cummings, 1994.

[BRA94] Bradac, M., D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Trans. Software Engineering,* vol. 20, no. 10, October 1994, pp. 774–784.

[BRO75] Brooks, F., *The Mythical Man-Month,* Addison-Wesley, 1975.

[BUT94] Butler, J., "Rapid Application Development in Action," *Managing System Development,* Applied Computer Research, vol. 14, no. 5, May 1994, pp. 6–8.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development,* Wiley, 1992.

[ELR01] Elrad, T., R. Filman, and A. Bader (eds.), "Aspect-Oriented Programming," *Comm. ACM,* vol. 44, no. 10, October 2001, special issue.

[FOW99] Fowler, M., and K. Scott, *UML Distilled,* 2nd ed., Addison-Wesley, 1999.

[GIL88] Gilb, T., *Principles of Software Engineering Management,* Addison-Wesley, 1988.

[GRA03] Gradecki, J., and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java,* Wiley, 2003.

[GRU02] Grundy, J., "Aspect-Oriented Component Engineering," 2002, http:// www.cs.auckland.ac.nz/~john-g/aspects.html.

[HAN95] Hanna, M., "Farewell to Waterfalls," *Software Magazine,* May 1995, pp. 38–46.

[HES96] Hesse, W., "Theory and Practice of the Software Process—A Field Study and its Implications for Project Management," *Software Process Technology, 5th European Workshop,* EWSPT 96, Springer LNCS 1149, 1996, pp. 241–256.