PART

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 the prescriptive process models, and what are their strengths and weaknesses?
- 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 PROCESS MODELS

Building computer software is an iterative social learning process, and the outcome, something that Baetjer [Bae98] would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

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 activities, actions, and tasks 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 so that it is appropriate for the products that they build and the demands of their marketplace.

KEY CONCEPTS

QUICK LOOK

What is it? When you work to build a product or system, it's important to follow a series of predictable steps (a road map) that helps you deliver a high-quality product on time. This road map is called a "software process."

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

Why is it important? A process provides stability, control, and organization to an activity so that it does not become chaotic. However, a modern software engineering process must be "agile." It must include only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? The process that you adopt depends on the software that you're building. A process might be appropriate for creating software for an aircraft avionics system but may not work well for the creation of a mobile app or video game.

What is the work product? The work products are the programs, documents, and data produced by the engineering activities and tasks included in the process.

How do I ensure that I've done it right? The quality, timeliness, and long-term viability of the product built are the best indicators of the success of the process used.

2.1 A Generic Process Model

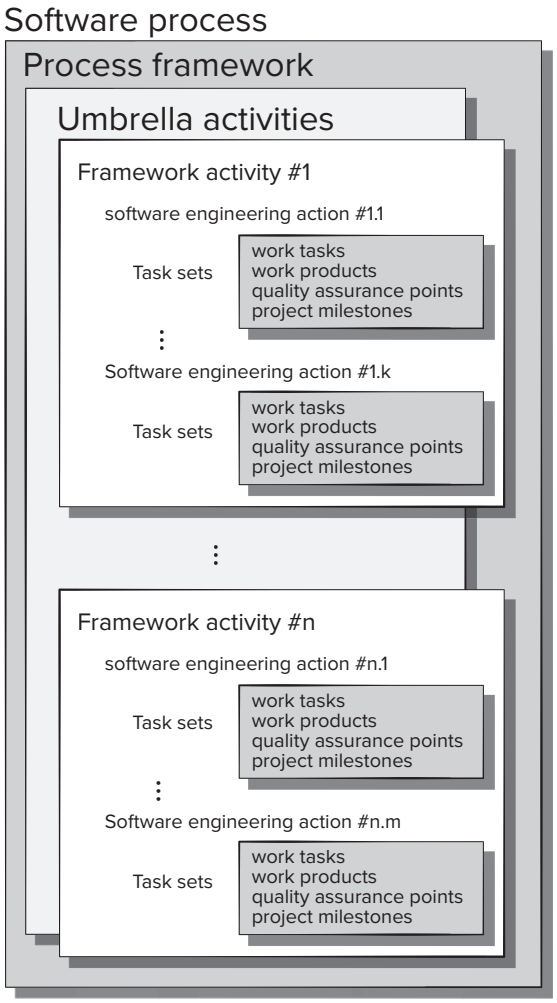
In Chapter 1, a process was defined as a collection of activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 2.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As we discussed in Chapter 1, a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control,

FIGURE 2.1

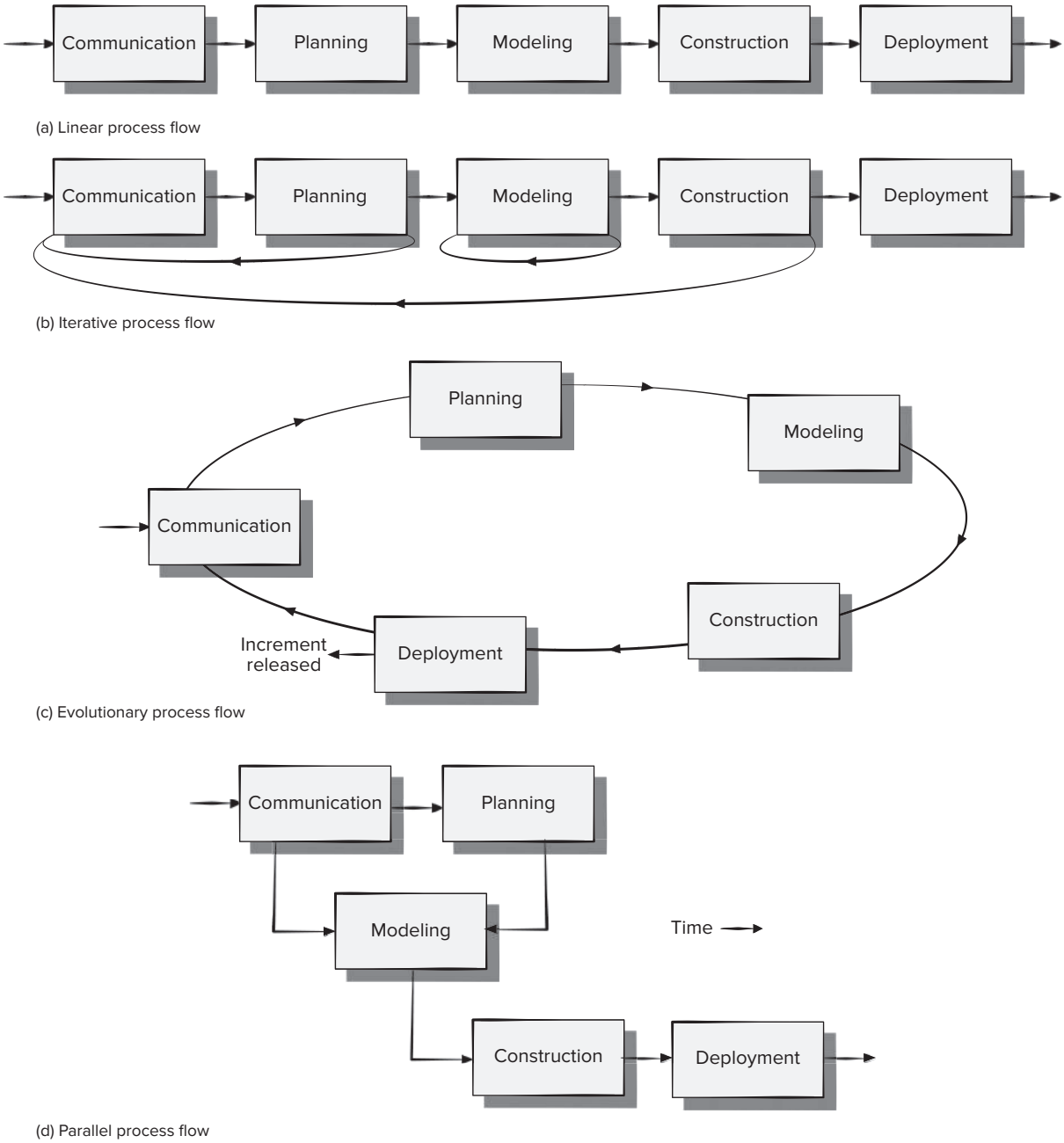
A software process framework



risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

You should note that one important aspect of the software process has not been discussed yet. This aspect—called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time. It is illustrated in Figure 2.2.

FIGURE 2.2 **Process flow**



A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 2.2b). An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c). A *parallel process flow* (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

2.2 Defining a Framework Activity

Although we have described five framework activities and provided a basic definition of each in Chapter 1, a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the **communication** activity might encompass little more than a phone call or e-mail with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and develop notes.
3. Organize notes into a brief written statement of requirements.
4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with lots of stakeholders, each with a different set of (sometimes conflicting) requirements, the communication activity might have six distinct actions: *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions might have many work tasks and in some cases a number of distinct work products.

2.3 Identifying a Task Set

Referring again to Figure 2.1, each software engineering action (e.g., *elicitation,* an action associated with the **communication** activity) can be 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. Different projects demand different task sets. You should choose a task set that best accommodates the

TASK SET

A task set defines the actual work that needs to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "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 application specification 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 level of formality. The software team chooses the task set that allows it to achieve the goal for each action and still maintain quality and agility.

needs of the project and the characteristics of your team. This implies that a software engineering action should be adapted to the specific needs of the software project and the characteristics of the project team.

2.4 PROCESS ASSESSMENT AND IMPROVEMENT

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 15). Process patterns must be coupled with solid software engineering practice (Part Two of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for successful software engineering.[1]

The current thinking among most engineers is that software processes and activities should be assessed using numeric measures or software analytics (metrics). The

1 The SEI's CMMI-DEV [CMM07] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

progress you have made in a journey toward an effective software process will define the degree to which you can measure improvement in a meaningful way. The use of software process metrics to assess process quality is introduced in Chapter 17. A more detailed discussion of process assessment and improvement methods is presented in Chapter 28.

2.5 PRESCRIPTIVE PROCESS MODELS

Prescriptive process models define a predefined set of process elements and a predictable process work flow. Prescriptive process models[2] strive for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional 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 the sections that follow, we provide an overview of the prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapter 1, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner. In Chapters 3 and 4 we will discuss software engineering practices that strive to accommodate the changes that are inevitable during the development of many software projects.

2.5.1 The Waterfall Model

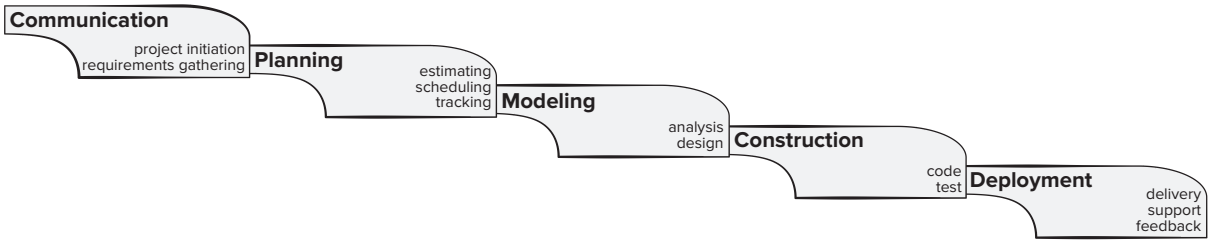
There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software because it needs to accommodate changes to mandated 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 *linear sequential model*, suggests a systematic, sequential approach[3] to software development that begins with customer

2 Prescriptive process models are sometimes referred to as "traditional" process models.

3 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.

FIGURE 2.3 The waterfall model



specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).

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

1. Real projects rarely follow the sequential work flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly at the beginning of most projects.
3. The customer must have patience because a working version of the program(s) will not be available until late in the project time span.
4. Major blunders may not be detected until the working program is reviewed.

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.

2.5.2 Prototyping Process Model

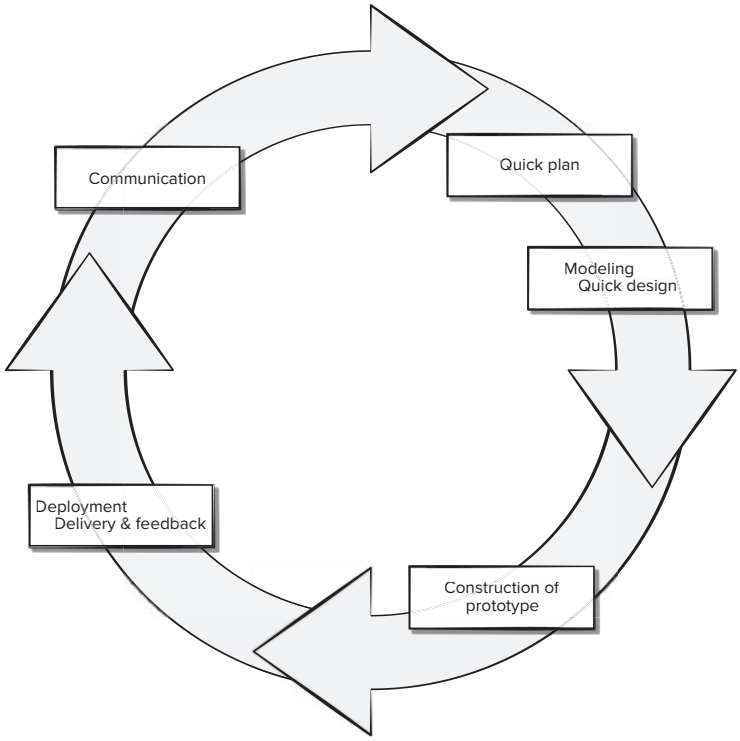
Often, a customer defines a set of general objectives for software but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction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

Although prototyping can be used as a stand-alone 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 you and other stakeholders to better understand what is to be built when requirements are fuzzy.

For example, a fitness app developed using incremental prototypes might deliver the basic user interface screens needed to sync a mobile phone with the fitness device

FIGURE 2.4

The prototyping paradigm



and display the current data; the ability to set goals and store the fitness device data on the cloud might be included in the second prototype, creating and modifying the user interface screens based on customers feedback; and a third prototype might include social media integration to allow users to set fitness goals and share progress toward them with a set of friends.

The prototyping paradigm (Figure 2.4) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

Both stakeholders and software engineers 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. Stakeholders see what appears to be a working version of the software. They may be unaware that the prototype architecture (program structure) is also evolving. This means that the developers may not have considered the overall software quality or long-term maintainability.
2. As a software engineer, you may be tempted to make implementation compromises to get a prototype working quickly. If you are not careful, these less-than-ideal choices have now become an integral part of the evolving system.

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 time line.

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 process framework described in Chapter 1 and the prescriptive process models presented to this point.)

Doug: So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

Vinod: Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

Doug: I agree.

Ed: That prototyping approach seems okay. A lot like what we do here anyway.

Vinod: That's a problem. I'm worried that it doesn't provide us with enough structure.

Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

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, all stakeholders should agree that the prototype is built in part to serve as a mechanism for defining requirements. It is often desirable to design a prototype so it can be evolved into the final product. The reality is developers may need to discard (at least in part) a prototype to better meet the customer's evolving needs.

2.5.3 Evolutionary Process Model

Software, like all complex systems, evolves over time. Business and product requirements often change as development proceeds, making a straight-line path to an end product unrealistic. Tight market deadlines may make completion of a comprehensive software product impossible. It might be possible to create a limited version of a product to meet competitive or business pressure and release a refined version once all system features are better understood. In a situation like this you need a process model that has been explicitly designed to accommodate a product that grows and changes.

Originally proposed by Barry Boehm [Boe88], the *spiral model* 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.

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

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.[4] Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.5. 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 26) 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.

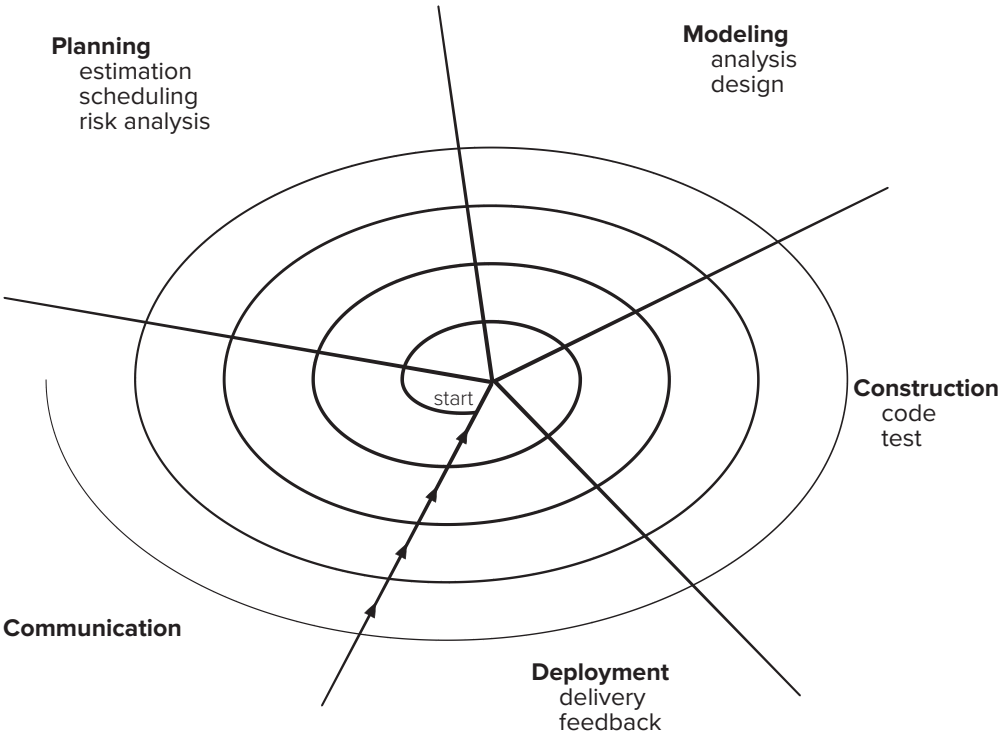
The first circuit around the spiral (beginning at the inside streamline nearest the center, as shown in Figure 2.5) might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass 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.

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. The spiral model

4 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] and [Boe01a].

FIGURE 2.5

A typical spiral model



is a realistic approach to the development of large-scale systems and software. It uses prototyping as a risk reduction mechanism. The spiral model demands a direct consideration of technical 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.

We have already noted that modern computer software is characterized by continual change, by very tight time lines, 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.[5]

The intent of evolutionary models is to develop high-quality software[6] in an iterative or incremental manner. However, it is possible to use an evolutionary process to

5 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).

6 In this context, software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Part Two of this book.

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; Vinod and Jamie, members of the software engineering team.

The conversation: (Doug describes evolutionary process options.)

Jamie: 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, re-plan, 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 the guys 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. It's way more realistic. What's the point of a plan if it doesn't reflect reality?

Lee (frowning): I suppose so, but . . . senior management's not going to like this . . . they want a fixed plan.

Doug (smiling): Then you'll have to reeducate them, buddy.

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).

2.5.4 Unified Process Model

In some ways the Unified Process (UP) [Jac99] is an attempt to draw on the best features and characteristics of traditional software process models but characterize them in a way that implements many of the best principles of agile software development (Chapter 3). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case).[7] It emphasizes the important role of software architecture 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.

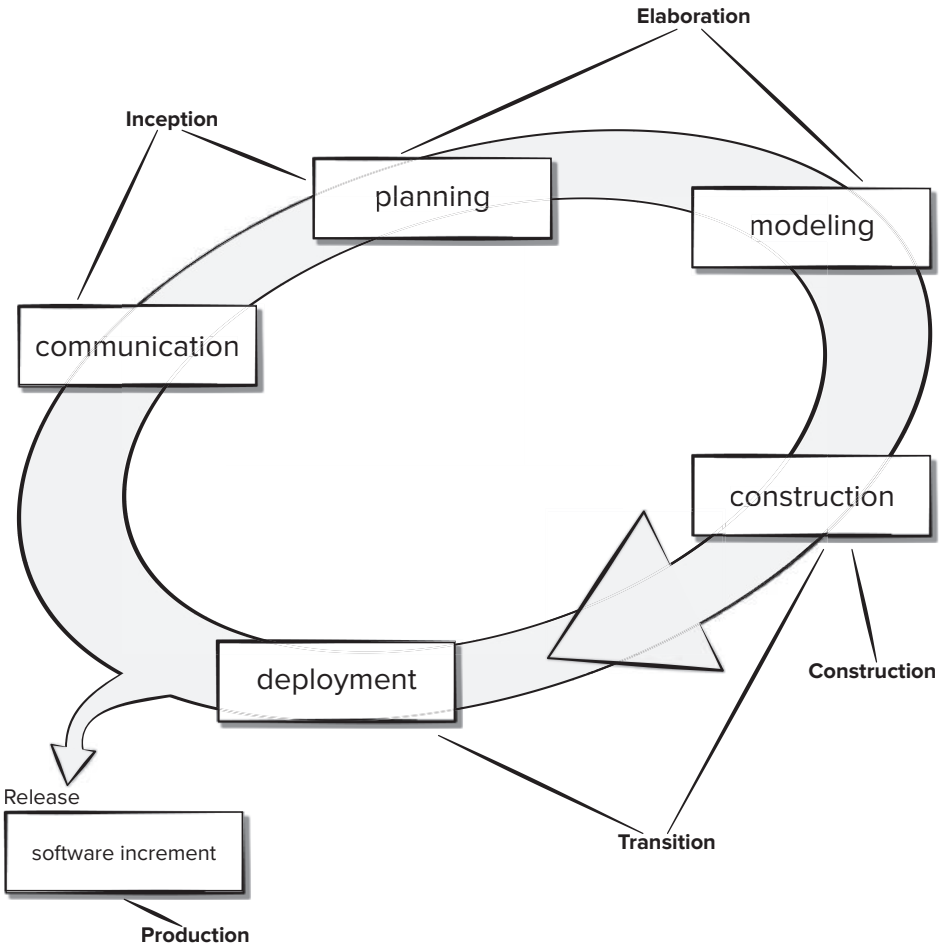
7 A *use case* (Chapter 7) is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

UML, the *unified modeling language,* was developed to support their work. UML contains a robust notation for the modeling and development of object-oriented systems and has became a de facto industry standard for modeling software of all types. UML is used throughout Part Two of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial and a list of recommended books for those who are unfamiliar with basic UML notation and modeling rules.

Figure 2.6 depicts the "phases" of the Unified Process and relates them to the generic activities that were discussed in Section 2.1.

The *inception phase* of the UP is where customer communication and planning takes place. Fundamental business requirements are described through a set of preliminary use cases (Chapter 7) that describe which features and functions each major class of users desires that will become realized in the software architecture. Planning identifies resources, assesses major risks, and defines a preliminary schedule for the software increments.

FIGURE 2.6
The Unified Process



The *elaboration phase* encompasses the planning and modeling activities of the generic process model (Figure 2.6). Elaboration refines and expands the preliminary use cases and creates an architectural baseline that includes five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model.[8] Modifications to the plan are often made at this time.

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests[9] 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 (delivery and feedback) activity. Software and supporting documentation is given to end users for beta testing, and user feedback reports both defects and necessary changes. 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 ongoing 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.

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.

2.6 PRODUCT AND PROCESS

Some of the strengths and weaknesses of the process models we have discussed are summarized in Table 2.1. In previous editions of this book we have discussed many others. The reality is that no process is perfect for every project. Usually the software team adapts one or more of the process models discussed in 2.5 or the agile process models discussed in Chapter 3 to meet their needs for the project at hand.

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

9 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 19 through 21).

TABLE 2.1
Comparing process models

Waterfall pros	It is easy to understand and plan. It works for well-understood small projects. Analysis and testing are straightfoward.
Waterfall cons	It does not accommodate change well. Testing occurs late in the process. Customer approval is at the end.
Prototyping pros	There is a reduced impact of requirement changes. The customer is involved early and often. It works well for small projects. There is reduced likelihood of product rejection.
Prototyping cons	Customer involvement may cause delays. There may be a temptation to "ship" a prototype. Work is lost in a throwaway prototype. It is hard to plan and manage.
Spiral pros	There is continuous customer involvement. Development risks are managed. It is suitable for large, complex projects. It works well for extensible products.
Spiral cons	Risk analysis failures can doom the project. The project may be hard to manage. It requires an expert development team.
Unified Process pros	Quality documentation is emphasized. There is continuous customer involvement. It accommodates requirements changes. It works well for maintenance projects.
Unified Process cons	Use cases are not always precise. It has tricky software increment integration. Overlapping phases can cause problems. It requires an expert development team.

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless 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. . . .
>
> 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 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.
>
> . . . 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. . . .

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. As a creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

2.7 SUMMARY

A generic process model for software engineering encompasses a set of framework and umbrella activities, actions, and work tasks. Each of a variety of process models can be described by a different process flow—a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

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

Sequential process models, such as the waterfall model, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental 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 Unified Process is a "use case driven, architecture-centric, iterative and incremental" software process designed as a framework for UML methods and tools.

PROBLEMS AND POINTS TO PONDER

2.1. Baetjer [Bae98] 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 (1) designers should ask users, (2) users should ask designers, (3) users should ask themselves about the software product that is to be built, and (4) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

2.2. Discuss the differences among the various process flows described in Section 2.1. Identify the types of problems that might be applicable to each of the generic flows described.

2.3. Try to develop a set of actions for the communication activity. Select one action, and define a task set for it.

2.4. A common problem during communication occurs when you encounter two stakeholders who have conflicting ideas about what the software should be. That is, they have mutually conflicting requirements. Develop a process pattern that addresses this problem and suggest an effective approach to it.

2.5. Provide three examples of software projects that would be amenable to the waterfall model. Be specific.

2.6. Provide three examples of software projects that would be amenable to the prototyping model. Be specific.

2.7. As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?

2.8. Is it possible to combine process models? If so, provide an example.

2.9. What are the advantages and disadvantages of developing software in which quality is "good enough"? That is, what happens when we emphasize development speed over product quality?

2.10. It is possible to prove that a software component and even an entire program is correct? So why doesn't everyone do this?

2.11. Are the Unified Process and UML the same thing? Explain your answer.