PART

Two

MODELING

In this part of *Software Engineering: A Practitioner's Approach,* you'll learn about the principles, concepts, and methods that are used to create high-quality requirements and design models. These questions are addressed in the chapters that follow:

- What concepts and principles guide software engineering practice?
- What is requirements engineering, and what are the underlying concepts that lead to good requirements analysis?
- How is the requirements model created, and what are its elements?
- What are the elements of a good design?
- How does architectural design establish a framework for all other design actions, and what models are used?
- How do we design high-quality software components?
- What concepts, models, and methods are applied as the user experience is designed?
- What is pattern-based design?
- What specialized strategies and methods are used to design mobile apps?

Once these questions are answered, you'll be better prepared to apply software engineering practice.

CHAPTER

6 PRINCIPLES THAT GUIDE PRACTICE

Software engineers are often depicted as working long hours by themselves to meet impossible deadlines without connecting to other people. This is a dark image of software engineering practice to be sure. Yet, as we discussed in the previous chapters, most software engineers work on teams and frequently interact with their stakeholders. If you search for surveys of technical professionals on the Internet, you will see software engineers listed among those experiencing the greatest satisfaction from their jobs.

KEY CONCEPTS

QUICK LOOK

What is it? Software engineering practice is a broad array of principles, concepts, methods, and tools that you must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted.

Who does it? Practitioners (software engineers) and their managers conduct a variety of software engineering tasks.

Why is it important? Software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Software practice provides you with the details you'll need to drive along the road. It tells you where the bridges, the roadblocks, and the forks are located. It helps you understand the concepts and principles that must be understood and followed to drive safely and rapidly. It instructs you on how to drive, where to slow down, and where to speed up. In the context of software engineering, practice is what you do day in and day out as software evolves from an idea to a reality.

What are the steps? Four elements of practice apply regardless of the process model that is chosen. They are principles, concepts, methods, and tools. Tools support the application of methods.

What is the work product? Practice encompasses the technical activities that produce all work products that are defined by the software process model that has been chosen.

How do I ensure that I've done it right? First, have a firm understanding of the principles that apply to the work (e.g., design) that you're doing at the moment. Then, be certain that you've chosen an appropriate method for the work, be sure that you understand how to apply the method, use automated tools when they're appropriate for the task, and be adamant about the need for techniques to ensure the quality of work products that are produced. You also need to be agile enough to make changes to your plans and methods as needed.

People who create computer software practice the art or craft or discipline[1] that is software engineering. But what is software engineering "practice"? In a generic sense, *practice* is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how-to's to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

Various aspects of software engineering practice will be examined throughout the remainder of this book. In this chapter, our focus is on principles and concepts that guide software engineering practice in general.

6.1 Core Principles

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs the framework and umbrella activities and produces a set of software engineering work products. At the practice level, core principles establish a collection of values and rules that can guide you in analyzing a problem, designing a solution, implementing and testing the solution, and ultimately deploying the software so stakeholders can use it.

6.1.1 Principles That Guide Process

In Part One of this book we discussed the importance of the software process and described several process models that have been proposed for software engineering work. Every project is unique, and every team is unique. That means you must adapt your process to best fit your needs. Regardless of the process model your team adopts, it contains elements of the generic process framework we described in Chapter 1. The following set of core principles can be applied to this framework and to every software process. A simplified view of this framework is shown in Figure 6.1.

Principle 1. ***Be agile.*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.

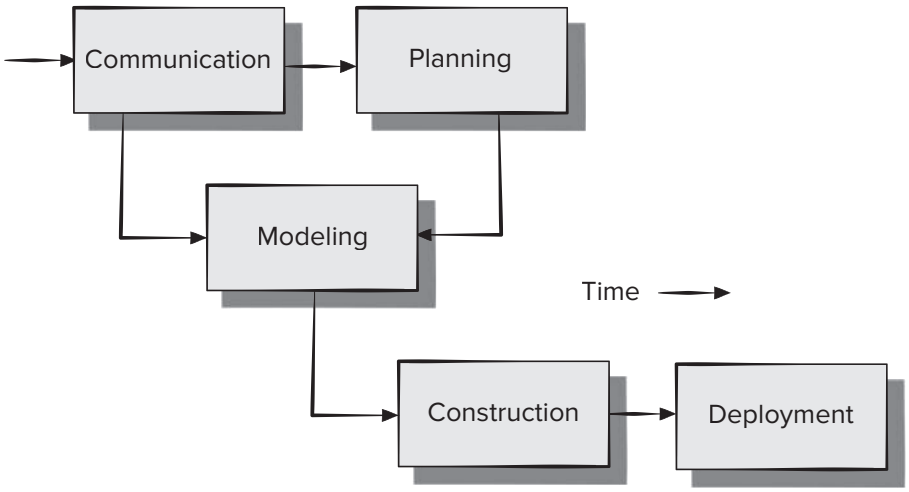
Principle 2. ***Focus on quality at every step.*** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle 3. ***Be ready to adapt.*** Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

1 Some writers argue for one of these terms to the exclusion of the others. In reality, software engineering is all three.

FIGURE 6.1

Simplified process framework



Principle 4. ***Build an effective team.*** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.[2]

Principle 5. ***Establish mechanisms for communication and coordination.*** Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues, and they must be addressed.

Principle 6. ***Manage change.*** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

Principle 7. ***Assess risk.*** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans. Some of these contingency plans will form the basis for security engineering tasks (Chapter 18).

Principle 8. ***Create work products that provide value for others.*** Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. Be sure that the work product imparts the necessary information without ambiguity or omission.

Part Four of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

6.1.2 Principles That Guide Practice

Software engineering practice has a single overriding goal: *to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders*. To achieve this goal, you should adopt a set of core principles that guide your technical work. These principles have merit regardless of the analysis

2 The characteristics of effective software teams were discussed in Chapter 5.

and design methods that you apply, the construction techniques (e.g., programming language, automated tools) that you use, or the verification and validation approach that you choose. The following set of core principles is fundamental to the practice of software engineering:

Principle 1. ***Divide and conquer.*** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoCs). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*).

Principle 2. ***Understand the use of abstraction.*** At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. When we use the abstraction *spreadsheet*, it is assumed that you understand what a spreadsheet is, the general structure of content that a spreadsheet presents, and the typical functions that can be applied to it. In software engineering practice, you use many different levels of abstraction, each imparting or implying meaning that must be communicated. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a *column* or the *SUM* function).

Principle 3. ***Strive for consistency.*** Whether it's creating an analysis model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a mobile app. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons help to create a highly effective user experience.

Principle 4. ***Focus on the transfer of information.*** Software is about information transfer—from a database to an end user, from a legacy system to a WebApp, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another—the list is almost endless. In every case, information flows across an interface, and this means there are opportunities for errors, omissions, or ambiguity. The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.

Principle 5. ***Build software that exhibits effective modularity.*** Separation of concerns (Principle 1) establishes a philosophy for software. *Modularity* provides a mechanism for realizing the philosophy. Any complex system can be divided into modules (components), but good software engineering practice demands more. Modularity must be *effective*. That is, each module should focus exclusively on one well-constrained aspect of the system. Additionally, modules should be interconnected in a relatively simple manner to other modules, to data sources, and to other environmental aspects.

Principle 6. ***Look for patterns.*** Software engineers use patterns as a means of cataloging and reusing solutions to problems they have encountered in the past. The use of these design patterns can be applied to wider systems engineering and systems integration problems, by allowing components in complex systems to evolve independently. Patterns will be discussed further in Chapter 14.

Principle 7. ***When possible, represent the problem and its solution from several different perspectives.*** When a problem and its solution are examined from different

perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered. The unified modeling language (UML) provides a means of describing a problem solution from multiple viewpoints, as described in Appendix 1.

Principle 8. ***Remember that someone will maintain the software.*** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

These principles are not all you'll need to build high-quality software, but they do establish a foundation for every software engineering method discussed in this book.

6.2 Principles That Guide Each Framework Activity

In the sections that follow, we consider principles that have a strong bearing on the success of each generic framework activity defined as part of the software process. In many cases, the principles that are discussed for each of the framework activities are a refinement of the principles presented in Section 6.1. They are simply core principles stated at a lower level of abstraction.

6.2.1 Communication Principles

Before customer requirements can be analyzed, modeled, or specified, they must be gathered through the communication activity. A customer has a problem that may be amenable to a computer-based solution. You respond to the customer's request for help. Communication has begun. But the road from communication to understanding is often full of potholes.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that you will confront. There are many ways to communicate, but it's important to recognize that not all are equal in richness or effectiveness (Figure 6.2). In this context, we discuss communication principles as they apply to customer communication. However, many of the principles apply equally to all forms of communication that occur within a software project.

INFO

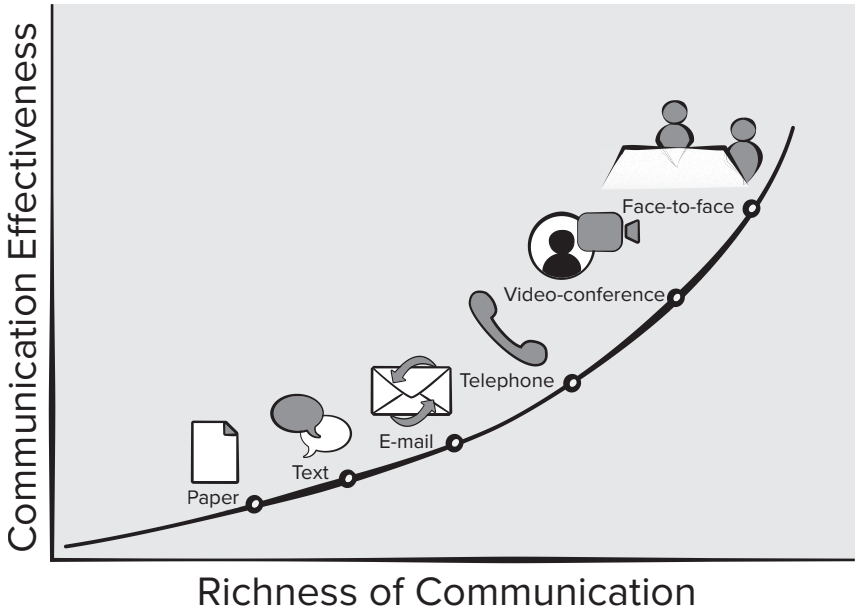
The Difference Between Customers and End Users

A *customer* is the person or group who (1) originally requested the software to be built, (2) defines overall business objectives for the software, (3) provides basic product requirements, and (4) coordinates funding for the project. In a product or system business, the customer is often the marketing group. In an information technology (IT) environment, the customer might be a business component or department.

An *end user* is the person or group who (1) will actually use the software that is built to achieve some business purpose and (2) will define operational details of the software so the business purpose can be achieved. In some cases, the customer and the end user may be one and the same, but for many projects that is not the case.

FIGURE 6.2

Effectiveness of communication modes



Principle 1. ***Listen.*** Before communicating, be sure you understand the point of view of the other party, know a bit about his or her needs, and then listen. Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, and avoid constant interruptions. *Never* become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2. ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3. ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure that other principles are followed.

Principle 4. ***Face-to-face communication is best.*** However, this communication usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

Principle 5. ***Take notes and document decisions.*** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle 6. ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle 7. ***Stay focused; modularize your discussion.*** The more people are involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved (however, see Principle 9).

Principle 8. ***If something is unclear, draw a picture.*** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

Principle 9. ***(a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified right now, move on.*** Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle 10. ***Negotiation is not a contest or a game. It works best when both parties win.*** There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

SAFEHOME

Communication Mistakes

The scene: Software engineering team workspace

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member.

The conversation:

Ed: What have you heard about this *SafeHome* project?

Vinod: The kickoff meeting is scheduled for next week.

Jamie: I've already done a little bit of investigation, but it didn't go well.

Ed: What do you mean?

Jamie: Well, I gave Lisa Perez a call. She's the marketing honcho on this thing.

Vinod: And . . . ?

Jamie: I wanted her to tell me about *SafeHome* features and functions . . . that sort of thing. Instead, she began asking me questions about security systems, surveillance systems . . . I'm no expert.

Vinod: What does that tell you?

(Jamie shrugs)

Vinod: That marketing will need us to act as consultants and that we'd better do some homework on this product area before our kickoff meeting. Doug said that he wanted us to "collaborate" with our customer, so we'd better learn how to do that.

Ed: Probably would have been better to stop by her office. Phone calls just don't work as well for this sort of thing.

Jamie: You're both right. We've got to get our act together or our early communications will be a struggle.

Vinod: I saw Doug reading a book on "requirements engineering." I'll bet that lists some principles of good communication. I'm going to borrow it from him.

Jamie: Good idea . . . then you can teach us.

Vinod (smiling): Yeah, right.

6.2.2 Planning Principles

The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.

Try as we might, it's impossible to predict exactly how a software project will evolve. There is no easy way to determine what unforeseen technical problems will be encountered, what important information will remain undiscovered until late in the project, what misunderstandings will occur, or what business issues will change. And yet, a good software team must plan its approach. Often planning is iterative (Figure 6.3).

There are different planning philosophies.[3] Some people are "minimalists," arguing that change often obviates the need for a detailed plan. Others are "traditionalists," arguing that the plan provides an effective road map and the more detail it has, the less likely the team will become lost.

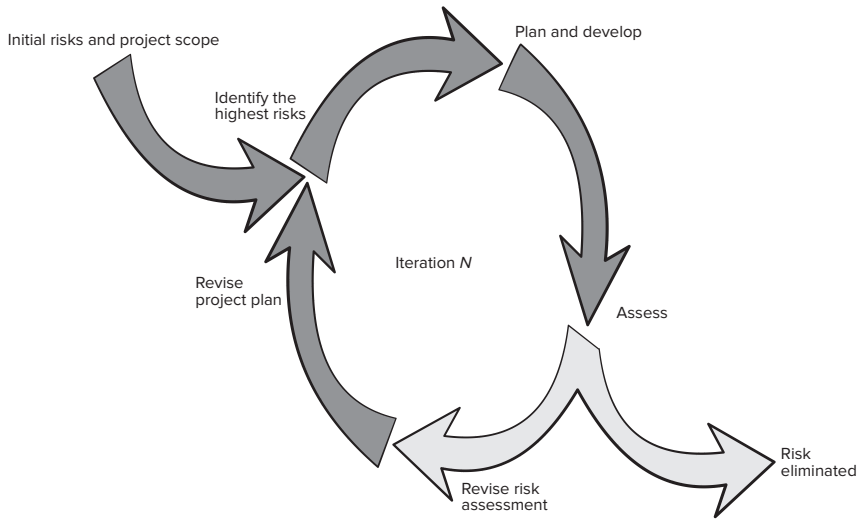
What to do? On many projects, overplanning is time consuming and fruitless (too many things change), but underplanning is a recipe for chaos. Like most things in life, planning should be agile and conducted in moderation, enough to provide useful guidance for the team—no more, no less. Regardless of the rigor with which planning is conducted, the following principles always apply:

Principle 1. ***Understand the scope of the project.*** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

Principle 2. ***Involve stakeholders in the planning activity.*** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

FIGURE 6.3

Iterative planning



3 A detailed discussion of software project planning and management is presented in Part Four of this book.

Principle 3. ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it is very likely that things will change. The plan will need to be adjusted. Iterative, incremental process models include time for revising plans after the delivery of each software increment based on feedback received from users.

Principle 4. ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle 5. ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary. The project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

Principle 6. ***Be realistic.*** People don't work 100 percent of every day. Changes will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

Principle 7. ***Adjust granularity as you define the plan.*** The term *granularity* refers to the detail with which some element of planning is represented or conducted. A *high-granularity* plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A *low-granularity* plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Activities that won't occur for many months do not require high granularity (too much can change).

Principle 8. ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality. If technical reviews[4] are to be conducted, they should be scheduled. If pair programming (Chapter 3) is to be used during construction, it should be explicitly defined within the plan.

Principle 9. ***Describe how you intend to accommodate change.*** Uncontrolled change can obviate even the best planning. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle 10. ***Track the plan frequently, and make adjustments as required.*** Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress daily, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

To be most effective, everyone on the software team should participate in the planning activity. Only then will team members "sign up" to the plan.

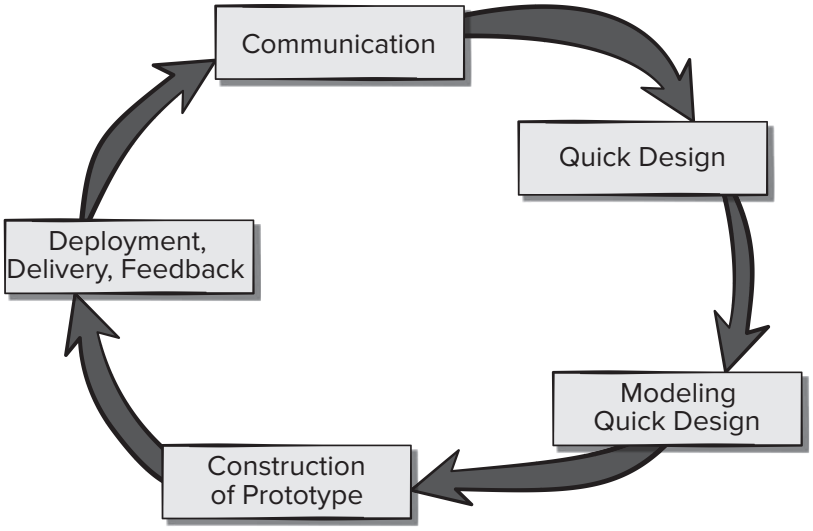
6.2.3 Modeling Principles

We create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a

4 Technical reviews are discussed in Chapter 16.

FIGURE 6.4

Role of software modeling



three-dimensional (3D) model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that users desire, and the behavior of the system as the transformation is taking place. Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level. Figure 6.4 shows how modeling may be used in agile software design.

In software engineering work, two classes of models can be created: requirements models and design models. *Requirements models* (also called *analysis models*) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain (Chapter 8). *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail (Chapters 9 through 12).

In their book on agile modeling, Scott Ambler and Ron Jeffries [Amb02] define a set of modeling principles[5] that are intended for those who use an agile process model (Chapter 3) but are appropriate for all software engineers who perform modeling action and tasks:

Principle 1. ***The primary goal of the software team is to build software, not create models.*** Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

5 The principles noted in this section have been abbreviated and rephrased for the purposes of this book.

Principle 2. ***Travel light—don't create more models than you need.*** Every model that is created must be kept up to date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

Principle 3. ***Strive to produce the simplest model that will describe the problem or the software.*** Don't overbuild the software [Amb02]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. ***Build models in a way that makes them amenable to change.*** Assume that your models will change, but in making this assumption don't get sloppy. The problem with this attitude is that you may not create a reasonably complete requirements model, which means you'll create a design (design model) that will invariably miss important functions and features.

Principle 5. ***Be able to state an explicit purpose for each model that is created.*** Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

Principle 6. ***Adapt the models you develop to the system at hand.*** It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software for adaptive cruise control in an automobile.

Principle 7. ***Try to build useful models, but forget about building perfect models.*** When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make a model that is complete and internally consistent may not be worth the benefits of these properties. Iterating endlessly to make a model "perfect" does not serve the need for agility.

Principle 8. ***Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.*** Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

Principle 9. ***If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.*** If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

Principle 10. ***Get feedback as soon as you can.*** The intent of any model is to communicate information. It should stand on its own. Assume that you won't be there to explain the model. Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

6.2.4 Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user.

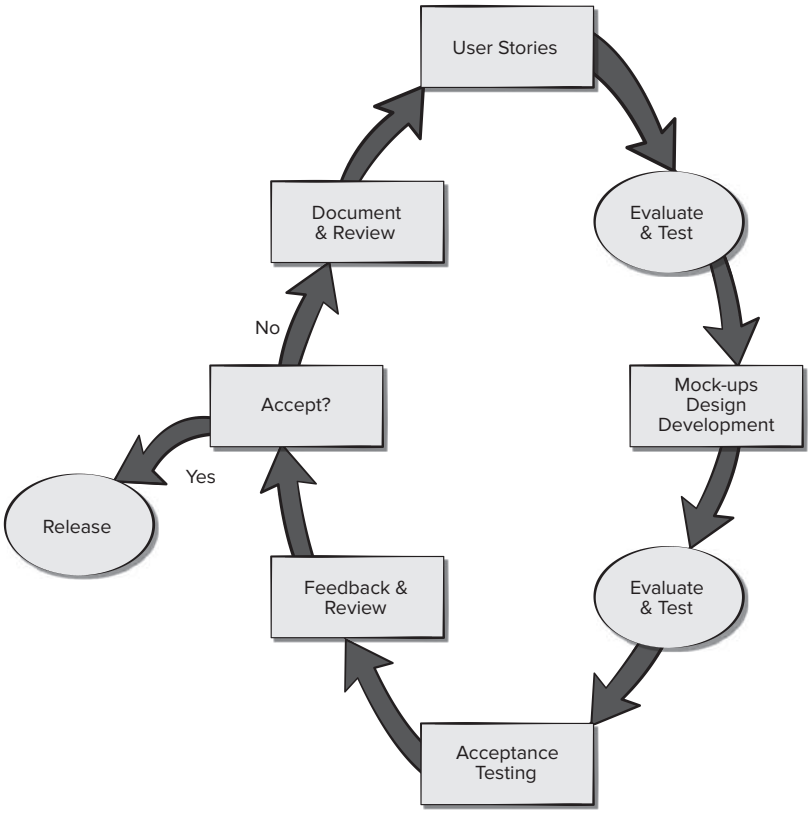
In modern software engineering work, coding may be: (1) the direct creation of programming language source code, (2) the automatic generation of source code using an intermediate design like representation of the component to be built, or (3) the automatic generation of executable code using a fourth-generation programming language (e.g., Unreal4 *Blueprints*).[6]

The initial focus of testing is at the component level, often called *unit testing*. Other levels of testing include (1) *integration testing* (conducted as the system is constructed), (2) *validation testing* that assesses whether requirements have been met for the complete system (or software increment), and (3) *acceptance testing* that is conducted by the customer in an effort to exercise all required features and functions. Figure 6.5 shows where testing and test case design is placed in agile processes.

Testing is considered in detail in Chapters 19 through 21. The following set of fundamental principles and concepts are applicable to coding and testing.

FIGURE 6.5

Testing in agile processes



6 *Blueprints* is a visual scripting tool created by Epic Games (https://docs.unrealengine.com/latest/INT/Engine/Blueprints/).

Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. There are some fundamental principles that might be followed:

Preparation Principles: Before you write one line of code, be sure you:

Principle 1. ***Understand the problem you're trying to solve.***

Principle 2. ***Understand basic design principles and concepts.***

Principle 3. ***Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.***

Principle 4. ***Select a programming environment that provides tools that will make your work easier.***

Principle 5. ***Create a set of unit tests that will be applied once the component you code is completed.***

Coding Principles: As you begin writing code, be sure you:

Principle 6. ***Constrain your algorithms by following structured programming [Boh00] practice.***

Principle 7. ***Consider the use of pair programming.***

Principle 8. ***Select data structures that will meet the needs of the design.***

Principle 9. ***Understand the software architecture and create interfaces that are consistent with it.***

Validation Principles: After you've completed your first coding pass, be sure you:

Principle 10. ***Conduct a code walkthrough when appropriate.***

Principle 11. ***Perform unit tests and correct errors you've uncovered.***

Principle 12. ***Refactor the code to improve its quality.***

Testing Principles. In a classic book on software testing, Glen Myers [Mye79] states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint for some software developers. They move counter to the commonly held view that a successful test is one in which no errors are found. Your objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

As a secondary benefit, testing demonstrates that software functions appear to be working according to specification and that behavioral and performance requirements appear to have been met. In addition, the data collected as testing is conducted provide a good indication of software reliability and some indication of software quality.

FIGURE 6.6

Testing is never complete

Testing cannot show the absence of errors and defects; it can show only that software errors and defects are present (Figure 6.6). It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

Davis [Dav95b] suggests a set of testing principles[7] that have been adapted for use in this book. In addition, Everett and Meyer [Eve09] suggest additional principles:

Principle 1. ***All tests should be traceable to customer requirements.***[8] The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. ***Tests should be planned long before testing begins.*** Test planning (Chapter 19) can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

Principle 3. ***The Pareto principle applies to software testing.*** In this context, the Pareto principle implies that 80 percent of all errors uncovered during testing will

7 Only a small subset of Davis's testing principles are noted here. For more information, see [Dav95b].

8 This principle refers to *functional tests*, that is, tests that focus on requirements. *Structural tests* (tests that focus on architectural or logical detail) may not address specific requirements directly.

likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. ***Testing should begin "in the small" and progress toward testing "in the large."*** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts to looking for errors in integrated clusters of components and ultimately in the entire system.

Principle 5. ***Exhaustive testing is not possible.*** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Principle 6. ***Apply to each module in the system a testing effort commensurate with its expected fault density.*** These are often the newest modules or the ones that are least understood by the developers.

Principle 7. ***Static testing techniques can yield high results.*** More than 85 percent of software defects originate in the software documentation (requirements, specifications, code walk-throughs, and user manuals) [Jon91]. There may be value in testing the system documentation.

Principle 8. ***Track defects and look for patterns in defects uncovered by testing.*** The total defects uncovered is a good indicator of software quality. The types of defects uncovered can be a good measure of software stability. Patterns of defects found over time can forecast numbers of expected defects.

Principle 9. ***Include test cases that demonstrate software is behaving correctly.*** As software components are being maintained or adapted, unexpected interactions cause unintended side effects in other components. It is important to have a set of regression test cases (Chapter 19) ready to check system behavior after changes are made to a software product.

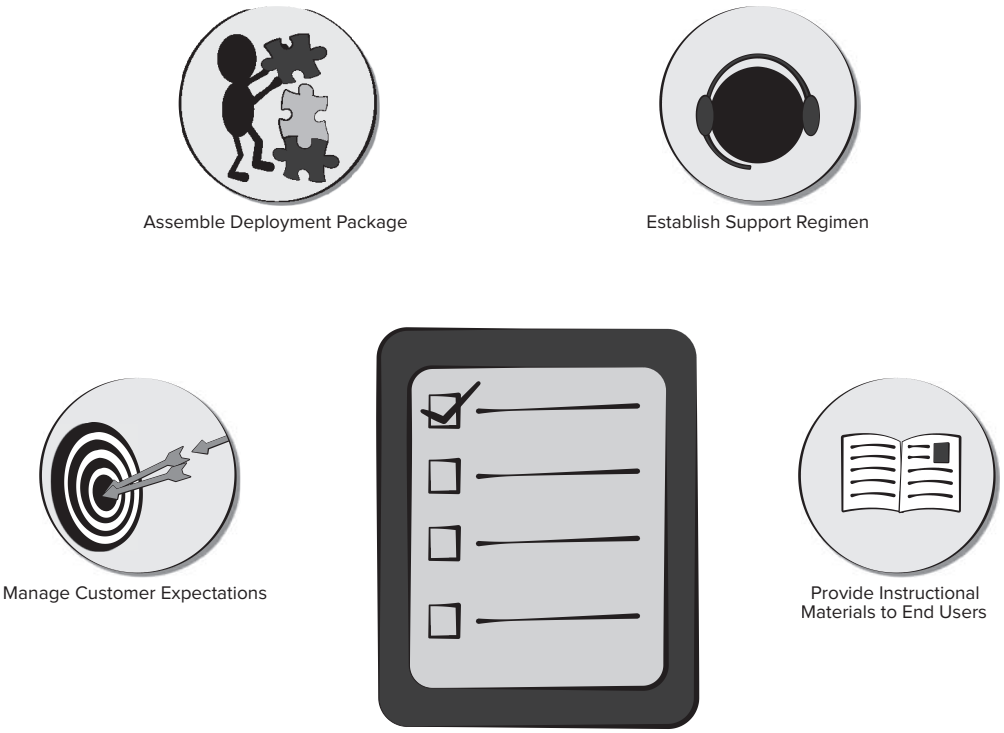
6.2.5 Deployment Principles

As we noted in Part One of this book, the deployment activity encompasses three actions: delivery, support, and feedback. Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but several times as software moves toward completion. Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features. Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date. Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment. Typical deployment actions are illustrated in Figure 6.7.

The delivery of a software increment represents an important milestone for any software project. Some key principles should be followed as the team prepares to deliver an increment:

Principle 1. ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment

FIGURE 6.7
Deployment actions



occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [Kar94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time period provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2. ***A complete delivery package should be assembled and tested.*** All executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in all possible computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements).

Principle 3. ***A support regimen must be established before the software is delivered.*** An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate record-keeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle 4. ***Appropriate instructional materials must be provided to end users.*** The software team delivers more than the software itself. Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided,

and when necessary, a "what's different about this software increment" description should be published.[9]

Principle 5. ***Buggy software should be fixed first, delivered later.*** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

6.3 SUMMARY

Software engineering practice encompasses principles, concepts, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different. Yet, a set of generic principles applies to the process and to the practice of each framework activity regardless of the project or the product.

A set of core principles helps in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it navigates through the software process. At the practice level, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles provide guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modeling principles serve as a foundation for the methods and notation that are used to create representations of the software. Modeling encompasses both analysis and design, describing representations of the software that progressively become more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested. Coding principles define generic actions that should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: Testing is a process of executing a program with the intent of finding an error.

Deployment occurs as each software increment is presented to the customer and encompasses delivery, support, and feedback. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support

9 During the communication activity, the software team should determine what types of help materials users want.

information for the software. Support demands advance preparation. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.

Problems and Points to Ponder

6.1. Because a focus on quality demands resources and time, is it possible to be agile and still maintain a quality focus?

6.2. Of the eight core principles that guide process (discussed in Section 6.1.1), which do you believe is most important?

6.3. Describe the concept of *separation of concerns* in your own words.

6.4. Why is it necessary to "move on"?

6.5. Do some research on "negotiation" for the communication activity, and prepare a set of guidelines that focus solely on negotiation.

6.6. Why are models important in software engineering work? Are they always necessary? Are there qualifiers to your answer about necessity?

6.7. What is a successful test?

6.8. Why is feedback important to the software team?