CHAPTER

1

SOFTWARE AND SOFTWARE ENGINEERING

KEY CONCEPTS

He had the classic look of a senior executive for a major software company—mid-40s, slightly graying at the temples, trim and athletic, with eyes that penetrated the listener as he spoke. But what he said shocked me.

"Software is *dead*."

I blinked with surprise and then smiled. "You're joking, right? The world is driven by software and your company has profited handsomely because of it. It isn't dead! It's alive and growing."

He shook his head emphatically. "No, it's dead . . . at least as we once knew it."

I leaned forward. "Go on."

He spoke while tapping the table for emphasis. "The old-school view of software—you buy it, you own it, and it's your job to manage it—that's coming to an end. Today, with Web 2.0 and pervasive computing coming on strong, we're going to be seeing a completely different generation of software. It'll be delivered via the Internet and will look exactly like it's residing on each user's computing device . . . but it'll reside on a far-away server."

QUICK LOOK

What is it? Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media. Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high-quality computer software.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

Why is it important? Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities. Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

What is the work product? From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software. But from the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

How do I ensure that I've done it right? Read the remainder of this book, select those ideas that are applicable to the software that you build, and apply them to your work.

I had to agree. "So, your life will be much simpler. You guys won't have to worry about five different versions of the same App in use across tens of thousands of users."

He smiled. "Absolutely. Only the most current version residing on our servers. When we make a change or a correction, we supply updated functionality and content to every user. Everyone has it instantly!"

I grimaced. "But if you make a mistake, everyone has that instantly as well."

He chuckled. "True, that's why we're redoubling our efforts to do even better software engineering. Problem is, we have to do it 'fast' because the market has accelerated in every application area."

I leaned back and put my hands behind my head. "You know what they say, . . . you can have it fast, you can have it right, or you can have it cheap. Pick two!"

"I'll take it fast and right," he said as he began to get up.

I stood as well. "Then you really do need software engineering."

"I know that," he said as he began to move away. "The problem is, we've got to convince still another generation of techies that it's true!"

Is software *really* dead? If it was, you wouldn't be reading this book!

Quote:

"Ideas and technological discoveries are the driving engines of economic growth."

Wall Street Journal

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Fifty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the printing industry); that software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than almost all industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

No one could foresee that software would become embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial, entertainment, office machines, . . . the list is almost endless. And if you believe the law of unintended consequences, there are many effects that we cannot yet predict.

No one could predict that millions of computer programs would have to be corrected, adapted, and enhanced as time passed. The burden of performing these "maintenance" activities would absorb more people and more resources than all work applied to the creation of new software.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive

to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad-based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering*.

1.1 The Nature of Software

Software is both a product and a vehicle that delivers a product.

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information*. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.

Quote:

"Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets."

Brad J. Cox

The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options, have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone

programmer are the same questions that are asked when modern computer-based systems are built:[1]

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has lead to the adoption of software engineering practice.

1.1.1 Defining Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:

How should we define *software*?

> Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered.

But a more formal definition probably won't measurably improve your understanding. To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

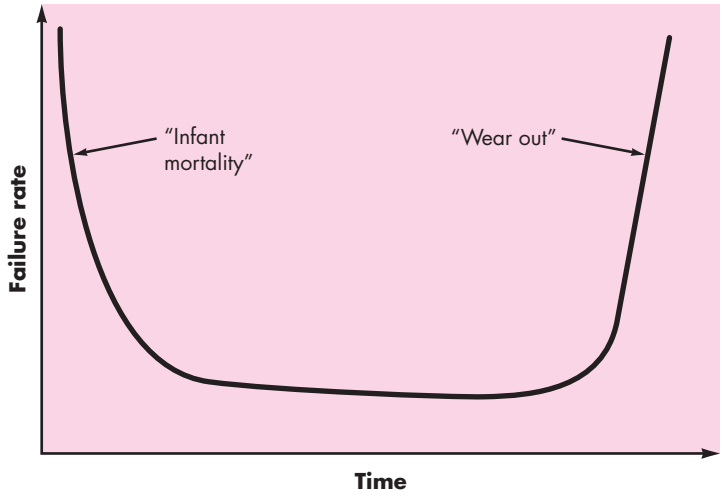
KEY POINT
Software is engineered, not manufactured.

1. *Software is developed or engineered; it is not manufactured in the classical sense.*

   Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent

1 In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin asking 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

FIGURE 1.1
Failure curve for hardware



(or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different (see Chapter 24). Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. *Software doesn't "wear out."*

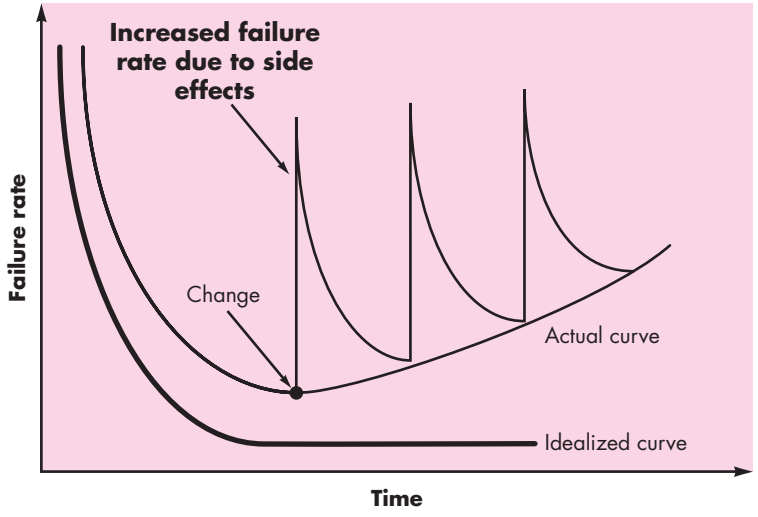
KEY POINT
Software doesn't wear out, but it does deteriorate.

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does *deteriorate!*

ADVICE
If you want to reduce software deterioration, you'll have to do better software design (Chapters 8 to 13).

FIGURE 1.2
Failure curves for software



KEY POINT
Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve in Figure 1.2.

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,[2] software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

Quote:
"Ideas are the building blocks of ideas."
Jason Zebehazy

3. *Although the industry is moving toward component-based construction, most software continues to be custom built.*

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent

2 In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by a variety of different stakeholders.

something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.[3] For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,[4] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

WebRef

One of the most comprehensive libraries of shareware/freeware can be found at **shareware.cnet.com**

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from

3 Component-based development is discussed in Chapter 10.

4 Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

Web applications—called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Quote:

"There is no computer that has common sense."

Marvin Minsky

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is! Past generations of software people have left a legacy in each of the categories I have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden of future software engineers. And yet, new challenges (Chapter 31) have appeared on the horizon:

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

Netsourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Quote:

"You can't always predict, but you can always prepare."

Anonymous

Each of these new challenges will undoubtedly obey the law of unintended consequences and have effects (for businesspeople, software engineers, and end users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and to business rules that are sure to come over the next decade.

1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

? What do I do if I encounter a legacy system that exhibits poor quality?

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.[5] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results

5 In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support "core business functions and are indispensable to the business." What to do?

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

What types of changes are made to legacy systems?

- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.

ADVICE

Every software engineer must recognize that change is natural. Don't try to fight it.

When these modes of evolution occur, a legacy system must be reengineered (Chapter 29) so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other" [Day99].

1.2 The Unique Nature of WebApps

Quote:

"By the time we see any sort of stabilization, the Web will have turned into something completely different."

Louis Monier

In the early days of the World Wide Web (circa 1990 to 1995), *websites* consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications*[6] (I refer to these collectively as *WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

As noted in Section 1.1.2, WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different. Powell [Pow98] suggests that Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between

6 In the context of this book, the term *Web application* (WebApp) encompasses everything from a simple Web page that might help a consumer compute an automobile lease payment to a comprehensive website that provides complete travel services for businesspeople and vacationers. Included within this category are complete websites, specialized functionality within websites, and information processing applications that reside on the Internet or on an Intranet or Extranet.

internal communications and external relations, and between art and technology." The following attributes are encountered in the vast majority of WebApps.

What characteristic differentiates WebApps from other software?

Network intensiveness. A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

Concurrency. A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

Performance. If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

Availability. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

Data driven. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

Content sensitive. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

Continuous evolution. Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

Immediacy. Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.[7]

Security. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

7 With modern tools, sophisticated Web pages can be produced in only a few hours.

of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

Aesthetics. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

It can be argued that other application categories discussed in Section 1.1.2 can exhibit some of the attributes noted. However, WebApps almost always exhibit all of them.

1.3 Software Engineering

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

KEY POINT
Understand the problem before you build a solution.

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application[8] has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*

KEY POINT
Design is a pivotal software engineering activity.

- The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. *It follows that design becomes a pivotal activity.*

KEY POINT
Both quality and maintainability are an outgrowth of good design.

- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*
- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use

8 I will call these people "stakeholders" later in this book.

increase, demands for adaptation and enhancement will also grow. *It follows that software should be maintainable.*

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book—*software engineering.*

Quote:

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

Scott Whitmir

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.

You will be tempted to add to this definition.[9] 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.

The IEEE [IEE93a] 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, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

KEY POINT

Software engineering encompasses a process, methods for managing and engineering software, and tools.

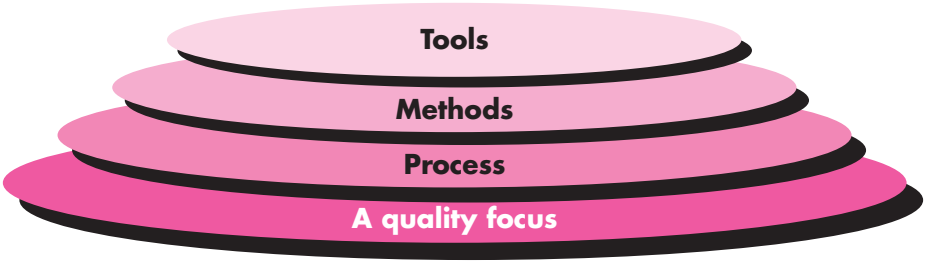
Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies[10] foster a continuous process improvement culture, and 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.

The foundation for software engineering is the *process* layer. The 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

9 For numerous additional definitions of *software engineering,* see www.answers.com/topic/software-engineering#wp-_note-13.

10 Quality management and related approaches are discussed in Chapter 14 and throughout Part 3 of this book.

FIGURE 1.3 Software engineering layers



WebRef

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

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.

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.

1.4 THE SOFTWARE PROCESS

? What are the elements of a software process?

Quote:

"A process defines who is doing *what* *when* and *how* to reach a certain goal."

Ivar Jacobson, Grady Booch, and James Rumbaugh

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A *process framework* establishes the foundation for a complete software engineering 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. A generic process framework for software engineering encompasses five activities:

? What are the five generic process framework activities?

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders[11]) The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing 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. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models 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.

Quote:

"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

These five generic framework activities can be used during the development of small, simple 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.

11 A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. 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.").

For many software projects, framework activities are applied iteratively as a project progresses. That is, **communication, planning, modeling, construction,** and **deployment** are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of *umbrella activities*. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

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

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

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

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

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

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' 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.

Each of these umbrella activities is discussed in detail later in this book.

KEY POINT
Software process adaptation is essential for project success.

Earlier in this section, I noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

? **How do process models differ from one another?**

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and tasks are defined within each framework activity
- Degree to which work products are identified and required

Quote:

"I feel a recipe is only a theme which an intelligent cook can play each time with a variation."

Madame Benoit

- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor with which the process is described
- Degree to which the customer and other stakeholders are involved with the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

In Part 1 of this book, I'll examine software process in considerable detail. *Prescriptive process models* (Chapter 2) stress detailed definition, identification, and application of process activities and tasks. Their intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and 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 all stakeholders.

? What characterizes an "agile" process?

Agile process models (Chapter 3) emphasize project "agility" and follow a set of principles that lead to a more informal (but, proponents argue, no less effective) approach to software process. These process models are generally characterized as "agile" because they emphasize maneuverability and adaptability. They are appropriate for many types of projects and are particularly useful when Web applications are engineered.

1.5 Software Engineering Practice

WebRef

A variety of thought-provoking quotes on the practice of software engineering can be found at **www.literateprogramming.com**

In Section 1.4, I introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.[12]

1.5.1 The Essence of Practice

ADVICE

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

In a classic book, *How to Solve It,* written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).

12 You should revisit relevant sections within this chapter as specific software engineering methods and umbrella activities are discussed later in this book.

3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing.* Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Quote:

"There is a grain of discovery in the solution of any problem."

George Polya

Plan the solution. Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry out the plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

1.5.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book I'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:[13]

Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is

13 Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

Quote:

"There is a certain majesty in simplicity which is far above all the quaintness of wit."

Alexander Pope (1688–1744)

not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume*

KEY POINT

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[14] This could very possibly lead to the reuse of an entire system.

14 This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[15] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

1.6 Software Myths

Quote:

"In the absence of meaningful standards, a new industry like software comes to depend instead on folklore."

Tom DeMarco

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

15 Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

WebRef

The Software Project Managers Network at **www.spmn.com** can help you dispel these and other myths.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived

iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.[16] However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Whenever you think, we don't have time for software engineering, ask yourself, "Will we have time to do it over again?"

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

16 Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 3.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a "legacy system" to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

SafeHome[17]

How a Project Starts

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

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

17 The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are purely fictitious, but the situations and problems are real.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* product line. The engineering effort will succeed only if *SafeHome* software succeeds. The market will accept the product only if the software embedded within it properly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

1.8 Summary

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these WebApps have unique features and requirements, they are software nonetheless.

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem solving activity that follows a set of core principles.

A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies required to build it grows. As you learn more about software engineering, you'll begin to understand why these myths should be debunked whenever they are encountered.

Problems and Points to Ponder

1.1. Provide at least five additional examples of how the law of unintended consequences applies to computer software.

1.2. Provide a number of examples (both positive and negative) that indicate the impact of software on our society.

1.3. Develop your own answers to the five questions asked at the beginning of Section 1.1. Discuss them with your fellow students.

1.4. Many modern applications change frequently—before they are presented to the end user and then after the first version has been put into use. Suggest a few ways to build software to stop deterioration due to change.

1.5. Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.

1.6. Figure 1.3 places the three software engineering layers on top of a layer entitled "a quality focus." This implies an organizational 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.

1.7. Is software engineering applicable when WebApps are built? If so, how might it be modified to accommodate the unique characteristics of WebApps?

1.8. As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm (either economic or human).

1.9. 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.

1.10. 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.

1.11. Add two additional myths to the list presented in Section 1.6. Also state the reality that accompanies the myth.

Further Readings and Information Sources[18]

There are literally thousands of books written about computer software. The vast majority discuss programming languages or software applications, but a few discuss software itself. Pressman and Herron (*Software Shock,* Dorset House, 1991) presented an early discussion (directed at the layperson) of software and the way professionals build it. Negroponte's best-selling book (*Being Digital*, Alfred A. Knopf, Inc., 1995) provides a view of computing and its overall impact in the twenty-first century. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software and the process through which it is developed.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do,* McGraw-Hill, 2000) argues that the "modern scourge" of software bugs can be eliminated and suggests ways to accomplish this. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth,* MIT Press, 2001) argues that the "divide" between those who have access to information resources (e.g., the Web) and those that do not is narrowing as we move into the first decade of this century. Books by Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing,* New Riders Publishing, 2006) and Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications,* Auerbach, 2006) introduce the concept of "open-world" software and predict a wireless environment in which software must adapt to requirements that emerge in real time.

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

18 The *Further Reading and Information Sources* section presented at the conclusion of each chapter presents a brief overview of print sources that can help to expand your understanding of the major topics presented in the chapter. I have created a comprehensive website to support *Software Engineering: A Practitioner's Approach at* **www.mhhe.com/compsci/pressman**. Among the many topics addressed within the website are chapter-by-chapter software engineering resources to Web-based information that can complement the material presented in each chapter. An Amazon.com link to every book noted in this section is contained within these resources.

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.* Tens of thousands of websites 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.

Andersson, E., et al., *Software Engineering for Internet Applications,* The MIT Press, 2006.

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.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2d ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice,* 3d ed., Prentice-Hall, 2005.

Schach, S., *Object-Oriented and Classical Software Engineering,* 7th ed., McGraw-Hill, 2006.

Sommerville, I., *Software Engineering,* 8th ed., Addison-Wesley, 2006.

Tsui, F., and O. Karam, *Essentials of Software Engineering,* Jones & Bartlett Publishers, 2006.

Many software engineering standards have been published by the IEEE, ISO, and their standards organizations over the past few decades. Moore (*The Road Map to Software Engineering: A Standards-Based Guide,* Wiley-IEEE Computer Society Press, 2006) provides a useful survey of relevant standards and how they apply to real projects.

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 website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.**

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

KEY CONCEPTS

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 social 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 work products 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 website.

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, I define a *software process* as a framework for the activities, actions, and 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 so that it is appropriate for the products that they build and the demands of their marketplace.

2.1 A Generic Process Model

In Chapter 1, a process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside 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.

KEY POINT

The hierarchy of technical work within the software process is activities, encompassing actions, populated by tasks.

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

uote:

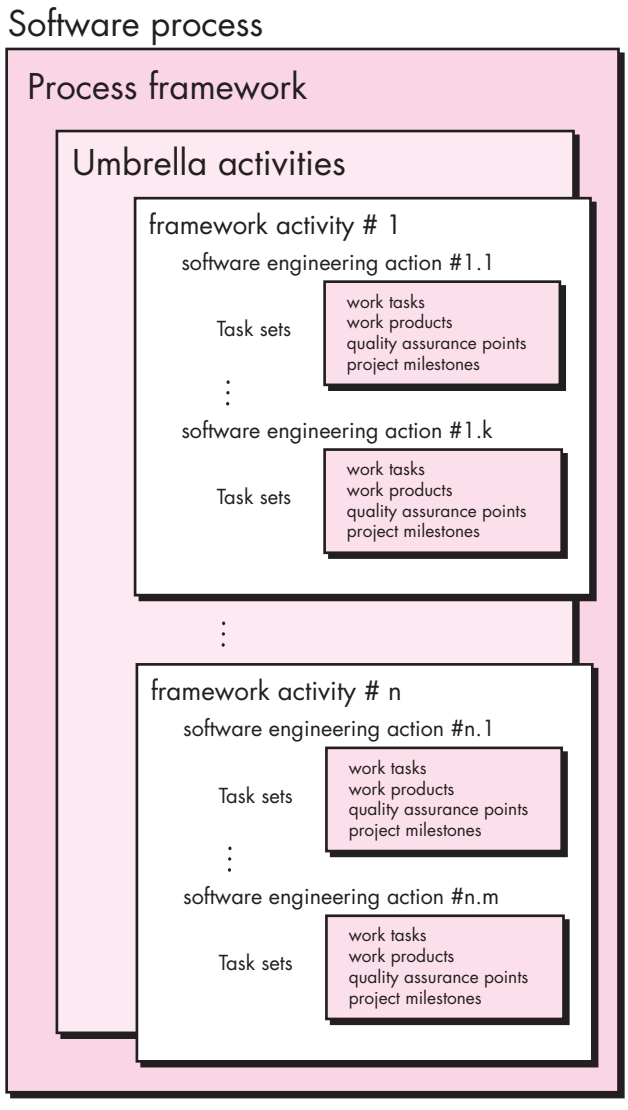
"We think that software developers are missing a vital truth: most organizations don't know what they do. They think they know, but they don't know."

Tom DeMarco

You should note that one important aspect of the software process has not yet been discussed. 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 and is illustrated in Figure 2.2.

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).

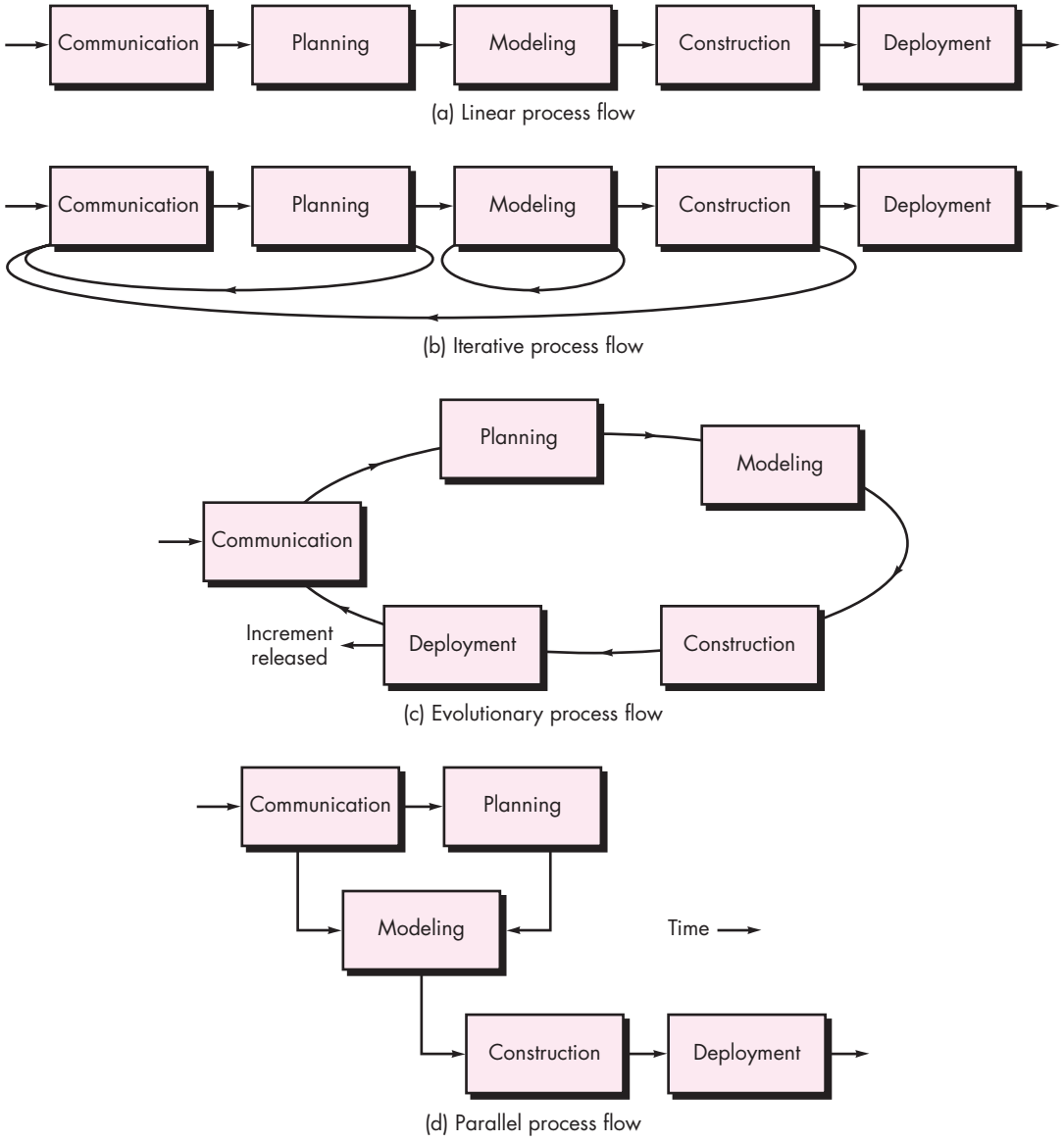
FIGURE 2.1
A software process framework



2.1.1 Defining a Framework Activity

Although I 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?*

FIGURE 2.2 Process flow



? How does a framework activity change as the nature of the project changes?

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 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 take 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 many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions (described in Chapter 5): *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

KEY POINT

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

2.1.2 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. You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action 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, *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 formality. The software team chooses the task set that will allow it to achieve the goal of each action and still maintain quality and agility.

2.1.3 Process Patterns

What is a process pattern?

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern*[1] describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template [Amb98]—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Quote:

"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.[2] In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

KEY POINT

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

Ambler [Amb98] has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specified. Ambler [Amb98] suggests three types:

1. *Stage pattern*—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.
2. *Task pattern*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).
3. *Phase pattern*—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**[3]

1 A detailed discussion of patterns is presented in Chapter 12.

2 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.

3 These phase patterns are discussed in Section 2.3.3.

Initial context. Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists?

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 **Communication** pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of 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. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: **ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription,** and **ScenarioCreation.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern). The description is then refined into a set of stage patterns that describe framework activities and are further refined in a 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.

WebRef

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

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. RequirementsUnclear

Intent. This pattern describes an approach for building 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 software 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 to be solved, 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 would be presented here and is described later in Section 2.3.3.

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: **CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.**

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

2.2 Process Assessment and Improvement

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 (Chapters 14 and 16). Process patterns must be coupled with solid software engineering practice (Part 2 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 a successful software engineering.[4]

What formal techniques are available for assessing the software process?

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

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

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

Quote:

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

NASA

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

SPICE (ISO/IEC15504)—a standard that 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 [ISO08].

ISO 9001:2000 for Software—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 [Ant06].

A more detailed discussion of software assessment and process improvement methods is presented in Chapter 30.

2.3 Prescriptive Process Models

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

Quote:

"If the process is right, the results will take care of themselves."

Takashi Osada

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.

The philosophical implications of this argument are significant for software engineering. If prescriptive process models[5] strive for structure and order, are they inappropriate for a software world that thrives on change? Yet, if we reject traditional process

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

KEY POINT

Prescriptive process models define a prescribed set of process elements and a predictable process work flow.

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, I examine the prescriptive process approach in which order and project consistency are dominant issues. I 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.

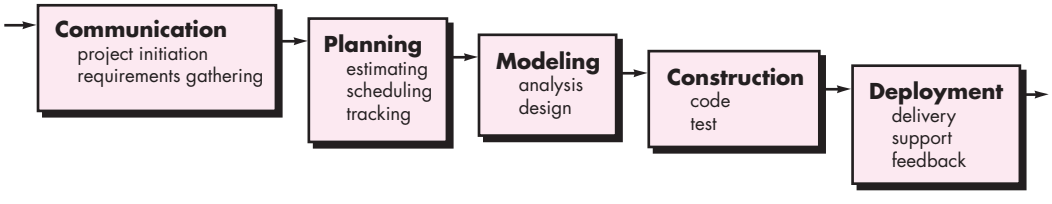
2.3.1 The Waterfall Model

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 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[6] to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3).

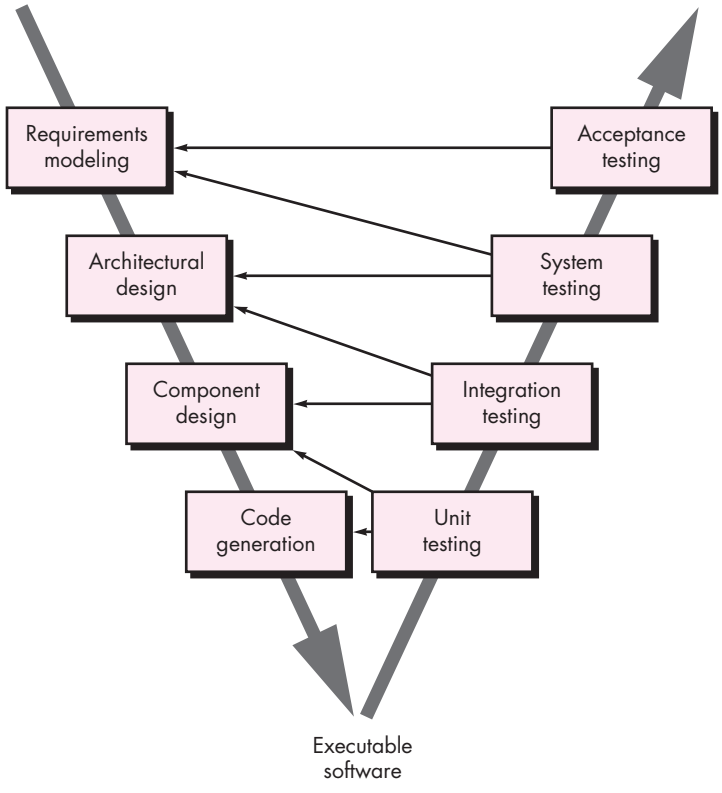
A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 2.4, the V-model [Buc99] depicts the relationship of quality

FIGURE 2.3 The waterfall model



6 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 2.4
The V-model



KEY POINT
The V-model illustrates how verification and validation actions are associated with earlier engineering actions.

assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.[7] In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. However, over the past three 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.

7 A detailed discussion of quality assurance actions is presented in Part 3 of this book.

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.

Quote:

"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

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle 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 states tend 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.

2.3.2 Incremental Process Models

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.

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, you can choose a process model that is designed to produce the software in increments.

The *incremental* model combines elements of linear and parallel process flows discussed in Section 2.1. Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow (Section 2.3.3).

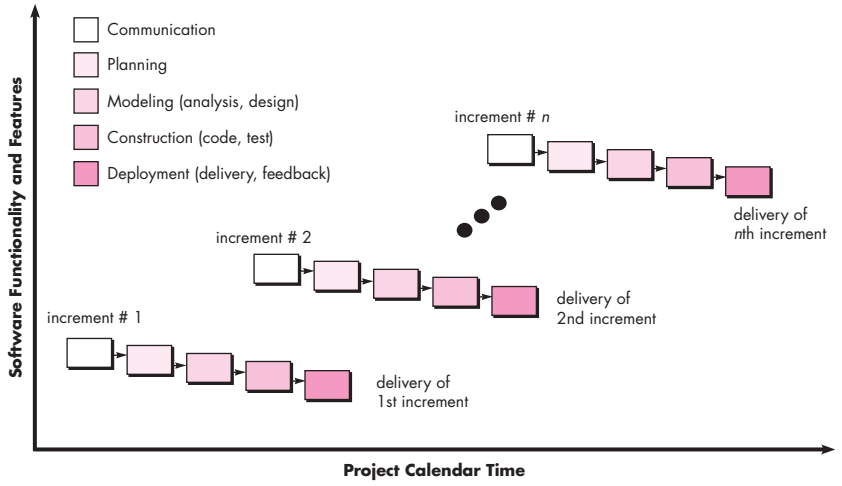
ADVICE

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.

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 can incorporate the prototyping paradigm.

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 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

FIGURE 2.5
The incremental model



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.

The incremental process 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 provide a platform for evaluation by the user.[8]

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, then 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.

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

2.3.3 Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. 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

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

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, you 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 you to develop increasingly more complete versions of the software. In the paragraphs that follow, I present two common evolutionary process models.

Quote:

"Plan to throw one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers."

Frederick P. Brooks

Prototyping. 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.

ADVICE

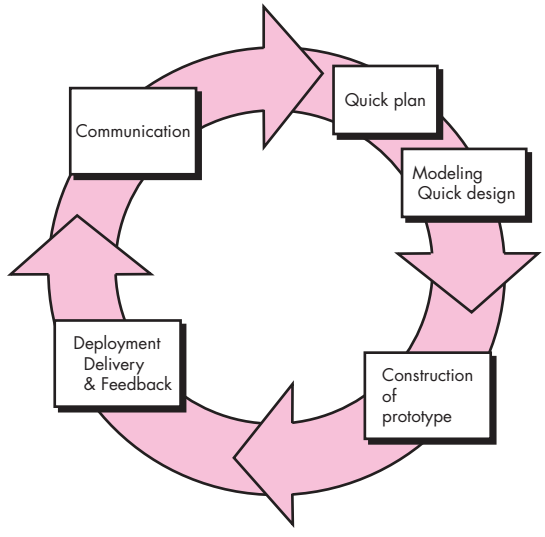
When your customer has a legitimate need, but is clueless about the details, develop a prototype as a first step.

Although prototyping can be used as a 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.

The prototyping paradigm (Figure 2.6) 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

FIGURE 2.6

The prototyping paradigm



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 (e.g., report generators and window managers) that enable working programs to be generated quickly.

But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] 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.

The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system.

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:

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

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you 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, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. As a software engineer, you often make 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, you 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.

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 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 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 this chapter 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 OK. 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.

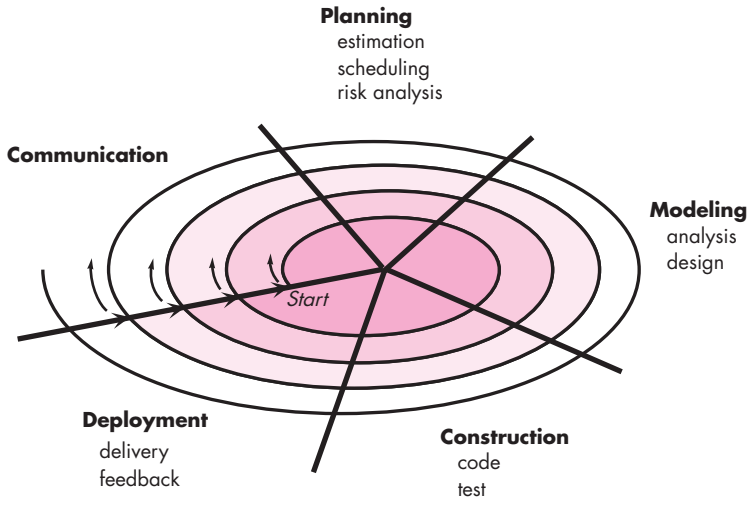
Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

The Spiral Model. 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. Boehm [Boe01a] 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 model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

FIGURE 2.7
A typical spiral model



KEY POINT
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, I use the generic framework activities discussed earlier.[9] Each of the framework activities represent one segment of the spiral path illustrated in Figure 2.7. 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 28) 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 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/publications/documents/00.reports/00sr008.html**.

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" that starts at the core of the spiral and continues for multiple iterations[10] until concept

9 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].

10 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.

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).

ADVICE

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

Quote:

"I'm only this far and only tomorrow leads my way."

Dave Matthews Band

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.

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, 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.

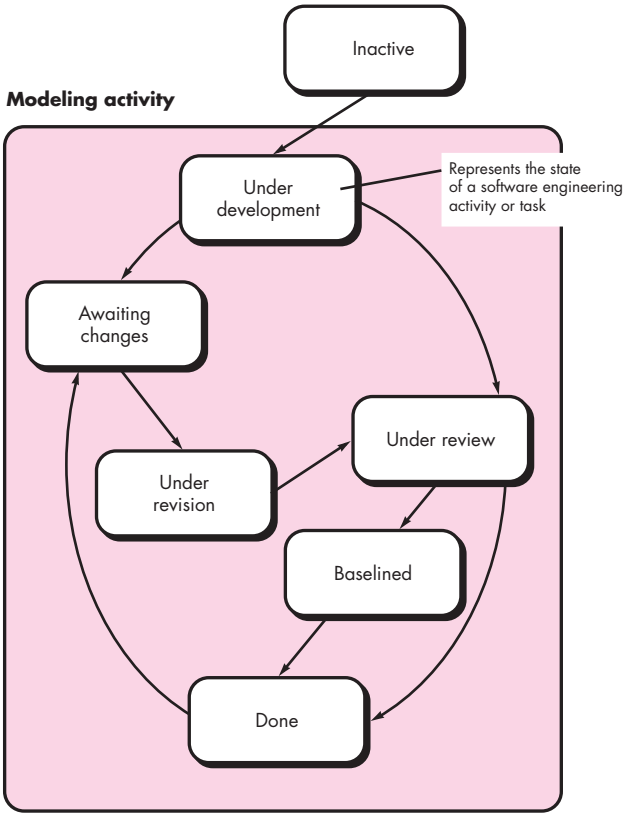
Lee: Wait a minute. Did you say that we regenerate the plan with each tour around the spiral, Doug? That's not so great; we need one plan, one schedule, and we've got to stick to it.

Doug: That's old-school thinking, Lee. Like 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.

FIGURE 2.8
One element of the concurrent process model



2.3.4 Concurrent Models

The *concurrent development model,* sometimes called *concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.[11]

The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—**modeling**—may be in any one of the states[12] noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

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

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

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 **inactive** 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.

Concurrent modeling 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 major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event *analysis model correction,* which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

Quote:

"Every process in your organization has a customer, and without a customer a process has no purpose."

V. Daniel Hunt

Concurrent modeling 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 process network. 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.

2.3.5 A Final Word on Evolutionary Processes

I 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.[13]

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] :

> 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 . . .

13 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).

> 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[14] 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).

2.4 Specialized Process Models

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.[15]

2.4.1 Component-Based Development

WebRef

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

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* 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 component-based development model constructs 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[16] of classes. Regardless of the

14 In this context software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Chapters 14 and 16.

15 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.

16 Object-oriented concepts are discussed in Appendix 2 and are used throughout Part 2 of this book. In this context, a class encompasses 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.

technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture. Component-based development is discussed in more detail in Chapter 10.

2.4.2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you 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.

When formal methods (Chapter 21) 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 you 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 that would suffer severe economic hardship should software errors occur.

2.4.3 Aspect-Oriented Software Development

WebRef

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

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).

KEY POINT

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

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 an 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].

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 evolutionary and concurrent process models. The evolutionary model 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. If you have further interest, see [Saf08], [Cla05], [Jac04], and [Gra03].

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 road map as software engineers do technical work and a template for managers who must track and control the software process.

Representative Tools:[17]

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.

ProVision BPMx, developed by Proforma (**www.proformacorp.com**), is representative of many tools that assist in process definition and workflow automation.

A worthwhile listing of many different tools associated with the software process can be found at **www.processwave.net/Links/tool_links.htm**.

2.5 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 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

17 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.

(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[18]). 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.

2.5.1 A Brief History

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 object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development.

UML is used throughout Part 2 of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial for those who are unfamiliar with basic UML notation and modeling rules. A comprehensive presentation of UML is best left to textbooks dedicated to the subject. Recommended books are listed in Appendix 1.

UML provided the necessary technology to support object-oriented software engineering practice, but it did 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 (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

2.5.2 Phases of the Unified Process[19]

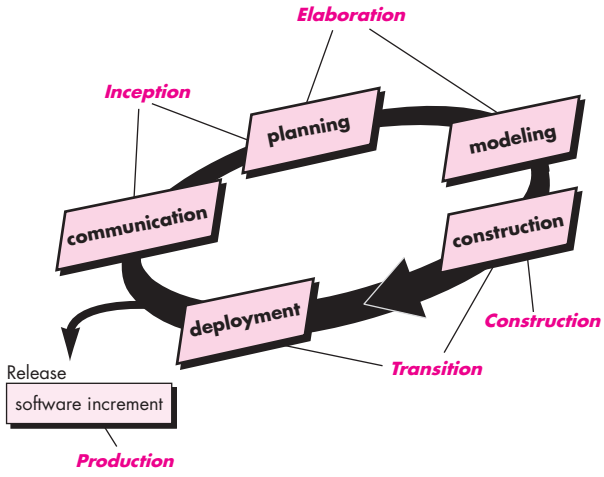
Earlier in this chapter, I discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process is no exception. Figure 2.9 depicts the "phases" of the UP and relates them to the generic activities that have been discussed in Chapter 1 and earlier in this chapter.

---

18 A *use case* (Chapter 5) 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 requirements model.

19 The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Corporation (subsequently acquired by IBM), an early contributor to the development and refinement of the UP and a builder of complete environments (tools and technology) that support the process.

FIGURE 2.9
The Unified Process



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

The *inception phase* of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, 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 (Chapter 5) that describe which features and functions each major class of users desires. 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 communication and modeling activities of the generic process model (Figure 2.9). 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 requirements 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.[20] 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 are often made at this time.

20 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.

WebRef

An interesting discussion of the UP in the context of agile development can be found at **www.ambysoft.com/unifiedprocess/agileUP.html**.

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 use case operational for end users. To accomplish this, requirements 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 for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests[21] 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 is given to end users for beta testing and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, 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 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.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (described earlier in this chapter). 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.

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

21 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 17 through 20.

Quote:

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

Dexter Yager

the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can 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.[22]

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 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 model defines five framework activities:

WebRef

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

Planning. This activity isolates requirements and 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.

? What framework activities are used during PSP?

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 21) 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 (this is 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.

22 It's worth noting the proponents of agile software development (Chapter 3) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

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

PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.

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 entirely, 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 (IPTs) 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[23] Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

ADVICE
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; 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.

23 The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30.

TSP defines the following framework activities: **project 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 system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

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

TSP recognizes that the best software teams are self-directed.[24] Team members set project objectives, adapt the process to meet their needs, control the project 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

One or more of the process models discussed in the preceding sections must be adapted for use by a software 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.

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Section 2.1. 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 activities, actions, and 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 software engineering tools that are appropriate for a particular work task.

24 In Chapter 3 I discuss the importance of "self-organizing" teams as a key element in agile software development.

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:[25]

Igrafx Process Tools—tools that enable a team to map, measure, and model the software process (**www.micrografx.com**)

Adeptia BPM Server—designed to manage, automate, and optimize business processes (**www.adeptia.com**)

SpeedDev Suite—a collection of six tools with a heavy emphasis on the management of communication and modeling activities (**www.speedev.com**)

2.8 PRODUCT AND PROCESS

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. 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 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

25 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.

> 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. As 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.

2.9 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 and V models, 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 concurrent process model allows a software team to represent iterative and concurrent elements of any process model. Specialized models include the component-based model that emphasizes component reuse and assembly; the formal methods model that encourages a mathematically based approach to software development and verification; and the aspect-oriented model that accommodates crosscutting concerns spanning 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.

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.

Problems and Points to Ponder

2.1. 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, (d) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

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

2.3. A common problem during **communication** occurs when you encounter two stakeholders who have conflicting ideas about what the software should be. That is, you have mutually conflicting requirements. Develop a process pattern (this would be a stage pattern) using the template presented in Section 2.1.3 that addresses this problem and suggest an effective approach to it.

2.4. Do some research on PSP and present a brief presentation that describes the types of measurements that an individual software engineer is asked to make and how those measurement can be used to improve personal effectiveness.

2.5. 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.

2.6. Read [Nog00] and write a two- or three-page paper that discusses the impact of "chaos" on software engineering.

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

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

2.9. What process adaptations are required if the prototype will evolve into a deliverable system or product?

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

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

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

2.13. The concurrent process model defines a set of "states." Describe what these states represent in your own words, and then indicate how they come into play within the concurrent process model.

2.14. 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?

2.15. Provide three examples of software projects that would be amenable to the component-based model. Be specific.

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

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

Further Readings and Information Sources

Most software engineering textbooks consider traditional process models in some detail. Books by Sommerville (*Software Engineering,* 8th ed., Addison-Wesley, 2006), Pfleeger and Atlee (*Software Engineering,* 3d ed., Prentice-Hall, 2005), and Schach (*Object-Oriented and Classical Software Engineering,* 7th ed., McGraw-Hill, 2006) consider traditional paradigms and discuss their strengths and weaknesses. Glass (*Facts and Fallacies of Software Engineering,* Prentice-Hall, 2002) provides an unvarnished, pragmatic view of the software engineering process. Although not specifically dedicated to process, Brooks (*The Mythical Man-Month,* 2d ed., Addison-Wesley, 1995) presents age-old project wisdom that has everything to do with process.

Firesmith and Henderson-Sellers (*The OPEN Process Framework: An Introduction,* Addison-Wesley, 2001) present a general template for creating "flexible, yet discipline software processes" and discuss process attributes and objectives. Madachy (*Software Process Dynamics,* Wiley-IEEE, 2008) discusses modeling techniques that allow the interrelated technical and social elements of the software process to be analyzed. Sharpe and McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development,* Artech House, 2001) present tools for modeling both software and business processes.

Lim (*Managing Software Reuse,* Prentice Hall, 2004) discusses reuse from a manager's perspective. Ezran, Morisio, and Tully (*Practical Software Reuse,* Springer, 2002) and Jacobson, Griss, and Jonsson (*Software Reuse,* Addison-Wesley, 1997) present much useful information on component-based development. Heineman and Council (*Component-Based Software Engineering,* Addison-Wesley, 2001) describe the process required to implement component-based systems. Kenett and Baker (*Software Process Quality: Management and Control,* Marcel Dekker, 1999) consider how quality management and process design are intimately connected to one another.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007) and Richardson and Gwaltney (*Ship it! A Practical Guide to Successful Software Projects,* Pragmatic Bookshelf, 2005) present a broad collection of useful guidelines that are applicable to the deployment activity.

In addition to Jacobson, Rumbaugh, and Booch's seminal book on the Unified Process [Jac99], books by Arlow and Neustadt (*UML 2 and the Unified Process,* Addison-Wesley, 2005), Kroll and Kruchten (*The Rational Unified Process Made Easy,* Addison-Wesley, 2003), and Farve (*UML and the Unified Process,* IRM Press, 2003) provide excellent complementary information. Gibbs (*Project Management with the IBM Rational Unified Process,* IBM Press, 2006) discusses project management within the context of the UP.

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 website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

3

AGILE DEVELOPMENT

KEY CONCEPTS

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01a] (referred to as the "Agile Alliance") signed the "Manifesto for Agile Software Development." It stated:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> *Individuals and interactions* over processes and tools
>
> *Working software* over comprehensive documentation
>
> *Customer collaboration* over contract negotiation
>
> *Responding to change* over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

QUICK LOOK

What is it? Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

Who does it? Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

Why is it important? The modern business environment that spawns computer-based systems and software products is fast-paced and ever-changing. Agile software engineering represents a reasonable alternative to conventional software engineering for certain classes of software and certain types of software projects. It has been demonstrated to deliver successful systems quickly.

What are the steps? Agile development might best be termed "software engineering lite." The basic framework activities—communication, planning, modeling, construction, and deployment—remain. But they morph into a minimal task set that pushes the project team toward construction and delivery (some would argue that this is done at the expense of problem analysis and solution design).

What is the work product? Both the customer and the software engineer have the same view—the only really important work product is an operational "software increment" that is delivered to the customer on the appropriate commitment date.

How do I ensure that I've done it right? If the agile team agrees that the process works, and the team produces deliverable software increments that satisfy the customer, you've done it right.

A manifesto is normally associated with an emerging political movement—one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that's exactly what agile development is all about.

Although the underlying ideas that guide agile development have been with us for many years, it has been less than two decades since these ideas have crystallized into a "movement." In essence, agile[1] methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a Web-based application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.

Fluidity implies change, and change is expensive. Particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change throughout the software process.

Does this mean that a recognition of challenges posed by modern realities causes you to discard valuable software engineering principles, concepts, methods, and tools? Absolutely not! Like all engineering disciplines, software engineering continues to evolve. It can be adapted easily to meet the challenges posed by a demand for agility.

Quote:

"Agility: 1, everything else: 0."

Tom DeMarco

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 2 have a major failing: *they forget the frailties of the people who build computer software.* Software engineers are not robots. They exhibit great variation in working styles; significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. Cockburn argues that process models can "deal with people's common weaknesses with [either] discipline or tolerance" and that most prescriptive process models choose discipline. He states: "Because consistency in action is a human weakness, high discipline methodologies are fragile."

If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows "tolerance" for the people who do software engineering work. Invariably, tolerant practices are easier for software people to adopt and sustain, but (as Cockburn admits) they may be less productive. Like most things in life, trade-offs must be considered.

1 Agile methods are sometimes referred to as *light methods* or *lean methods*.

3.1 What Is Agility?

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] provides a useful discussion:

> *Agility* has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.

Don't make the mistake of assuming that agility gives you license to hack out solutions. A process is required and discipline is essential.

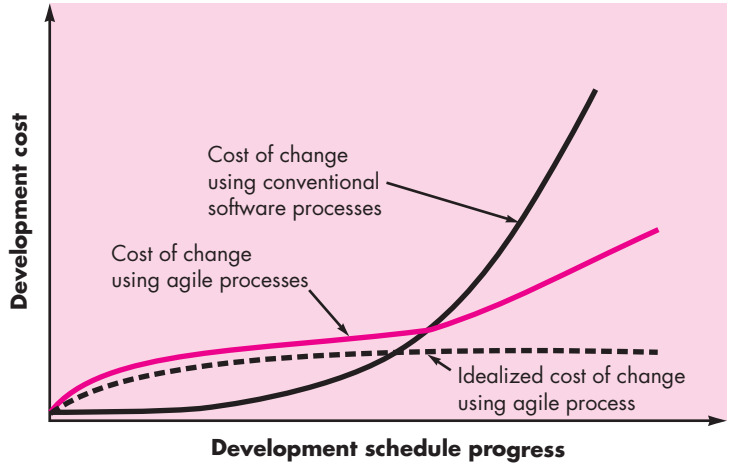
But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

3.2 Agility and the Cost of Change

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 3.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will

FIGURE 3.1
Change costs as a function of time in development



Quote:

"Agility is dynamic, content specific, aggressively change embracing, and growth oriented."

–Steven Goldman et al.

not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

KEY POINT

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed later in this chapter), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

3.3 What Is an Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

WebRef

A comprehensive collection of articles on the agile process can be found at **www.aanpo.org/articles/index**.

2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability?* The answer, as I have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable.*

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally.* To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

3.3.1 Agility Principles

KEY POINT

Although agile processes embrace change, it is still important to examine the reasons for change.

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

ADVICE

Working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability.

9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self–organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

3.3.2 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

You don't have to choose between agility and software engineering. Rather, define a software engineering approach that is agile.

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making.

No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 3.4), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

If you have further interest, see [Hig01], [Hig02a], and [DeM02] for an entertaining summary of other important technical and political issues.

3.3.3 Human Factors

Quote:

"Agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans."

Barry Boehm

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith [Coc01a] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team,* not the other way around.[2]

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

What key traits must exist among the people on an effective software team?

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Quote:

"What counts as barely sufficient for one team is either overly sufficient or insufficient for another."

Alistair Cockburn

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving

2 Successful software engineering organizations recognize this reality regardless of the process model they choose.

activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what DeMarco and Lister [DeM98] call a "jelled" team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts." [DeM98]

Self-organization. In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber [Sch02] addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."

KEY POINT

A self-organizing team is in control of the work it performs. The team makes its own commitments and defines plans to achieve them.

3.4 Extreme Programming (XP)

In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed [Ker05]. IXP refines XP and targets the agile process specifically for use within large organizations.

3.4.1 XP Values

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors[3] for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

3 In the XP context, a *metaphor* is "a story that everyone—customers, programmers, and managers—can tell about how the system works" [Bec04a].

ADVICE

Keep it simple whenever you can, but recognize that continual "refactoring" can absorb significant time and resources.

To achieve *simplicity,* XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored*[4] at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy (Chapters 17 through 20), the software (via test results) provides the agile team with feedback. XP makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* (Chapter 5) that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Quote:

"XP is the answer to the question, 'How little can we do and still build great software?'"

Anonymous

Beck [Bec04a] argues that strict adherence to certain XP practices demands *courage.* A better word might be *discipline.* For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates *respect* among it members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

3.4.2 The XP Process

WebRef

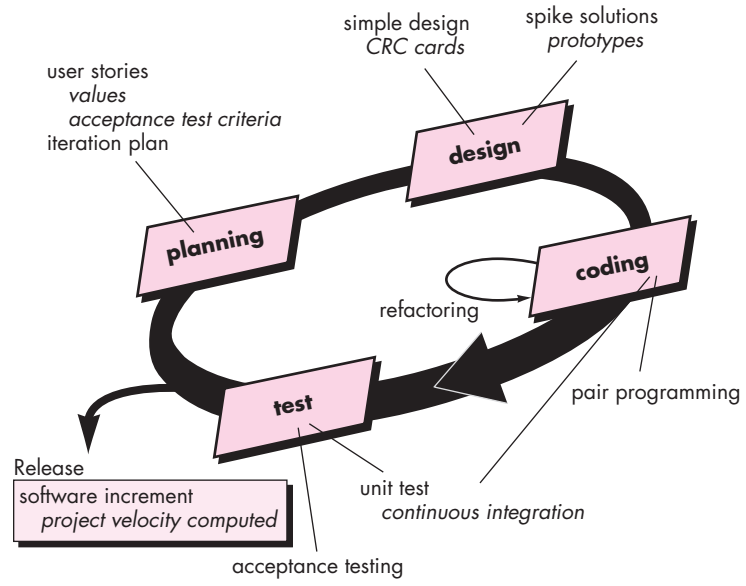
An excellent overview of "rules" for XP can be found at **www.extremeprogramming.org/rules.html**.

Extreme Programming uses an object-oriented approach (Appendix 2) as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.2 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad

4 Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior. In essence, refactoring can be used to improve the efficiency, readability, or performance of a design or the code that implements a design.

FIGURE 3.2
The Extreme Programming process



feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* (similar to use cases described in Chapter 5) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.[5] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

? What is an XP "story"?

WebRef
A worthwhile XP "planning game" can be found at: **c2.com/cgi/wiki?planningGame**.

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the

5 The value of a story may also be dependent on the presence of another story.

Project velocity is a subtle measure of team productivity.

number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an overcommitment has been made for all stories across the entire development project. If an overcommitment occurs, the content of releases is modified or end delivery dates are changed.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[6]

XP deemphasizes the importance of design. Not everyone agrees. In fact, there are times when design should be emphasized.

XP encourages the use of CRC cards (Chapter 7) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes[7] that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 8. The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution*, the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

WebRef

Refactoring techniques and tools can be found at: **www.refactoring.com**.

In the preceding section, we noted that XP encourages *refactoring*—a construction technique that is also a method for design optimization. Fowler [Fow00] describes refactoring in the following manner:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

Because XP design uses virtually no notation and produces few, if any, work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes

6 These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

7 Object-oriented classes are discussed in Appendix 2, in Chapter 8, and throughout Part 2 of this book.

that "can radically improve the design" [Fow00]. It should be noted, however, that the effort required for refactoring can grow dramatically as the size of an application grows.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

KEY POINT
Refactoring improves the internal structure of a design (or source code) without changing its external functionality or behavior.

Coding. After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).[8] Once the unit test[9] has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

WebRef
Useful information on XP can be obtained at **www.xprogramming.com**.

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

What is pair programming?

ADVICE
Many software teams are populated by individualists. You'll have to work to change that culture if pair programming is to work effectively.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment (Chapter 17) that helps to uncover errors early.

Testing. I have already noted that the creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 17) whenever code is modified (which is often, given the XP refactoring philosophy).

How are unit tests used in XP?

8 This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

9 Unit testing, discussed in detail in Chapter 17, focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.

As the individual unit tests are organized into a "universal testing suite" [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

XP *acceptance tests*, also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

KEY POINT
XP acceptance tests are derived from user stories.

3.4.3 Industrial XP

Joshua Kerievsky [Ker05] describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

? What new practices are appended to XP to create IXP?

Readiness assessment. Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.

Quote:
"Ability is what you're capable of doing. Motivation determines what you do. Attitude determines how well you do it."
Lou Holtz

Project community. Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project" [Ker05]. In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the

organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

Test-driven management. An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" [Ker05] and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. An IXP team conducts a specialized technical review (Chapter 15) after a software increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" [Ker05] across a software increment and/or the entire software release. The intent is to improve the IXP process.

Continuous learning. Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

In addition to the six new practices discussed, IXP modifies a number of existing XP practices. *Story-driven development* (SDD) insists that stories for acceptance tests be written before a single line of code is generated. *Domain-driven design* (DDD) is an improvement on the "system metaphor" concept used in XP. DDD [Eva03] suggests the evolutionary creation of a domain model that "accurately represents how domain experts think about their subject" [Ker05]. *Pairing* extends the XP pair-programming concept to include managers and other stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development. *Iterative usability* discourages front-loaded interface design in favor of usability design that evolves as software increments are delivered and users' interaction with the software is studied.

IXP makes smaller modifications to other XP practices and redefines certain roles and responsibilities to make them more amenable to significant projects for large organizations. For further discussion of IXP, visit **http://industrialxp.org**.

3.4.4 The XP Debate

All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP, Stephens and Rosenberg [Ste03] argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic. The authors suggest that the codependent nature of XP practices are both its strength and its weakness. Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process. Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been

addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are:[10]

What are some of the issues that lead to an XP debate?

- *Requirements volatility*. Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.
- *Conflicting customer needs*. Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.
- *Requirements are expressed informally*. User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.
- *Lack of formal design*. XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

You should note that every software process has flaws and that many software organizations have used XP successfully. The key is to recognize where a process may have weaknesses and to adapt it to the specific needs of your organization.

SAFEHOME

Considering Agile Software Development

The scene: Doug Miller's office.

The Players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation:

(A knock on the door, Jamie and Vinod enter Doug's office)

Jamie: Doug, you got a minute?

10 For a detailed look at some thoughtful criticism that has been leveled at XP, visit www.softwarereality.com/ExtremeProgramming.jsp.

Doug: Sure Jamie, what's up?

Jamie: We've been thinking about our process discussion yesterday . . . you know, what process we're going to choose for this new *SafeHome* project.

Doug: And?

Vinod: I was talking to a friend at another company, and he was telling me about Extreme Programming. It's an agile process model . . . heard of it?

Doug: Yeah, some good, some bad.

Jamie: Well, it sounds pretty good to us. Lets you develop software really fast, uses something called pair programming to do real-time quality checks . . . it's pretty cool, I think.

Doug: It does have a lot of really good ideas. I like the pair-programming concept, for instance, and the idea that stakeholders should be part of the team.

Jamie: Huh? You mean that marketing will work on the project team with us?

Doug (nodding): They're a stakeholder, aren't they?

Jamie: Jeez . . . they'll be requesting changes every five minutes.

Vinod: Not necessarily. My friend said that there are ways to "embrace" changes during an XP project.

Doug: So you guys think we should use XP?

Jamie: It's definitely worth considering.

Doug: I agree. And even if we choose an incremental model as our approach, there's no reason why we can't incorporate much of what XP has to offer.

Vinod: Doug, before you said "some good, some bad." What was the "bad"?

Doug: The thing I don't like is the way XP downplays analysis and design . . . sort of says that writing code is where the action is . . .

(The team members look at one another and smile.)

Doug: So you agree with the XP approach?

Jamie (speaking for both): Writing code is what we do, Boss!

Doug (laughing): True, but I'd like to see you spend a little less time coding and then recoding and a little more time analyzing what has to be done and designing a solution that works.

Vinod: Maybe we can have it both ways, agility with a little discipline.

Doug: I think we can, Vinod. In fact, I'm sure of it.

3.5 Other Agile Process Models

Quote:

"Our profession goes through methodologies like a 14-year-old goes through clothing."

Stephen Hawrysh and Jim Ruprecht

The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.[11]

As I noted in the last section, the most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)

11 This is not a bad thing. Before one or more models or methods are accepted as a de facto standard, all must contend for the hearts and minds of software engineers. The "winners" evolve into best practice, while the "losers" either disappear or merge with the winning models.

- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

In the sections that follow, I present a very brief overview of each of these agile process models. It is important to note that *all* agile process models conform (to a greater or lesser degree) to the *Manifesto for Agile Software Development* and the principles noted in Section 3.3.1. For additional detail, refer to the references noted in each subsection or for a survey, examine the "agile software development" entry in Wikipedia.[12]

3.5.1 Adaptive Software Development (ASD)

WebRef

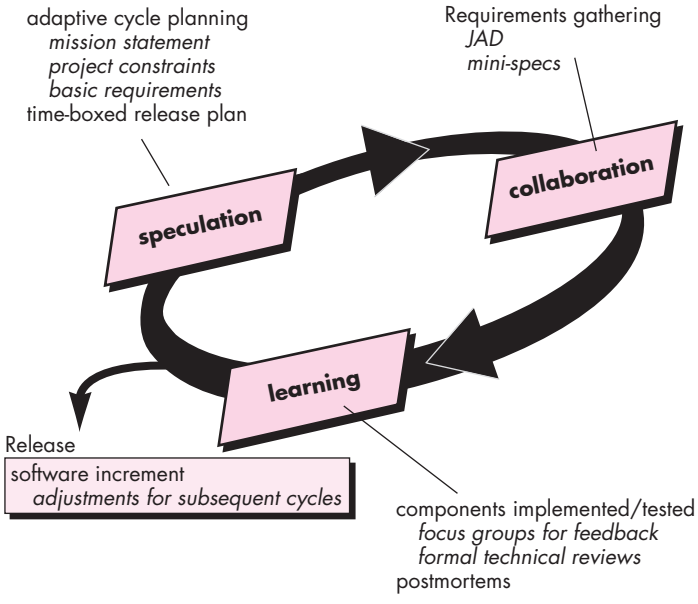
Useful resources for ASD can be found at **www.adaptivesd.com**.

Adaptive Software Development (ASD) has been proposed by Jim Highsmith [Hig00] as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

Highsmith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" (Figure 3.3) that incorporates three phases, speculation, collaboration, and learning.

FIGURE 3.3

Adaptive software development



12 See http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

During *speculation,* the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

Effective collaboration with your customer will only occur if you jettison any "us and them" attitudes.

No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

Motivated people use *collaboration* in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

ASD emphasizes learning as a key element in achieving a "self-organizing" team.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews (Chapter 14), and project postmortems.

The ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

3.5.2 Scrum

Scrum (the name is derived from an activity that occurs during a rugby match[13]) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle [Sch01a].

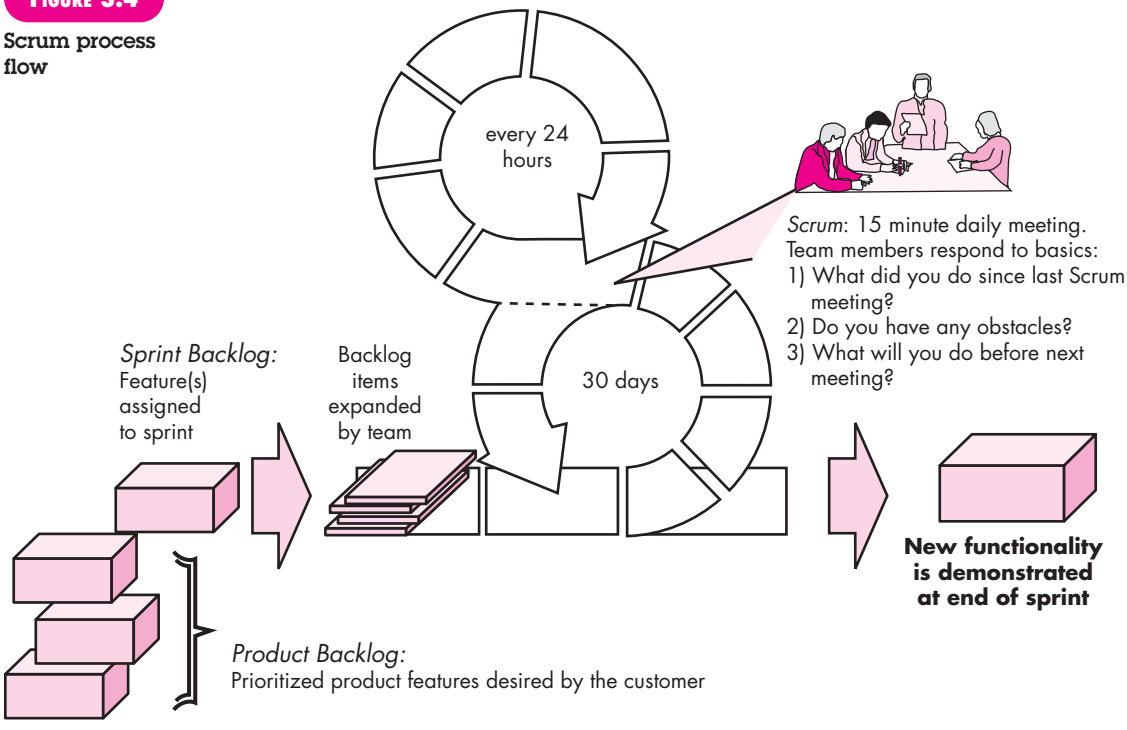
WebRef

Useful Scrum information and resources can be found at **www.controlchaos.com**.

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each

13 A group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

FIGURE 3.4
Scrum process flow



framework activity, work tasks occur within a process pattern (discussed in the following paragraph) called a *sprint.* The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure 3.4.

KEY POINT

Scrum incorporates a set of process patterns that emphasize project priorities, compartmentalized work units, communication, and frequent customer feedback.

Scrum emphasizes the use of a set of software process patterns [Noy02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box[14] (typically 30 days).

14 A *time-box* is a project management term (see Part 4 of this book) that indicates a period of time that has been allocated to accomplish some task.

Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [Noy02]:

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a *Scrum master,* leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Beedle and his colleagues [Bee99] present a comprehensive discussion of these patterns in which they state: "Scrum assumes up-front the existence of chaos. . . . " The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

3.5.3 Dynamic Systems Development Method (DSDM)

WebRef
Useful resources for DSSD can be found at **www.dsdm.org**.

The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" [CCS02]. The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The DSDM Consortium (**www.dsdm.org**) is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the *DSDM life cycle* that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

KEY POINT
DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—revisits prototypes built during *functional model iteration* to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.

Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

DSDM can be combined with XP (Section 3.4) to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

3.5.4 Crystal

Alistair Cockburn [Coc05] and Jim Highsmith [Hig02b] created the *Crystal family of agile methods*[15] in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource-limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game" [Coc02].

KEY POINT
Crystal is a family of process models with the same "genetic code" but different methods for adapting to project characteristics.

To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

15 The name "crystal" is derived from the characteristics of geological crystals, each with its own color, shape, and hardness.

3.5.5 Feature Driven Development (FDD)

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues [Coa99] as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing [Pal02] have extended and improved Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

WebRef
A wide variety of articles and presentations on FDD can be found at: **www.featuredrivendevelopment.com/**.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means. FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits (Chapter 16), the collection of metrics, and the use of patterns (for analysis, design, and construction).

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" [Coa99]. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues [Coa99] suggest the following template for defining a feature:

<action> the **<result>** **<by** | **for** | **of** | **to>** a(n) **<object>**

where an **<object>** is "a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)." Examples of features for an e-commerce application might be:

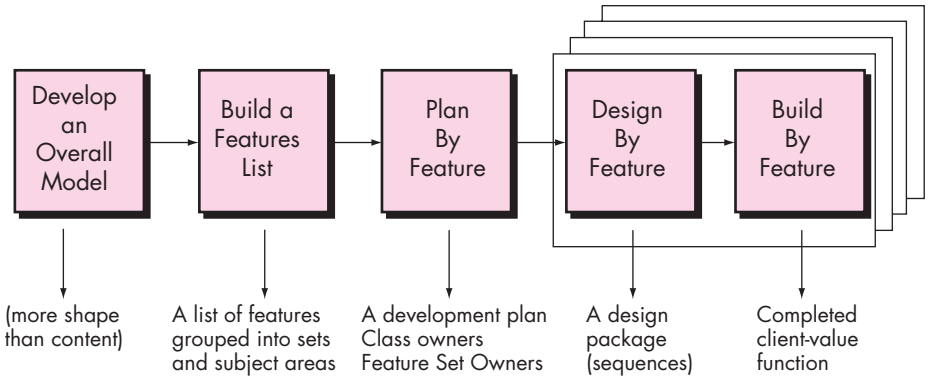
Add the product to shopping cart

Display the technical-specifications of the product

Store the shipping-information for the customer

FIGURE 3.5

Feature Driven Development [Coa99] (with permission)



A feature set groups related features into business-related categories and is defined [Coa99] as:

<action><-ing> a(n) **<object>**

For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.

The FDD approach defines five "collaborating" [Coa99] framework activities (in FDD these are called "processes") as shown in Figure 3.5.

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: "design walkthrough, design, design inspection, code, code inspection, promote to build" [Coa99].

3.5.6 Lean Software Development (LSD)

Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized ([Pop03], [Pop06a]) as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,* and *optimize the whole.*

Each of these principles can be adapted to the software process. For example, *eliminate waste* within the context of an agile software project can be interpreted to mean [Das05]: (1) adding no extraneous features or functions, (2) assessing the cost and schedule impact of any newly requested requirement, (3) removing any superfluous process steps, (4) establishing mechanisms to improve the way team members find information, (5) ensuring the testing finds as many errors as possible,

(6) reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and (7) streamlining the manner in which information is transmitted to all stakeholders involved in the process.

For a detailed discussion of LSD and pragmatic guidelines for implementing the process, you should examine [Pop06a] and [Pop06b].

3.5.7 Agile Modeling (AM)

WebRef

Comprehensive information on agile modeling can be found at: **www.agilemodeling.com**.

There are many situations in which software engineers must build large, business-critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built.

Over the past 30 years, a wide variety of software engineering modeling methods and notation have been proposed for analysis and design (both architectural and component-level). These methods have merit, but they have proven to be difficult to apply and challenging to sustain (over many projects). Part of the problem is the "weight" of these modeling methods. By this I mean the volume of notation required, the degree of formalism suggested, the sheer size of the models for large projects, and the difficulty in maintaining the model(s) as changes occur. Yet analysis and design modeling have substantial benefit for large projects—if for no other reason than to make these projects intellectually manageable. Is there an agile approach to software engineering modeling that might provide an alternative?

Quote:

"I was in the drug store the other day trying to get a cold medication . . . not easy. There's an entire wall of products you need. You stand there going, Well, this one is quick acting but this is long lasting. . . . Which is more important, the present or the future?"

Jerry Seinfeld

At "The Official Agile Modeling Site," Scott Ambler [Amb02a] describes *agile modeling* (AM) in the following manner:

> Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Although AM suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are [Amb02a]:

> **Model with a purpose.** A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

Use multiple models. There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

"Traveling light" is an appropriate philosophy for all software engineering work. Build only those models that provide value … no more, no less.

Travel light. As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler [Amb02a] notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner (hence potentially enhancing communication within your team as well as with project stakeholders)."

Content is more important than representation. Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.

Adapt locally. The modeling approach should be adapted to the needs of the agile team.

A major segment of the software engineering community has adopted the Unified Modeling Language (UML)[16] as the preferred method for representing analysis and design models. The Unified Process (Chapter 2) has been developed to provide a framework for the application of UML. Scott Ambler [Amb06] has developed a simplified version of the UP that integrates his agile modeling philosophy.

3.5.8 Agile Unified Process (AUP)

The *Agile Unified Process* (AUP) adopts a "serial in the large" and "iterative in the small" [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—*inception, elaboration, construction,* and *transition*—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities [Amb06]:

- *Modeling.* UML representations of the business and problem domains are created. However, to stay agile, these models should be "just barely good enough" [Amb06] to allow the team to proceed.

16 A brief tutorial on UML is presented in Appendix 1.

- *Implementation.* Models are translated into source code.
- *Testing.* Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- *Deployment.* Like the generic process activity discussed in Chapters 1 and 2, deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- *Configuration and project management.* In the context of AUP, configuration management (Chapter 22) addresses change management, risk management, and the control of any persistent work products[17] that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.
- *Environment management.* Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Although the AUP has historical and technical connections to the Unified Modeling Language, it is important to note that UML modeling can be using in conjunction with any of the agile process models described in Section 3.5.

SOFTWARE TOOLS

Agile Development

Objective: The objective of agile development tools is to assist in one or more aspects of agile development with an emphasis on facilitating the rapid generation of operational software. These tools can also be used when prescriptive process models (Chapter 2) are applied.

Mechanics: Tool mechanics vary. In general, agile tool sets encompass automated support for project planning, use case development and requirements gathering, rapid design, code generation, and testing.

Representative Tools:[18]

Note: Because agile development is a hot topic, most software tools vendors purport to sell tools that support the agile approach. The tools noted here have characteristics that make them particularly useful for agile projects.

OnTime, developed by Axosoft (**www.axosoft.com**), provides agile process management support for various technical activities within the process.

Ideogramic UML, developed by Ideogramic (**www.ideogramic.com**) is a UML tool set specifically developed for use within an agile process.

Together Tool Set, distributed by Borland (**www.borland.com**), provides a tools suite that supports many technical activities within XP and other agile processes.

17 A *persistent work product* is a model or document or test case produced by the team that will be kept for an indeterminate period of time. It will *not* be discarded once the software increment is delivered.

18 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.

3.6 A Tool Set for the Agile Process

The "tool set" that supports agile processes focuses more on people issues than it does on technology issues.

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops."

Because acquiring the right people (hiring), team collaboration, stakeholder communication, and indirect management are key elements in virtually all agile process models, Cockburn argues that "tools" that address these issues are critical success factors for agility. For example, a hiring "tool" might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The "fit" can be assessed immediately.

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04]) that provides information and coordination among agile developers. Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by "information radiators" (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or "graphs of tests created versus passed . . . other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)" [Coc04].

Are any of these things really tools? They are, if they facilitate the work performed by an agile team member and enhance the quality of the end product.

3.7 Summary

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile—to define maneuverable, adaptive, lean processes that can accommodate the needs of modern business.

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform, communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity, and

an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Extreme programming (XP) is the most widely used agile process. Organized as four framework activities—planning, design, coding, and testing—XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and then prioritized by stakeholders.

Other agile process models also stress human collaboration and team self-organization, but define their own framework activities and select different points of emphasis. For example, ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirement gathering methods, and an iterative development cycle that incorporates customer focus groups and formal technical reviews as real-time feedback mechanisms. Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project. The Dynamic Systems Development Method (DSDM) advocates the use of time-box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment. Crystal is a family of agile process models that can be adopted to the specific characteristics of a project.

Feature Driven Development (FDD) is somewhat more "formal" than other agile methods, but still maintains agility by focusing the project team on the development of features—a client-valued function that can be implemented in two weeks or less. Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. Agile modeling (AM) suggests that modeling is essential for all systems, but that the complexity, type, and size of the model must be tuned to the software to be built. The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building software.

Problems and Points to Ponder

3.1. Reread "The Manifesto for Agile Software Development" at the beginning of this chapter. Can you think of a situation in which one or more of the four "values" could get a software team into trouble?

3.2. Describe agility (for software projects) in your own words.

3.3. Why does an iterative process make it easier to manage change? Is every agile process discussed in this chapter iterative? Is it possible to complete a project in just one iteration and still be agile? Explain your answers.

3.4. Could each of the agile processes be described using the generic framework activities noted in Chapter 2? Build a table that maps the generic activities into the activities defined for each agile process.

3.5. Try to come up with one more "agility principle" that would help a software engineering team become even more maneuverable.

3.6. Select one agility principle noted in Section 3.3.1 and try to determine whether each of the process models presented in this chapter exhibits the principle. [Note: I have presented an overview of these process models only, so it may not be possible to determine whether a principle has been addressed by one or more of the models, unless you do additional research (which is not required for this problem).]

3.7. Why do requirements change so much? After all, don't people know what they want?

3.8. Most agile process models recommend face-to-face communication. Yet today, members of a software team and their customers may be geographically separated from one another. Do you think this implies that geographical separation is something to avoid? Can you think of ways to overcome this problem?

3.9. Write an XP user story that describes the "favorite places" or "bookmarks" feature available on most Web browsers.

3.10. What is a spike solution in XP?

3.11. Describe the XP concepts of refactoring and pair programming in your own words.

3.12. Do a bit more reading and describe what a time-box is. How does this assist an ASD team in delivering software increments in a short time period?

3.13. Do the 80 percent rule in DSDM and the time-boxing approach defined for ASD achieve the same result?

3.14. Using the process pattern template presented in Chapter 2, develop a process pattern for any one of the Scrum patterns presented in Section 3.5.2.

3.15. Why is Crystal called a family of agile methods?

3.16. Using the FDD feature template described in Section 3.5.5, define a feature set for a Web browser. Now develop a set of features for the feature set.

3.17. Visit the Official Agile Modeling Site and make a complete list of all core and supplementary AM principles.

3.18. The tool set proposed in Section 3.6 supports many of the "soft" aspects of agile methods. Since communication is so important, recommend an actual tool set that might be used to enhance communication among stakeholders on an agile team.

Further Readings and Information Sources

The overall philosophy and underlying principles of agile software development are considered in depth in many of the books referenced in the body of this chapter. In addition, books by Shaw and Warden (*The Art of Agile Development,* O'Reilly Media, Inc., 2008), Hunt (*Agile Software Construction,* Springer, 2005), and Carmichael and Haywood (*Better Software Faster,* Prentice-Hall, 2002) present useful discussions of the subject. Aguanno (*Managing Agile Projects,* Multi-Media Publications, 2005), Highsmith (*Agile Project Management: Creating Innovative Products,* Addison-Wesley, 2004), and Larman (*Agile and Iterative Development: A Manager's Guide,* Addison-Wesley, 2003) present a management overview and consider project management issues. Highsmith (*Agile Software Development Ecosystems,* Addison-Wesley, 2002) presents a survey of agile principles, processes, and practices. A worthwhile discussion of the delicate balance between agility and discipline is presented by Booch and his colleagues (*Balancing Agility and Discipline,* Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship,* Prentice-Hall, 2009) presents the principles, patterns, and practices required to develop "clean code" in an agile software engineering environment. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises,* Addison-Wesley, 2007) discusses strategies for scaling up agile practices for large projects. Lippert and Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully,* Wiley, 2006) discuss the use of refactoring when applied in large, complex systems.

Stamelos and Sfetsos (*Agile Software Development Quality Assurance,* IGI Global, 2007) discuss SQA techniques that conform to the agile philosophy.

Dozens of books have been written about Extreme Programming over the past decade. Beck (*Extreme Programming Explained: Embrace Change,* 2d ed., Addison-Wesley, 2004) remains the definitive treatment of the subject. In addition, Jeffries and his colleagues (*Extreme Programming Installed,* Addison-Wesley, 2000), Succi and Marchesi (*Extreme Programming Examined,* Addison-Wesley, 2001), Newkirk and Martin (*Extreme Programming in Practice,* Addison-Wesley, 2001), and Auer and his colleagues (*Extreme Programming Applied: Play to Win,* Addison-Wesley, 2001) provide a nuts-and-bolts discussion of XP along with guidance on how best to apply it. McBreen (*Questioning Extreme Programming,* Addison-Wesley, 2003) takes a critical look at XP, defining when and where it is appropriate. An in-depth consideration of pair programming is presented by McBreen (*Pair Programming Illuminated,* Addison-Wesley, 2003).

ASD is addressed in depth by Highsmith [Hig00]. Schwaber (*The Enterprise and Scrum,* Microsoft Press, 2007) discusses the use of Scrum for projects that have a major business impact. The nuts and bolts of Scrum are discussed by Schwaber and Beedle (*Agile Software Development with SCRUM,* Prentice-Hall, 2001). Worthwhile treatments of DSDM have been written by the DSDM Consortium (*DSDM: Business Focused Development,* 2d ed., Pearson Education, 2003) and Stapleton (*DSDM: The Method in Practice,* Addison-Wesley, 1997). Cockburn (*Crystal Clear,* Addison-Wesley, 2005) presents an excellent overview of the Crystal family of processes. Palmer and Felsing [Pal02] present a detailed treatment of FDD. Carmichael and Haywood (*Better Software Faster,* Prentice-Hall, 2002) provides another useful treatment of FDD that includes a step-by-step journey through the mechanics of the process. Poppendieck and Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers,* Addison-Wesley, 2003) provide guidelines for managing and controlling agile projects. Ambler and Jeffries (*Agile Modeling,* Wiley, 2002) discuss AM in some depth.

A wide variety of information sources on agile software development are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the agile process can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

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 a user interface is designed?
- What is pattern-based design?
- What specialized strategies and methods are used to design WebApps?

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

CHAPTER

4 Principles That Guide Practice

Key Concepts

In a book that explores the lives and thoughts of software engineers, Ellen Ullman [Ull97] depicts a slice of life as she relates the thoughts of practitioner under pressure:

> I have no idea what time it is. There are no windows in this office and no clock, only the blinking red LED display of a microwave, which flashes 12:00, 12:00, 12:00, 12:00. Joel and I have been programming for days. We have a bug, a stubborn demon of a bug. So the red pulse no-time feels right, like a read-out of our brains, which have somehow synchronized themselves at the same blink rate . . .
>
> What are we working on? . . . The details escape me just now. We may be helping poor sick people or tuning a set of low-level routines to verify bits on a distributed database protocol—I don't care. I should care; in another part of my being—later, perhaps when we emerge from this room full of computers—I will care very much why and for whom and for what purpose I am writing software. But just now: no. I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer. . . .

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? The 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. Practice provides you with the detail 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? Three elements of practice apply regardless of the process model that is chosen. They are: principles, concepts, and methods. A fourth element of practice—tools—supports 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.

A dark image of software engineering practice to be sure, but upon reflection, many of the readers of this book will be able to relate to it.

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, my focus is on principles and concepts that guide software engineering practice in general.

4.1 Software Engineering Knowledge

In an editorial published in *IEEE Software* a decade ago, Steve McConnell [McC99] made the following comment:

> Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, html, C++, Linux, Windows NT, and so on. Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C++, you have to know something about C++ to get your program to work.
>
> You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

McConnell goes on to argue that the body of software engineering knowledge (circa the year 2000) had evolved to a "stable core" that he estimated represented about "75 percent of the knowledge needed to develop a complex system." But what resides within this stable core?

As McConnell indicates, core principles—the elemental ideas that guide software engineers in the work that they do—now provide a foundation from which software engineering models, methods, and tools can be applied and evaluated.

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

4.2 Core Principles

Quote:

"In theory there is no difference between theory and practice. But, in practice, there is."

Jan van de Snepscheut

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 framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the level of practice, 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.

In Chapter 1, I identified a set of general principles that span software engineering process and practice: (1) provide value to end users, (2) keep it simple, (3) maintain the vision (of the product and the project), (4) recognize that others consume (and must understand) what you produce, (5) be open to the future, (6) plan ahead for reuse, and (7) think! Although these general principles are important, they are characterized at such a high level of abstraction that they are sometimes difficult to translate into day-to-day software engineering practice. In the subsections that follow, I take a more detailed look at the core principles that guide process and practice.

4.2.1 Principles That Guide Process

In Part 1 of this book I discussed the importance of the software process and described the many different process models that have been proposed for software engineering work. Regardless of whether a model is linear or iterative, prescriptive or agile, it can be characterized using the generic process framework that is applicable for all process models. The following set of core principles can be applied to the framework, and by extension, to every software process.

ADVICE

Every project and every team is unique. That means that you must adapt your process to best fit your needs.

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.

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.

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.

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. A list of required functions and features will be passed along to the person (people) who will develop a design, the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

Quote:

"The truth of the matter is that you always know the right thing to do. The hard part is doing it."

General H. Norman Schwarzkopf

Part 4 of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

4.2.2 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 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 are 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* (SoC). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.

KEY POINT

Problems are easier to solve when they are subdivided into separate concerns, each distinct, individually solvable, and verifiable.

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 I 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).

Joel Spolsky [Spo02] suggests that "all non-trivial abstractions, to some degree, are leaky." The intent of an abstraction is to eliminate the need to communicate details. But sometimes, problematic effects precipitated by these details "leak" through. Without an understanding of the details, the cause of a problem cannot be easily diagnosed.

Principle 3. Strive for consistency. Whether it's creating a requirements 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 WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.

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 as a consequence, there are opportunities for error, or omission, 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—it should be cohesive in its function and/or constrained in the content it represents. Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

Use patterns (Chapter 12) to capture knowledge and experience for future generations of software engineers.

Principle 6. *Look for patterns.* Brad Appleton [App00] suggests that:

> The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the

body of knowledge which defines our understanding of good architectures that meet the needs of their users.

Principle 7. ***When possible, represent the problem and its solution from a number of different perspectives.*** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered. For example, a requirements model can be represented using a data-oriented viewpoint, a function-oriented viewpoint, or a behavioral viewpoint (Chapters 6 and 7). Each provides a different view of the problem and its requirements.

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.

4.3 Principles That Guide Each Framework Activity

Quote:

"The ideal engineer is a composite. . . . He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems."

N. W. Dougherty

In the sections that follow I 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 4.2. They are simply core principles stated at a lower level of abstraction.

4.3.1 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. In this context, I 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.

Principle 1. ***Listen.*** Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but 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.

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.

"Plain questions and plain answers make the shortest road to most perplexities."

Mark Twain

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 than other principles are followed.

Principle 4. ***Face-to-face communication is best.*** But it 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 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 at the moment, 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.

What happens if I can't come to an agreement with the customer on some project-related issue?

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.

INFO

The Difference Between Customers and End Users

Software engineers communicate with many different stakeholders, but customers and end users have the most significant impact on the technical work that follows. In some cases the customer and the end user are one and the same, but for many projects, the customer and the end user are different people, working for different managers, in different business organizations.

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 department. 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.

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 kick-off 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 kick-off 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."

4.3.2 Planning Principles

The communication activity helps you to define your overall goals and objectives (subject, of course, to change as time passes). However, understanding these goals and objectives is not the same as defining a plan for getting there. 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.

Quote:

"In preparing for battle I have always found that plans are useless, but planning is indispensable."

General Dwight D. Eisenhower

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.

There are many different planning philosophies.[2] 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. Still others are "agilists," arguing that a quick "planning game" may be necessary, but that the road map will emerge as "real work" on the software begins.

WebRef

An excellent repository of planning and project management information can be found at **www.4pm.com/repository.htm**.

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 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.

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. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

Quote:

"Success is more a function of consistent common sense than it is of genius."

An Wang

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. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

2 A detailed discussion of software project planning and management is presented in Part 4 of this book.

Principle 6. ***Be realistic.*** People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

The term *granularity* refers to the detail with which some element of planning is represented or conducted.

Principle 7. ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed. 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. Over the next few weeks or months, the project can be planned in significant detail. 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[3] 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.*** Even the best planning can be obviated by uncontrolled change. 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 on a daily basis, 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.

4.3.3 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 model that is identical in form and shape but smaller in scale. However, when the

3 Technical reviews are discussed in Chapter 15.

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.

Requirements models represent customer requirements. Design models provide a concrete specification for the construction of the software.

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. *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

In their book on agile modeling, Scott Ambler and Ron Jeffries [Amb02b] define a set of modeling principles[4] that are intended for those who use the agile process model (Chapter 3) but are appropriate for all software engineers who perform modeling actions 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.

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.

The intent of any model is to communicate information. To accomplish this, use a consistent format. Assume that you won't be there to explain the model. It should stand on its own.

Principle 3. ***Strive to produce the simplest model that will describe the problem or the software.*** Don't overbuild the software [Amb02b]. 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

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

get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, 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 that controls an automobile engine.

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 the model absolutely complete and internally consistent is not worth the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. 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.*** 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.

Requirements modeling principles. Over the past three decades, a large number of requirements modeling methods have been developed. Investigators have

identified requirements analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

Principle 1. ***The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

Principle 2. ***The functions that the software performs must be defined.*** Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

KEY POINT

Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered, and behavior to be exhibited.

Principle 3. ***The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns,* and it is a key strategy in requirements modeling.

Quote:

"The engineer's first problem in any design situation is to discover what the problem really is."

Author unknown

Principle 5. ***The analysis task should move from essential information toward implementation detail.*** Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively,

a keyboard command might be typed, a joystick (or mouse) might be pointed in a specific direction, or a motion-sensitive device might be waved in the air.

By applying these principles, a software engineer approaches a problem systematically. But how are these principles applied in practice? This question will be answered in Chapters 5 through 7.

Design Modeling Principles. The software design model is analogous to an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

There is no shortage of methods for deriving the various elements of a software design. Some methods are data driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern driven, using information about the problem domain (the requirements model) to develop architectural styles and processing patterns. Still others are object oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied regardless of the method that is used:

Quote:

"See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect."

William Shakespeare

Principle 1. ***Design should be traceable to the requirements model.*** The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

WebRef

Insightful comments on the design process, along with a discussion of design aesthetics, can be found at **cs.wwc.edu/~aabyan/Design/**.

Principle 2. ***Always consider the architecture of the system to be built.*** Software architecture (Chapter 9) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. ***Design of data is as important as design of processing functions.*** Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Quote:

"The differences are not minor—they are rather like the differences between Salieri and Mozart. Study after study shows that the very best designers produce structures that are faster, smaller, simpler, clearer, and produced with less effort."

Frederick P. Brooks

Principle 4. ***Interfaces (both internal and external) must be designed with care.*** The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. ***User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use.*** The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. ***Component-level design should be functionally independent.*** Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.[5]

Principle 7. ***Components should be loosely coupled to one another and to the external environment.*** Coupling is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle 8. ***Design representations (models) should be easily understandable.*** The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. ***The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.*** Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

When these design principles are properly applied, you create a design that exhibits both external and internal quality factors [Mye78]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts (Chapter 8).

5 Additional discussion of cohesion can be found in Chapter 8.

4.3.4 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 (e.g., Java), (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., Visual C++).

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), *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. The following set of fundamental principles and concepts are applicable to coding and testing:

Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

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

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: ***As you begin writing code, be sure you***

- Constrain your algorithms by following structured programming [Boh00] practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.

Quote:

"For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change."

David Parnas

ADVICE

Avoid developing an elegant program that solves the wrong problem. Pay particular attention to the first preparation principle.

- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

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

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

WebRef

A wide variety of links to coding standards can be found at **www.literateprogramming.com/fpstyle.html**.

More books have been written about programming (coding) and the principles and concepts that guide it than about any other topic in the software process. Books on the subject include early works on programming style [Ker78], practical software construction [McC04], programming pearls [Ben99], the art of programming [Knu98], pragmatic programming issues [Hun99], and many, many other subjects. A comprehensive discussion of these principles and concepts is beyond the scope of this book. If you have further interest, examine one or more of the references noted.

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

? What are the objectives of software testing?

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

ADVICE

In a broader software design context, recall that you begin "in the large" by focusing on software architecture and end "in the small" focusing on components. For testing, you simply reverse the focus and test your way out.

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.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. 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 as a whole. But testing cannot show the absence of errors and defects; it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

Davis [Dav95b] suggests a set of testing principles[6] that have been adapted for use in this book:

Principle 1. ***All tests should be traceable to customer requirements.***[7] 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 17) 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 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 in an attempt to find 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.

4.3.5 Deployment Principles

As I noted earlier in Part 1 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 a number of 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

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

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

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.

Be sure that your customer knows what to expect before a software increment is delivered. Otherwise, you can bet the customer will expect more than you deliver.

The delivery of a software increment represents an important milestone for any software project. A number of 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 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 frame 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.*** A CD-ROM or other media (including Web-based downloads) containing 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 as many different computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements) as possible.

Principle 3. ***A support regime 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.[8]

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

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."

The delivered software provides benefit for the end user, but it also provides useful feedback for the software team. As the increment is put into use, end users should be encouraged to comment on features and functions, ease of use, reliability, and any other characteristics that are appropriate.

4.4 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 apply to the process as a whole and to the practice of each framework activity regardless of the project or the product.

A set of core principles 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 navigates through the software process. At the level of practice, 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 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. Modeling principles serve as a foundation for the methods and notation that are used to create representations of 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 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

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

4.2. Of the eight core principles that guide process (discussed in Section 4.2.1), which do you believe is most important?

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

4.4. An important communication principle states "prepare before you communicate." How should this preparation manifest itself in the early work that you do? What work products might result as a consequence of early preparation?

4.5. Do some research on "facilitation" for the communication activity (use the references provided or others) and prepare a set of guidelines that focus solely on facilitation.

4.6. How does agile communication differ from traditional software engineering communication? How is it similar?

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

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

4.9. Describe what *granularity* means in the context of a project schedule.

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

4.11. What three "domains" are considered during requirements modeling?

4.12. Try to add one additional principle to those stated for coding in Section 4.3.4.

4.13. What is a successful test?

4.14. Do you agree or disagree with the following statement: "Since we deliver multiple increments to the customer, why should we be concerned about quality in the early increments—we can fix problems in later iterations." Explain your answer.

4.15. Why is feedback important to the software team?

Further Readings and Information Sources

Customer communication is a critically important activity in software engineering, yet few practitioners spend any time reading about it. Withall (*Software Requirements Patterns,* Microsoft Press, 2007) presents a variety of useful patterns that address communications problems. Sutliff

(*User-Centred Requirements Engineering,* Springer, 2002) focuses heavily on communications-related challenges. Books by Weigers (*Software Requirements,* 2d ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer,* Dorset House, 1996), and Karten [Kar94] provide much insight into methods for effective customer interaction. Although their book does not focus on software, Hooks and Farry (*Customer Centered Products*, American Management Association, 2000) present useful generic guidelines for customer communication. Young (*Effective Requirements Practices,* Addison-Wesley, 2001) emphasizes a "joint team" of customers and developers who develop requirements collaboratively. Somerville and Kotonya (*Requirements Engineering: Processes and Techniques,* Wiley, 1998) discuss "elicitation" concepts and techniques and other requirements engineering principles.

Communication and planning concepts and principles are considered in many project management books. Useful project management offerings include books by Bechtold (*Essentials of Software Project Management,* 2d ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme,* 4th ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success,* BookSurge Publishing, 2006), Hughes (*Software Project Management,* McGraw-Hill, 2005), and Stellman and Greene (*Applied Software Project Management,* O'Reilly Media, Inc., 2005).

Davis [Dav95] has compiled an excellent collection of software engineering principles. In addition, virtually every book on software engineering contains a useful discussion of concepts and principles for analysis, design, and testing. Among the most widely used offerings (in addition to this book!) are:

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

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

Jalote, P., *An Integrated Approach to Software Engineering,* Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice,* 3d ed., Prentice-Hall, 2005.

Schach, S., *Object-Oriented and Classical Software Engineering,* McGraw-Hill, 7th ed., 2006.

Sommerville, I., *Software Engineering,* 8th ed., Addison-Wesley, 2006.

These books also present detailed discussion of modeling and construction principles.

Modeling principles are considered in many books dedicated to requirements analysis and/or software design. Books by Lieberman (*The Art of Software Modeling,* Auerbach, 2007), Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Roques (*UML in Practice,* Wiley, 2004), Penker and Eriksson (*Business Modeling with UML: Business Patterns at Work,* Wiley, 2001) discuss modeling principles and methods.

Norman's (*The Design of Everyday Things,* Currency/Doubleday, 1990) is must reading for every software engineer who intends to do design work. Winograd and his colleagues (*Bringing Design to Software,* Addison-Wesley, 1996) have edited an excellent collection of essays that address practical issues for software design. Constantine and Lockwood (*Software for Use,* Addison-Wesley, 1999) present the concepts associated with "user centered design." Tognazzini (*Tog on Software Design,* Addison-Wesley, 1995) presents a worthwhile philosophical discussion of the nature of design and the impact of decisions on quality and a team's ability to produce software that provides great value to its customer. Stahl and his colleagues (*Model-Driven Software Development: Technology, Engineering,* Wiley, 2006) discuss the principles of model-driven development.

Hundreds of books address one or more elements of the construction activity. Kernighan and Plauger [Ker78] have written a classic text on programming style, McConnell [McC93] presents pragmatic guidelines for practical software construction, Bentley [Ben99] suggests a wide variety of programming pearls, Knuth [Knu99] has written a classic three-volume series on the art of programming, and Hunt [Hun99] suggests pragmatic programming guidelines.

Myers and his colleagues (*The Art of Software Testing,* 2d ed., Wiley, 2004) have developed a major revision of his classic text and discuss many important testing principles. Books by Perry

(*Effective Methods for Software Testing,* 3d ed., Wiley, 2006), Whittaker (*How to Break Software,* Addison-Wesley, 2002), Kaner and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001), and Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) each present important testing concepts and principles and much pragmatic guidance.

A wide variety of information sources on software engineering practice are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software engineering practice can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

5

UNDERSTANDING REQUIREMENTS

KEY CONCEPTS

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard. After all, doesn't the customer know what is required? Shouldn't the end users have a good understanding of the features and functions that will provide benefit? Surprisingly, in many instances the answer to these questions is "no." And even if customers and end-users are explicit in their needs, those needs will change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, I wrote:

> It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late

QUICK LOOK

What is it? Before you begin any technical work, it's a good idea to apply a set of requirements engineering tasks. These tasks lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software.

Who does it? Software engineers (sometimes referred to as system engineers or "analysts" in the IT world) and other project stakeholders (managers, customers, end users) all participate in requirements engineering.

Why is it important? Designing and building an elegant computer program that solves the wrong problem serves no one's needs. That's why it's important to understand what the customer wants before you begin to design and build a computer-based system.

What are the steps? Requirements engineering begins with inception—a task that defines the scope and nature of the problem to be solved. It moves onwards to elicitation—a task that helps stakeholders define what is required, and then elaboration—where basic requirements are refined and modified. As stakeholders define the problem, negotiation occurs—what are the priorities, what is essential, when is it required? Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

What is the work product? The intent of requirements engineering is to provide all parties with a written understanding of the problem. This can be achieved though a number of work products: usage scenarios, functions and features lists, requirements models, or a specification.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that what you have learned is what they really meant. A word of warning: even after all parties agree, things will change, and they will continue to change throughout the project.

in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a disorganized manner, and we spend far too little time verifying what we do record. We allow change to control us, rather than establishing mechanisms to control change. In short, we fail to establish a solid foundation for the system or software. Each of these problems is challenging. When they are combined, the outlook is daunting for even the most experienced managers and practitioners. But solutions do exist.

It's reasonable to argue that the techniques I'll discuss in this chapter are not a true "solution" to the challenges just noted. But they do provide a solid approach for addressing these challenges.

5.1 Requirements Engineering

Quote:

"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

Fred Brooks

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program and all else is secondary. What makes these arguments seductive is that they contain elements of truth.[1] But each is flawed and can lead to a failed software project.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *requirements engineering*. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

KEY POINT

Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, end users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it begins with a broader system definition, where software is but one component of the larger system domain. But regardless of the starting point, the journey across the bridge takes you

1 This is particularly true for small projects (less than one month) and smaller, relatively simple software efforts. As software grows in size and complexity, these arguments begin to break down.

ADVICE

Expect to do a bit of design during requirements work and a bit of requirements work during design.

high above the project, allowing you to examine the context of the software work to be performed; the specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [Tha97]. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Quote:

"The seeds of major software disasters are usually sown in the first three months of commencing the software project."

Caper Jones

Inception. How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort. But in general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.[2]

At project inception,[3] you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

? Why is it difficult to gain a clear understanding of what the customer wants?

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs.

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

2 If a computer-based system is to be developed, discussions begin within the context of a system engineering process. For a detailed discussion of system engineering, visit the website that accompanies this book.

3 Recall that the Unified Process (Chapter 2) defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- **Problems of volatility.** The requirements change over time.

To help overcome these problems, you must approach requirements gathering in an organized manner.

Elaboration is a good thing, but you have to know when to stop. The key is to describe the problem in a way that establishes a firm base for design. If you work beyond that point, you're doing design.

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 6 and 7) that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services[4] that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

There should be no winner and no loser in an effective negotiation. Both sides win, because a "deal" that both can live with is solidified.

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification. In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a

4 A *service* manipulates the data encapsulated by the class. The terms *operation* and *method* are also used. If you are unfamiliar with object-oriented concepts, a basic introduction is presented in Appendix 2.

The formality and format of a specification varies with the size and the complexity of the software to be built.

consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents
Revision History

1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Project Scope
1.5 References

2. Overall Description
2.1 Product Perspective
2.2 Product Features
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies

3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)

4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces

5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification[5] to ensure that all software requirements have been

5 Recall that the nature of the specification will vary with each project. In some cases, the "specification" is a collection of user scenarios and little else. In others, the specification may be a document that contains scenarios, models, and written descriptions.

stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated.

The primary requirements validation mechanism is the technical review (Chapter 15). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

INFO

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.[6] Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 22.

6 Formal requirements management is initiated only for large projects that have hundreds of identifiable requirements. For small projects, this requirements engineering action is considerably less formal.

SOFTWARE TOOLS

Requirements Engineering

Objective: Requirements engineering tools assist in requirements gathering, requirements modeling, requirements management, and requirements validation.

Mechanics: Tool mechanics vary. In general, requirements engineering tools build a variety of graphical (e.g., UML) models that depict the informational, functional, and behavioral aspects of a system. These models form the basis for all other activities in the software process.

Representative Tools:[7]
A reasonably comprehensive (and up-to-date) listing of requirements engineering tools can be found at the Volvere Requirements resources site at **www.volere.co.uk/tools.htm**. Requirements modeling tools are discussed in Chapters 6 and 7. Tools noted below focus on requirement management.

EasyRM, developed by Cybernetic Intelligence GmbH (**www.easy-rm.com**), builds a project-specific dictionary/glossary that contains detailed requirements descriptions and attributes.

Rational RequisitePro, developed by Rational Software (**www-306.ibm.com/software/awdtools/reqpro/**), allows users to build a requirements database; represent relationships among requirements; and organize, prioritize, and trace requirements.

Many additional requirements management tools can be found at the Volvere site noted earlier and at **www.jiludwig.com/Requirements_Management_Tools.html**.

5.2 ESTABLISHING THE GROUNDWORK

In an ideal setting, stakeholders and software engineers work together on the same team.[8] In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation.

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed.

In the sections that follow, I discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

5.2.1 Identifying Stakeholders

Sommerville and Sawyer [Som97] define a stakeholder as "anyone who benefits in a direct or indirect way from the system which is being developed." I have already

7 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.

8 This approach is strongly recommended for projects that adopt an agile software development philosophy.

identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 5.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

5.2.2 Recognizing Multiple Viewpoints

Quote:

"Put three stakeholders in a room and ask them what kind of system they want. You're likely to get four or more different opinions."

Author unknown

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

5.2.3 Working toward Collaboration

If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, I have noted that customers (and other stakeholders) must collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

INFO

Using "Priority Points"

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points*. All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of each (from his or her viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

5.2.4 Asking the First Questions

Questions asked at the inception of the project should be "context free" [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

Quote:

"It is better to know some of the questions than all of the answers."

James Thurber

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

? What questions will help you gain a preliminary understanding of the problem?

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Quote:

"He who asks a question is a fool for five minutes; he who does not ask a question is a fool forever."

Chinese proverb

These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 5.3.

5.3 Eliciting Requirements

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements [Zah90].[9]

5.3.1 Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

? What are the basic guidelines for conducting a collaborative requirements gathering meeting?

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

9 This approach is sometimes called a *facilitated application specification technique* (FAST).

Quote:

"We spend a lot of time—the majority of project effort—not implementing or testing, but trying to decide what to build."

Brian Lawrence

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur, I present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting.

During inception (Section 5.2) basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request."

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

WebRef

Joint Application Development (JAD) is a popular technique for requirements gathering. A good description can be found at **www.carolla.com/wp-jad.htm**.

As an example,[10] consider an excerpt from a product request written by a marketing person involved in the *SafeHome* project. This person writes the following narrative about the home security function that is to be part of *SafeHome:*

> Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with "alarm systems" so this would be an easy sell.
>
> The home security function would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, carbon monoxide levels, and others. It'll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

ADVICE

If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross section of users. If only one user defines all requirements, acceptance risk is high.

In reality, others would contribute to this narrative during the requirements gathering meeting and considerably more information would be available. But even with additional information, ambiguity would be present, omissions would likely exist, and errors might occur. For now, the preceding "functional description" will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

10 This example (with extensions and variations) is used to illustrate important software engineering methods in many of the chapters that follow. As an exercise, it would be worthwhile to conduct your own requirements gathering meeting and develop a set of lists for it.

Quote:

"Facts do not cease to exist because they are ignored."

Aldous Huxley

Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

ADVICE

Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board, at an internal website, or posed in a chat room environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be modified, and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for entries on the lists.[11] Each mini-specification is an elaboration of an object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be:

> The control panel is a wall-mounted unit that is approximately 9 × 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 × 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.

11 Rather than creating a mini-specification, many software teams elect to develop user scenarios called *use cases.* These are considered in detail in Section 5.4 and in Chapter 6.

SAFEHOME

Conducting a Requirements Gathering Meeting

The scene: A meeting room. The first requirements gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

5.3.2 Quality Function Deployment

QFD defines requirements in a way that maximizes customer satisfaction.

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process" [Zul92]. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements [Zul92]:

Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Everyone wants to implement lots of exciting requirements, but be careful. That's how "requirements creep" sets in. On the other hand, exciting requirements lead to a breakthrough product!

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

WebRef

Useful information on QFD can be obtained at **www.qfdi.org**.

Although QFD concepts can be applied across the entire software process [Par96a], specific QFD techniques are applicable to the requirements elicitation activity. QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [Aka04].

5.3.3 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* [Jac92], provide a description of how the system will be used. Use cases are discussed in greater detail in Section 5.4.

SAFEHOME

Developing a Preliminary User Scenario

The scene: A meeting room, continuing the first requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user id and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

5.3.4 Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

? What information is produced as a consequence of requirements gathering?

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

5.4 Developing Use Cases

In a book that discusses how to write effective use cases, Alistair Cockburn [Coc01b] notes that "a use case captures a contract … [that] describes the system's behavior under various conditions as the system responds to a request from one of its stakeholders . . ." In essence, a use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user's point of view.

KEY POINT

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

WebRef

An excellent paper on use cases can be downloaded from **www.ibm.com/developerworks/webservices/library/codesign7.html**.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

? What do I need to know in order to develop an effective use case?

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions[12] that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

12 Jacobson's questions have been extended to provide a more complete view of use-case content.

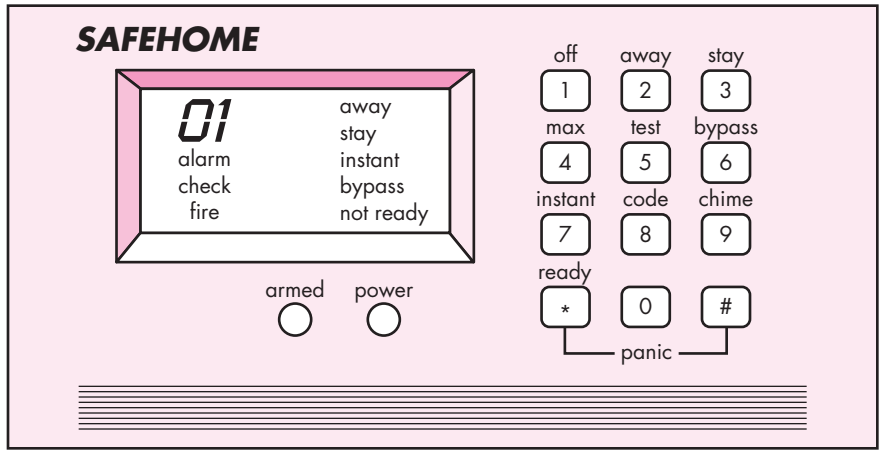
Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:[13]

1. The homeowner observes the *SafeHome* control panel (Figure 5.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

FIGURE 5.1
SafeHome **control panel**



13 Note that this use case differs from the situation in which the system is accessed via the Internet. In this case, interaction occurs via the control panel, not the graphical user interface (GUI) provided when a PC is used.

2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 5.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.

Use cases are often written informally. However, use the template shown here to ensure that you've addressed all key issues.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case: *InitiateMonitoring*

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects "stay" or "away"
4. Homeowner: observes read alarm light to indicate that *SafeHome* has been armed

Exceptions:

1. Control panel is *not ready:* homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. *Stay* is selected: control panel beeps twice and a *stay* light is lit; perimeter sensors are activated.
5. *Away* is selected: control panel beeps three times and an *away* light is lit; all sensors are activated.

Priority: Essential, must be implemented

When available: First increment

Frequency of use: Many times per day

Channel to actor: Via control panel interface

Secondary actors: Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

Use cases for other **homeowner** interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

SafeHome

Developing a High-Level Use-Case Diagram

The scene: A meeting room, continuing the requirements gathering meeting

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 5.2.)

Jamie: I'm just beginning to learn UML notation.[14] So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

Facilitator: Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

Doug: Is that legal in UML?

Facilitator: Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

Vinod: Okay, so we have use-case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

Facilitator: Probably, but that can wait until we've considered other *SafeHome* functions.

Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

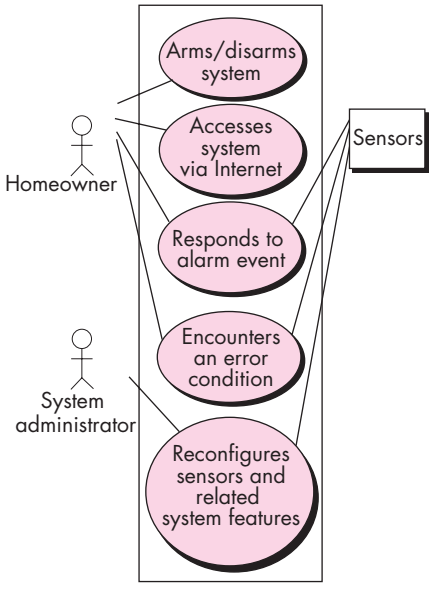
Facilitator: Oh really. Tell me what we've missed.

(The meeting continues.)

14 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

FIGURE 5.2

UML use case diagram for *SafeHome* home security function



SOFTWARE TOOLS

Use-Case Development

Objective: Assist in the development of use cases by providing automated templates and mechanisms for assessing clarity and consistency.

Mechanics: Tool mechanics vary. In general, use-case tools provide fill-in-the-blank templates for creating effective use cases. Most use-case functionality is embedded into a set of broader requirements engineering functions.

Representative Tools:[15]

The vast majority of UML-based analysis modeling tools provide both text and graphical support for use-case development and modeling.

Objects by Design (**www.objectsbydesign.com/tools/umltools_byCompany.html**) provides comprehensive links to tools of this type.

5.5 BUILDING THE REQUIREMENTS MODEL[16]

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

15 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.

16 Throughout this book, I use the terms *analysis model* and *requirements model* synonymously. Both refer to representations of the information, functional, and behavioral domains that describe problem requirements.

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system. The analysis model and the methods that are used to build it are presented in detail in Chapters 6 and 7. I present a brief overview in the sections that follow.

5.5.1 Elements of the Requirements Model

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the requirements model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

The specific elements of the requirements model are dictated by the analysis modeling method (Chapters 6 and 7) that is to be used. However, a set of generic elements is common to most requirements models.

It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 5.4) and their corresponding use-case diagrams (Figure 5.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 5.3 depicts a UML activity diagram[17] for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

One way to isolate classes is to look for descriptive nouns in a use-case script. At least some of the nouns will be candidate classes. More on this in the Chapter 8.

Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 5.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify, enable*) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 7.

Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

17 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

FIGURE 5.3
UML activity diagrams for eliciting requirements

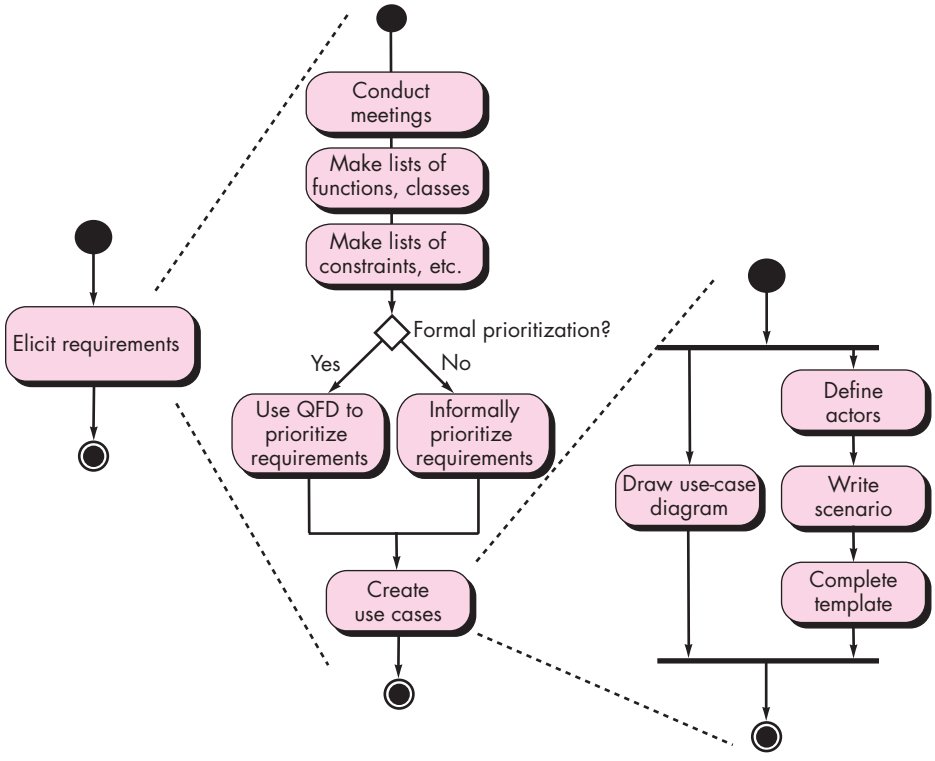
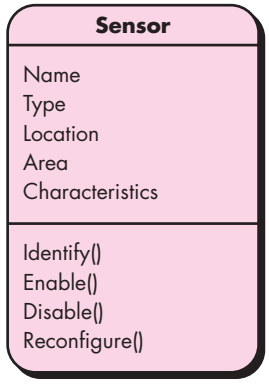


FIGURE 5.4
Class diagram for sensor

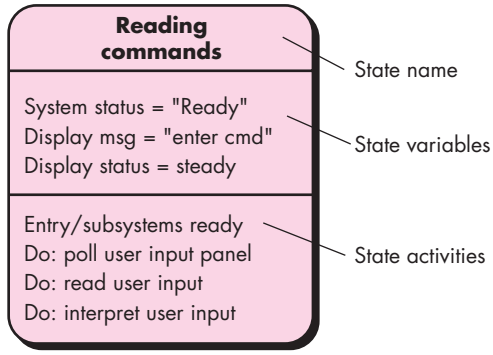


KEY POINT
A state is an externally observable mode of behavior. External stimuli cause transitions between states.

The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.

To illustrate the use of a state diagram, consider software embedded within the *SafeHome* control panel that is responsible for reading user input. A simplified UML state diagram is shown in Figure 5.5.

FIGURE 5.5
UML state diagram notation



In addition to behavioral representations of the system as a whole, the behavior of individual classes can also be modeled. Further discussion of behavioral modeling is presented in Chapter 7.

SAFEHOME

Preliminary Behavioral Modeling

The scene: A meeting room, continuing the requirements meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've just about finished talking about *SafeHome* home security functionality. But before we do, I want to discuss the behavior of the function.

Marketing person: I don't understand what you mean by behavior.

Ed (smiling): That's when you give the product a "timeout" if it misbehaves.

Facilitator: Not exactly. Let me explain.

(The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

Marketing person: This seems a little technical. I'm not sure I can help here.

Facilitator: Sure you can. What behavior do you observe from the user's point of view?

Marketing person: Uh . . . well, the system will be *monitoring* the sensors. It'll be *reading commands* from the homeowner. It'll be *displaying* its status.

Facilitator: See, you can do it.

Jamie: It'll also be *polling* the PC to determine if there is any input from it, for example, Internet-based access or configuration information.

Vinod: Yeah, in fact, *configuring the system* is a state in its own right.

Doug: You guys are rolling. Let's give this a bit more thought . . . is there a way to diagram this stuff?

Facilitator: There is, but let's postpone that until after the meeting.

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a

packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. A more detailed discussion of flow modeling is presented in Chapter 7.

5.5.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.[18] These *analysis patterns* [Fow97] suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler [Gey01] suggest two benefits that can be associated with the use of analysis patterns:

> First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations. Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01][19] that is discussed in more detail in Chapter 12. Examples of analysis patterns and further discussion of this topic are presented in Chapter 7.

5.6 Negotiating Requirements

Quote:

"A compromise is the art of dividing a cake in such a way that everyone believes he has the biggest piece."

Ludwig Erhard

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. In reality, you may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market. The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the

18 In some cases, problems reoccur regardless of the application domain. For example, the features and functions used to solve user interface problems are common regardless of the application domain under consideration.

19 A variety of patterns templates have been proposed in the literature. If you have interest, see [Fow97], [Gam95], [Yac03], and [Bus07] among many sources.

same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a "win-win" result.[20] That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

WebRef

A brief paper on negotiation for software requirements can be downloaded from **www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html**.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

INFO

The Art of Negotiation

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve; what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen to her. It's likely you'll gain knowledge that will help you to better negotiate your position.
4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle; commit to it and move on.

SAFEHOME

The Start of a Negotiation

The scene: Lisa Perez's office, after the first requirements gathering meeting.

The players: Doug Miller, software engineering manager and Lisa Perez, marketing manager.

The conversation:

Lisa: So, I hear the first meeting went really well.

Doug: Actually, it did. You sent some good people to the meeting . . . they really contributed.

20 Dozens of books have been written on negotiating skills (e.g., [Lew06], [Rai06], [Fis06]). It is one of the more important skills that you can learn. Read one.

Lisa (smiling): Yeah, they actually told me they got into it and it wasn't a "propeller head activity."

Doug (laughing): I'll be sure to take off my techie beanie the next time I visit . . . Look, Lisa, I think we may have a problem with getting all of the functionality for the home security system out by the dates your management is talking about. It's early, I know, but I've already been doing a little back-of-the-envelope planning and . . .

Lisa (frowning): We've got to have it by that date, Doug. What functionality are you talking about?

Doug: I figure we can get full home security functionality out by the drop-dead date, but we'll have to delay Internet access 'til the second release.

Lisa: Doug, it's the Internet access that gives *SafeHome* "gee whiz" appeal. We're going to build our entire marketing campaign around it. We've gotta have it!

Doug: I understand your situation, I really do. The problem is that in order to give you Internet access, we'll have to have a fully secure website up and running. That takes time and people. We'll also have to build a lot of additional functionality into the first release . . . I don't think we can do it with the resources we've got.

Lisa (still frowning): I see, but you've got to figure out a way to get it done. It's pivotal to home security functions and to other functions as well . . . those can wait until the next releases . . . I'll agree to that.

Lisa and Doug appear to be at an impasse, and yet they must negotiate a solution to this problem. Can they both "win" here? Playing the role of a mediator, what would you suggest?

5.7 Validating Requirements

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

? When I review requirements, what questions should I ask?

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?

- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

5.8 Summary

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modeling activities that have been defined for the generic software process. Seven distinct requirements engineering functions—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, QFD, and the development of usage scenarios.

Elaboration further expands requirements in a model—a collection of scenario-based, class-based, behavioral, and flow-oriented elements. The model may reference analysis patterns, solutions for analysis problems that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. In addition, each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built.

Problems and Points to Ponder

5.1. Why is it that many software developers don't pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?

5.2. You have been given the responsibility to elicit requirements from a customer who tells you he is too busy to meet with you. What should you do?

5.3. Discuss some of the problems that occur when requirements must be elicited from three or four different customers.

5.4. Why do we say that the requirements model represents a snapshot of a system in time?

5.5. Let's assume that you've convinced the customer (you're a very good salesperson) to agree to every demand that you have as a developer. Does that make you a master negotiator? Why?

5.6. Develop at least three additional "context-free questions" that you might ask a stakeholder during inception.

5.7. Develop a requirements gathering "kit." The kit should include a set of guidelines for conducting a requirements gathering meeting and materials that can be used to facilitate the creation of lists and any other items that might help in defining requirements.

5.8. Your instructor will divide the class into groups of four to six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security function described in this chapter. Conduct a requirements gathering meeting using the guidelines presented in this chapter.

5.9. Develop a complete use case for one of the following activities:

a. Making a withdrawal at an ATM
b. Using your charge card for a meal at a restaurant
c. Buying a stock using an on-line brokerage account
d. Searching for books (on a specific topic) using an on-line bookstore
e. An activity specified by your instructor.

5.10. What do use case "exceptions" represent?

5.11. Describe what an *analysis pattern* is in your own words.

5.12. Using the template presented in Section 5.5.2, suggest one or more analysis pattern for the following application domains:

a. Accounting software
b. E-mail software
c. Internet browsers
d. Word-processing software
e. Website creation software
f. An application domain specified by your instructor

5.13. What does *win-win* mean in the context of negotiation during the requirements engineering activity?

5.14. What do you think happens when requirement validation uncovers an error? Who is involved in correcting the error?

Further Readings and Information Sources

Because it is pivotal to the successful creation of any complex computer-based system, requirements engineering is discussed in a wide array of books. Hood and his colleagues (*Requirements Management,* Springer, 2007) discuss a variety of requirements engineering issues that span both systems and software engineering. Young (*The Requirements Engineering Handbook,* Artech House Publishers, 2007) presents an in-depth discussion of requirements engineering tasks. Wiegers (*More About Software Requirements,* Microsoft Press, 2006) provides many practical techniques for requirements gathering and management. Hull and her colleagues (*Requirements Engineering,* 2d ed., Springer-Verlag, 2004), Bray (*An Introduction to Requirements Engineering,* Addison-Wesley, 2002), Arlow (*Requirements Engineering,* Addison-Wesley, 2001), Gilb (*Requirements Engineering,* Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development,* Addison-Wesley, 1999), and Sommerville and Kotonya (*Requirement Engineering: Processes and Techniques,* Wiley, 1998) are but a few of many books dedicated to the subject. Gottesdiener (*Requirements by Collaboration: Workshops for Defining*

Needs, Addison-Wesley, 2002) provides useful guidance for those who must establish a collaborative requirements gathering environment with stakeholders.

Lauesen (*Software Requirements: Styles and Techniques,* Addison-Wesley, 2002) presents a comprehensive survey of requirement analysis methods and notation. Weigers (*Software Requirements,* Microsoft Press, 1999) and Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach,* 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices and suggest pragmatic guidelines for most aspects of the requirements engineering process.

A patterns-based view of requirements engineering is described by Withall (*Software Requirement Patterns,* Microsoft Press, 2007). Ploesch (*Assertions, Scenarios and Prototypes,* Springer-Verlag, 2003) discusses advanced techniques for developing software requirements. Windle and Abreo (*Software Requirements Using the Unified Process,* Prentice-Hall, 2002) discuss requirements engineering within the context of the Unified Process and UML notation. Alexander and Steven (*Writing Better Requirements,* Addison-Wesley, 2002) present a brief set of guidelines for writing clear requirements, representing them as scenarios, and reviewing the end result.

Use-case modeling is often the driver for the creation of all other aspects of the analysis model. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach,* 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices. Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01], Armour and Miller (*Advanced Use Case Modeling: Software Systems,* Addison-Wesley, 2000), and Kulak and his colleagues (*Use Cases: Requirements in Context,* Addison-Wesley, 2000) discuss requirements gathering with an emphasis on use-case modeling.

A wide variety of information sources on requirements engineering and analysis is available on the Internet. An up-to-date list of World Wide Web references that are relevant to requirements engineering and analysis can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

6 REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES

KEY CONCEPTS

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model[1]—actually a set of models—is the first technical representation of a system.

In a seminal book on requirements modeling methods, Tom DeMarco [DeM79] describes the process in this way:

> Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals. The products of analysis must be highly maintainable. This applies particularly to the

QUICK LOOK

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an "analyst") builds the model using requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. In this chapter you'll consider requirements modeling from three different perspectives: scenario-based models, data (information) models, and class-based models. Each represents requirements in a different "dimension," thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Scenario-based modeling represents the system from the user's point of view. Data modeling represents the information space and depicts the data objects that the software will manipulate and the relationships among them. Class-based modeling defines objects, attributes, and relationships. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency. In Chapter 7, we extend the modeling dimensions noted here with additional representations, providing a more robust view of requirements.

What is the work product? A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

1 In past editions of this book, I used the term *analysis model,* rather than *requirements model.* In this edition, I've decided to use both phrases to represent the modeling activity that defines various aspects of the problem to be solved. *Analysis* is the action that occurs as *requirements* are derived.

> Target Document [software requirements specification]. Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out. Graphics have to be used whenever possible. We have to differentiate between logical [essential] and physical [implementation] considerations. . . . At the very least, we need. . . . Something to help us partition our requirements and document that partitioning before specification. . . . Some means of keeping track of and evaluating interfaces. . . . New tools to describe logic and policy, something better than narrative text.

Although DeMarco wrote about the attributes of analysis modeling more than a quarter century ago, his comments still apply to modern requirements modeling methods and notation.

6.1 Requirements Analysis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer,* an *analyst,* or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 5).

Quote:

"Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."

Alan M. Davis

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors"
- *Data models* that depict the information domain for the problem
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external "events"

KEY POINT

The analysis model and requirements specification provide a means for assessing quality once the software is built.

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this chapter, I focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class*

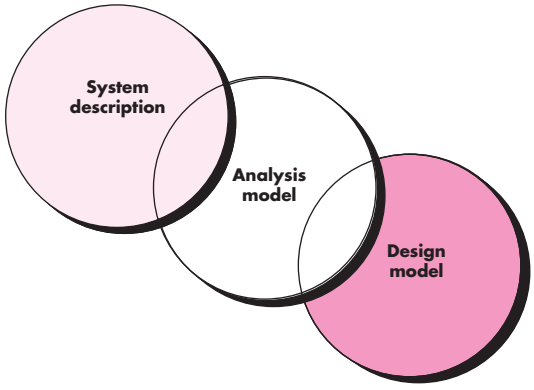


FIGURE 6.1 The requirements model as a bridge between the system description and the design model

modeling—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models, pattern-based modeling, and WebApp models are discussed in Chapter 7.

6.1.1 Overall Objectives and Philosophy

Quote:

"Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need."

Andrew Hunt and David Thomas

Throughout requirements modeling, your primary focus is on *what,* not *how.* What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?[2]

In earlier chapters, I noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.[3]

KEY POINT

The analysis model should describe what the customer wants, establish a basis for design, and establish a target for validation.

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design (Chapters 8 through 13) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 6.1.

2 It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what.* However, the primary focus should remain on *what.*

3 Alternatively, the software team may choose to create a prototype (Chapter 2) in an effort to better understand requirements for the system.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

6.1.2 Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

? Are there basic guidelines that can help us as we do requirements analysis work?

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" [Arl02] that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.
- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.
- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

Quote:

"Problems worthy of attack, prove their worth by hitting back."

Piet Hein

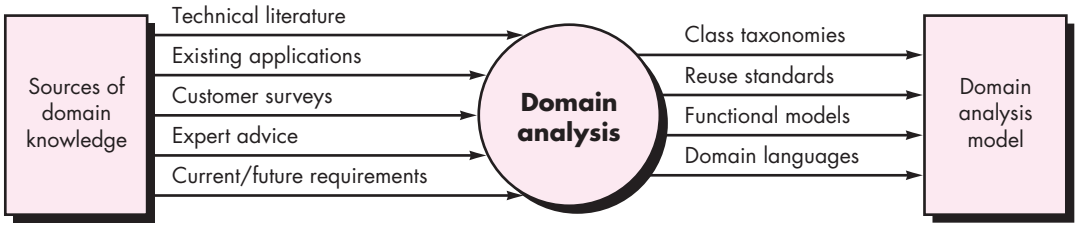
6.1.3 Domain Analysis

WebRef

Many useful resources for domain analysis can be found at **www.iturls.com/English/Software Engineering/SE_mod5.asp**.

In the discussion of requirements engineering (Chapter 5), I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

FIGURE 6.2 Input and output for domain analysis



But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis.* Firesmith [Fir93] describes domain analysis in the following way:

> Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

KEY POINT

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.[4]

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst[5] is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 [Ara89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

4 A complementary view of domain analysis "involves modeling the domain so that software engineers and other stakeholders can better learn about it . . . not all domain classes necessarily result in the development of reusable classes . . ." [Let03a].

5 Do not make the assumption that because a domain analyst is at work, a software engineer need not understand the application domain. Every member of a software team should have some understanding of the domain in which the software is to be placed.

SAFEHOME

Domain Analysis

The scene: Doug Miller's office, after a meeting with marketing.

The players: Doug Miller, software engineering manager, and Vinod Raman, a member of the software engineering team.

The conversation:

Doug: I need you for a special project, Vinod. I'm going to pull you out of the requirements gathering meetings.

Vinod (frowning): Too bad. That format actually works . . . I was getting something out of it. What's up?

Doug: Jamie and Ed will cover for you. Anyway, marketing insists that we deliver the Internet capability along with the home security function in the first release of *SafeHome*. We're under the gun on this . . . not enough time or people, so we've got to solve both problems—the PC interface and the Web interface—at once.

Vinod (looking confused): I didn't know the plan was set . . . we're not even finished with requirements gathering.

Doug (a wan smile): I know, but the time lines are so short that I decided to begin strategizing with marketing right now . . . anyhow, we'll revisit any tentative plan once we have the info from all of the requirements gathering meetings.

Vinod: Okay, what's up? What do you want me to do?

Doug: Do you know what "domain analysis" is?

Vinod: Sort of. You look for similar patterns in Apps that do the same kinds of things as the App you're building. If possible, you then steal the patterns and reuse them in your work.

Doug: Not sure I like the word *steal,* but basically you have it right. What I'd like you to do is to begin researching existing user interfaces for systems that control something like *SafeHome*. I want you to propose a set of patterns and analysis classes that can be common to both the PC-based interface that'll sit in the house and the browser-based interface that is accessible via the Internet.

Vinod: We can save time by making them the same . . . why don't we just do that?

Doug: Ah . . . it's nice to have people who think like you do. That's the whole point—we can save time and effort if both interfaces are nearly identical, implemented with the same code, blah, blah, that marketing insists on.

Vinod: So you want, what—classes, analysis patterns, design patterns?

Doug: All of 'em. Nothing formal at this point. I just want to get a head start on our internal analysis and design work.

Vinod: I'll go to our class library and see what we've got. I'll also use a patterns template I saw in a book I was reading a few months back.

Doug: Good. Go to work.

Quote:

"… analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you."

Tom DeMarco

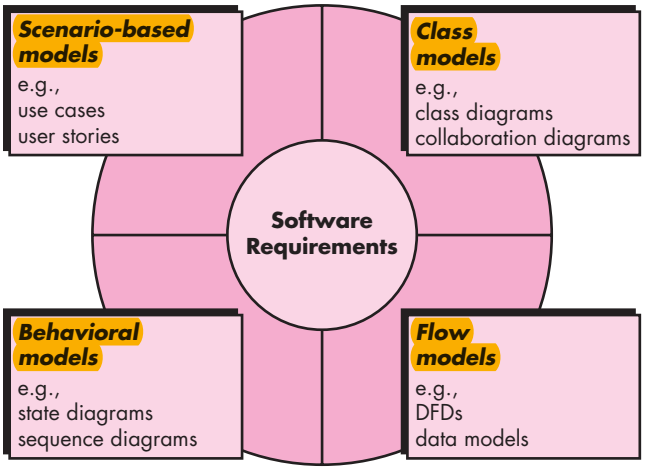
6.1.4 Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis,* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process (Chapter 2) are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what

FIGURE 6.3
Elements of the analysis model



? What different points of view can be used to describe the requirements model?

Quote:

"Why should we build models? Why not just build the system itself? The answer is that we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system."

Ed Yourdon

combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

6.2 SCENARIO-BASED MODELING

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design

models. Hence, requirements modeling with UML[6] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

6.2.1 Creating a Preliminary Use Case

Quote:

"[Use cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)."

Ivar Jacobson

Alistair Cockburn characterizes a use case as a "contract for behavior" [Coc01b]. As we discussed in Chapter 5, the "contract" defines the way in which an actor[7] uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, I examine how use cases are developed as part of the requirements modeling activity.[8]

In Chapter 5, I noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

ADVICE

In some situations, use cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard other modeling methods when they are appropriate.

What to write about? The first two requirements engineering tasks—inception and elicitation—provide you with the information you'll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 6.3.1) developed as part of requirements modeling.

SAFEHOME

Developing Another Preliminary User Scenario

The scene: A meeting room, during the second requirements gathering meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: It's time that we begin talking about the *SafeHome* surveillance function. Let's develop a user scenario for access to the surveillance function.

Jamie: Who plays the role of the actor on this?

---

6 UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.

7 An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor "calls on the system to deliver one of its services" [Coc01b].

8 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis is discussed in detail in Chapter 11.

Facilitator: I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

Meredith: You want to do it the same way we did it last time, right?

Facilitator: Right . . . same way.

Meredith: Well, obviously the reason for surveillance is to allow the homeowner to check out the house while he or she is away, to record and play back video that is captured . . . that sort of thing.

Ed: Will we use compression to store the video?

Facilitator: Good question, Ed, but let's postpone implementation issues for now. Meredith?

Meredith: Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we have to have tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

Facilitator (smiling): Took the words right out of my mouth.

Meredith: Um . . . I want to gain access to the surveillance function either via the PC or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

Jamie: Those are called thumbnail views.

Meredith: Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

Facilitator: Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 5 and reproduced later in this section as a sidebar.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views* **(ACS-DCV).** The stakeholder who takes on the role of the **homeowner** actor might write the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free-flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

Quote:

"Use cases can be used in many [software] processes. Our favorite is a process that is iterative and risk driven."

Geri Schneider and Jason Winters

6.2.2 Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98a]:

How do I examine alternative courses of action when I develop a use case?

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is "yes." Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is "yes." A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."[9] This error condition becomes a secondary scenario.

Is it possible that the actor will encounter some other behavior at this point? Again the answer to the question is "yes." As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

9 In this case, another actor, the **system administrator,** would have to configure the floor plan, install and initialize (e.g., assign an equipment ID) all cameras, and test each camera to be certain that it is accessible via the system and through the floor plan.

Each of the situations described in the preceding paragraphs is characterized as a use-case exception. An *exception* describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends using a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be "rationalized" [Co01b] using the following criteria: an exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

6.2.3 Writing a Formal Use Case

The informal use cases presented in Section 6.2.1 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case. The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that "gets the use case started" [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 6.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

SAFEHOME

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Moderate frequency.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

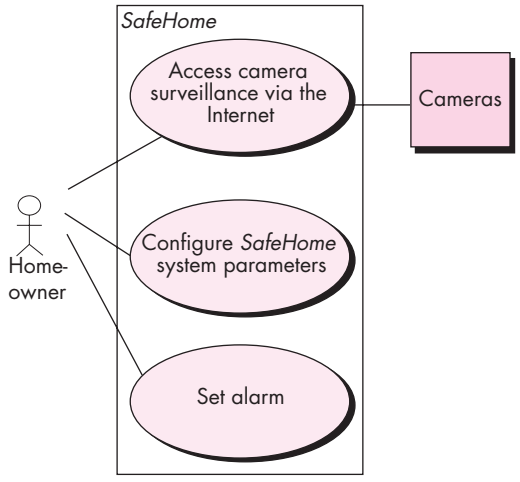
1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

WebRef

When are you finished writing use cases? For a worthwhile discussion of this topic, see **ootips.org/use-cases-done.html**.

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use-case diagramming capability. Figure 6.4 depicts a preliminary use-case diagram for the *SafeHome* product. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

FIGURE 6.4
Preliminary use-case diagram for the *SafeHome* system



Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

6.3 UML MODELS THAT SUPPLEMENT THE USE CASE

There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.

6.3.1 Developing an Activity Diagram

KEY POINT

A UML activity diagram represents the actions and decisions that occur as some function is performed.

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the **ACS-DCV** use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case.

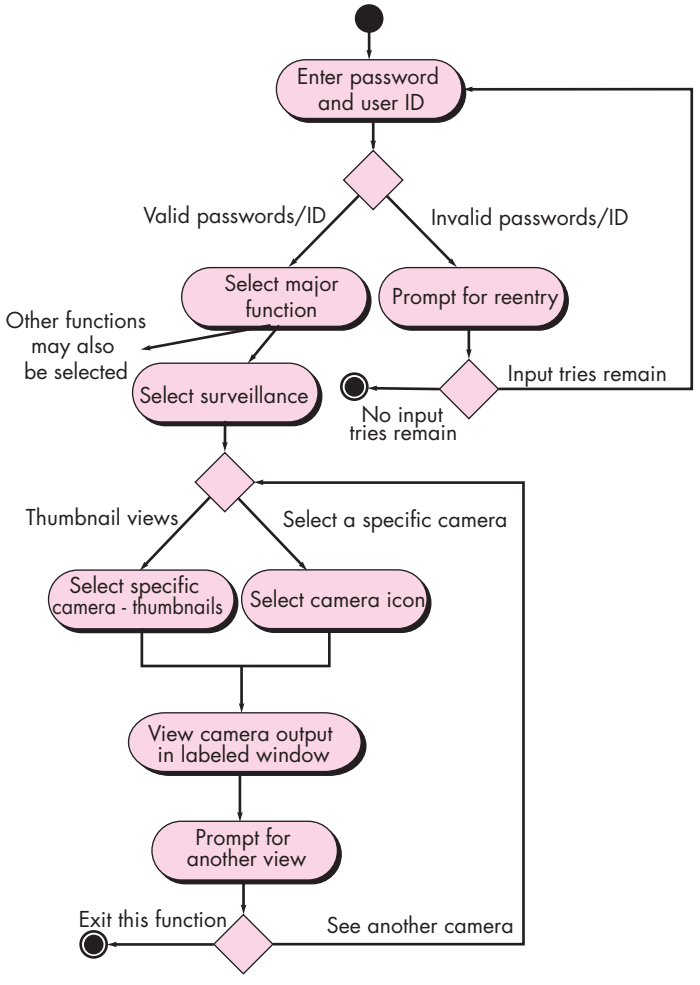


FIGURE 6.5
Activity diagram for Access camera surveillance via the Internet—display camera views function.

For example, a user may only attempt to enter **userID** and **password** a limited number of times. This is represented by a decision diamond below "Prompt for reentry."

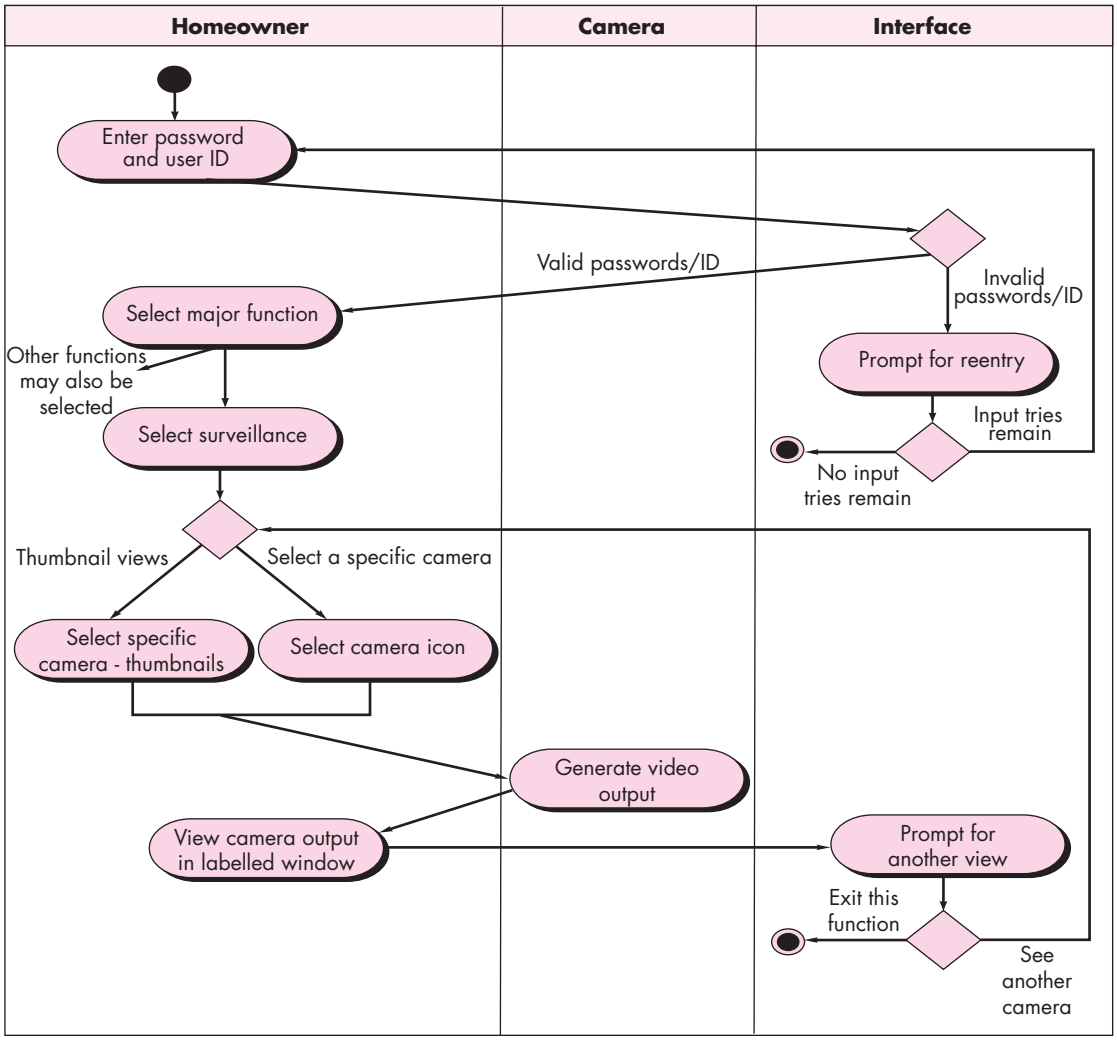
6.3.2 Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

KEY POINT
A UML swimlane diagram represents the flow of actions and decisions and indicates which actors perform each.

Three analysis classes—**Homeowner, Camera,** and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 6.5.

FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Quote:

"A good model guides your thinking, a bad one warps it."

Brian Marick

Referring to Figure 6.6, the activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the homeowner. The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the **Interface** swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions

(or other procedural steps) to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system. In Section 6.4, I examine the information space and how data requirements can be represented.

6.4 Data Modeling Concepts

WebRef
Useful information on data modeling can be found at **www.datamodel.org**.

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

6.4.1 Data Objects

? How does a data object manifest itself within the context of an application?

A *data object* is a representation of composite information that must be understood by software. By *composite information,* I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

KEY POINT
A data object is a representation of any composite information that is processed by software.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.[10] Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of **make, model, ID number, body type, color,** and **owner**. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car.**

KEY POINT
Attributes name a data object, describe its characteristics, and in some cases, make reference to another object.

6.4.2 Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier

10 This distinction separates the data object from the class or object defined as part of the object-oriented approach (Appendix 2).

FIGURE 6.7
Tabular representation of data objects

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car,** a reasonable identifier might be the **ID number.**

WebRef
A concept called "normalization" is important to those who intend to do thorough data modeling. A useful introduction can be found at **www.datamodel.org**.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include **ID number, body type,** and **color,** but many additional attributes (e.g., **interior code, drive train type, trim package designator, transmission type**) would have to be added to make **car** a meaningful object in the manufacturing control context.

INFO

Data Objects and Object-Oriented Classes—Are They the Same Thing?

A common question occurs when data objects are discussed: Is a data object the same thing as an object-oriented[11] class? The answer is "no."

A data object defines a composite data item; that is, it incorporates a collection of individual data items (attributes) and gives the collection of items a name (the name of the data object).

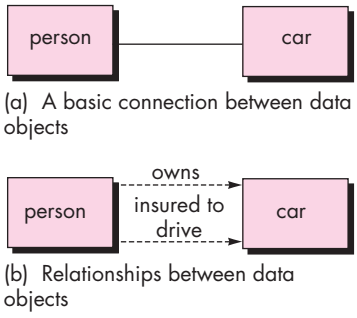
An object-oriented class encapsulates data attributes but also incorporates the operations (methods) that manipulate the data implied by those attributes. In addition, the definition of classes implies a comprehensive infrastructure that is part of the object-oriented software engineering approach. Classes communicate with one another via messages, they can be organized into hierarchies, and they provide inheritance characteristics for objects that are an instance of a class.

6.4.3 Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the simple notation

11 Readers who are unfamiliar with object-oriented concepts and terminology should refer to the brief tutorial presented in Appendix 2.

FIGURE 6.8
Relationships between data objects



KEY POINT
Relationships indicate the manner in which data objects are connected to one another.

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships. For example,

- A person *owns* a car.
- A person *is insured to drive* a car.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.** Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

INFO

Entity-Relationship Diagrams

The object-relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity-relationship diagram (ERD).[12] The ERD was originally proposed by Peter Chen [Che77] for the design of relational database systems and has been extended by others. A set of primary components is identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.[13] If you desire further information about data modeling and the entity-relationship diagram, see [Hob06] or [Sim05].

12 Although the ERD is still used in some database design applications, UML notation (Appendix 1) can now be used for data design.

13 The *cardinality* of an object-relationship pair specifies "the number of occurrences of one [object] that can be related to the number of occurrences of another [object]" [Til93]. The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

SOFTWARE TOOLS

Data Modeling

Objective: Data modeling tools provide a software engineer with the ability to represent data objects, their characteristics, and their relationships. Used primarily for large database applications and other information systems projects, data modeling tools provide an automated means for creating comprehensive entity-relation diagrams, data object dictionaries, and related models.

Mechanics: Tools in this category enable the user to describe data objects and their relationships. In some cases, the tools use ERD notation. In others, the tools model relations using some other mechanism. Tools in this category are often used as part of database design and enable the creation of a database model by generating a database schema for common database management systems (DBMS).

Representative Tools:[14]

AllFusion ERWin, developed by Computer Associates (**www3.ca.com**), assists in the design of data objects, proper structure, and key elements for databases.

ER/Studio, developed by Embarcadero Software (**www.embarcadero.com**), supports entity-relationship modeling.

Oracle Designer, developed by Oracle Systems (**www.oracle.com**), "models business processes, data entities and relationships [that] are transformed into designs from which complete applications and databases are generated."

Visible Analyst, developed by Visible Systems (**www.visible.com**), supports a variety of analysis modeling functions including data modeling.

6.5 CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The sections that follow present a series of informal guidelines that will assist in their identification and representation.

6.5.1 Identifying Analysis Classes

Quote:

"The really hard problem is discovering what are the right objects [classes] in the first place."

Carl Argila

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you "look around" the problem space of a software application, the classes (and objects) may be more difficult to comprehend.

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "grammatical parse" [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

14 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.

But what should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

? How do analysis classes manifest themselves as elements of the solution space?

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

This categorization is but one of many that have been proposed in the literature.[15] For example, Budd [Bud96] suggests a taxonomy of classes that includes *producers* (sources) and *consumers* (sinks) of data, *data managers, view* or *observer classes*, and *helper classes*.

It is also important to note what classes or objects are not. In general, a class should never have an "imperative procedural name" [Cas89]. For example, if the developers of software for a medical imaging system defined an object with the name **InvertImage** or even **ImageInversion,** they would be making a subtle mistake. The **Image** obtained from the software could, of course, be a class (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the object. It is likely that inversion would be defined as an operation for the object **Image,** but it would not be defined as a separate class to connote "image inversion." As Cashman [Cas89] states: "the intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data."

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[16] for the *SafeHome* security function.

15 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 6.5.4.

16 A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor's point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.*

The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC ,or the browser window. Homeowner interaction takes the following form . . .

ADVICE

The grammatical parse is not foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

Extracting the nouns, we can propose a number of potential classes:

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that I call each entry in the list a potential object. You must consider each further before a final decision is made.

? How do I determine whether a potential class should, in fact, become an analysis class?

Coad and Yourdon [Coa91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

1. *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.

3. *Multiple attributes.* During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. *Common operations.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

Quote:

"Classes struggle, some classes triumph, others are eliminated."

Mao Zedong

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. The decision for inclusion of potential classes in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated. However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must be made. With this in mind, you should apply the selection characteristics to the list of potential *SafeHome* classes:

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that (1) the preceding list is not all-inclusive, additional classes would have to be added to complete the model; (2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., **number** and **type** are attributes of **Sensor,** and **master password** and **telephone number** may become attributes of **System**); (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

6.5.2 Specifying Attributes

KEY POINT

Attributes are the set of data objects that fully define the class within the context of the problem.

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be quite different than the attributes of the same class when it is used in the context of the professional baseball pension system. In the former, attributes such as **name, position, batting average, fielding percentage, years played,** and **games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary, credit toward full vesting, pension plan options chosen, mailing address,** and the like.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class. In addition, the following question should be answered for each class: "What data items (composite and/or elementary) fully define this class in the context of the problem at hand?"

To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

identification information = system ID + verification phone number + system status
alarm response information = delay time + telephone number
activation/deactivation information = master password + number of allowable tries + temporary password

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 6.9).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 6.9. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

ADVICE

When you define operations for an analysis class, focus on problem-oriented behavior rather than behaviors required for implementation.

6.5.3 Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state

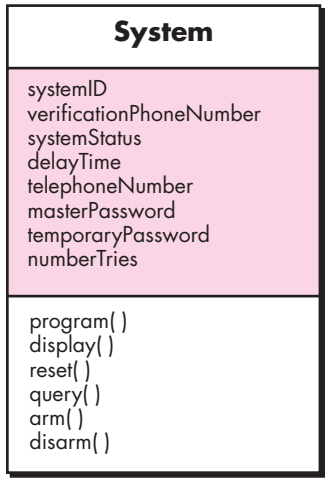


FIGURE 6.9
Class diagram for the system class

of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations (Section 6.5.5). Therefore, an operation must have "knowledge" of the nature of the class' attributes and associations.

As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is *assigned* a number and type" or "a master password is *programmed* for *arming and disarming* the system." These phrases indicate a number of things:

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

Upon further investigation, it is likely that the operation *program()* will be divided into a number of more specific suboperations required to configure the system. For example, *program()* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program()* as a single operation.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, I explore this matter in a bit more detail.

SAFEHOME

Class Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

The conversation:

[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

Ed: So when the homeowner wants to pick a camera, he or she has to pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 6.10.)

Jamie: So **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

Ed: Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 6.5.5.]

Vinod: So the actual floor plan is made up of walls and contains cameras and sensors that are placed within those walls. How does the floor plan know where to put those objects?

Ed: It doesn't, but the other classes do. See the attributes under, say, **WallSegment,** which is used to build a wall. The wall segment has start and stop coordinates and the *draw()* operation does the rest.

Jamie: And the same goes for windows and doors. Looks like camera has a few extra attributes.

Ed: Yeah, I need them to provide pan and zoom info.

Vinod: I have a question. Why does the camera have an ID but the others don't? I notice you have an attribute called **nextWall.** How will **WallSegment** know what the next wall will be?

Ed: Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

Jamie: Give me a break . . . I'll bet you've already figured it out.

Ed (smiling sheepishly): True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

Jamie: Looks pretty good to me, but I have a few more questions.

(Jamie asks questions which result in minor modifications)

Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

Ed: I'm not quite sure how to do them.

Vinod: It's not hard and they really pay off. I'll show you.

Quote:

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code."

C. Horstmann

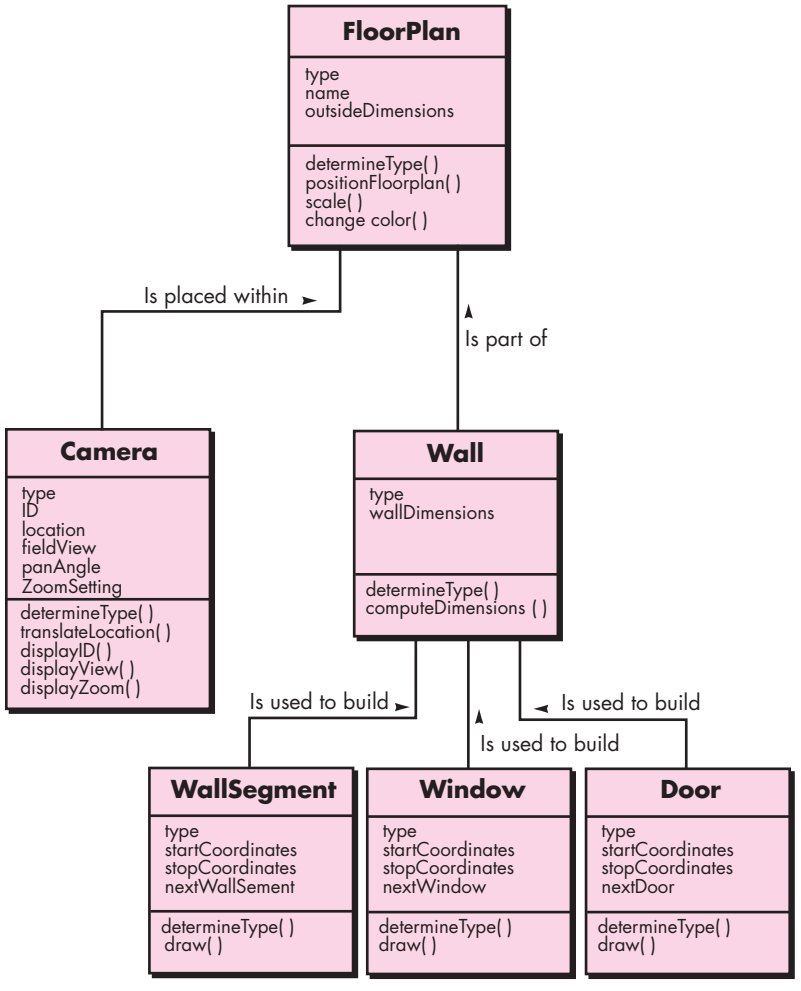
6.5.4 Class-Responsibility-Collaborator (CRC) Modeling

Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:

> A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [Amb95]. *Collaborators* are those classes that are

FIGURE 6.10
Class diagram for FloorPlan (see sidebar discussion)



required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action.

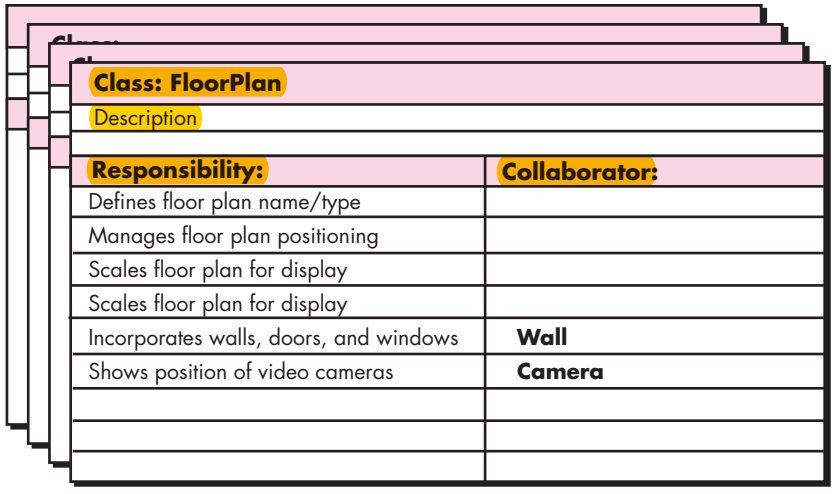
A simple CRC index card for the **FloorPlan** class is illustrated in Figure 6.11. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

WebRef
An excellent discussion of these class types can be found at **www.theumlcafe.com/a0079.htm**.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 6.5.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**). These

FIGURE 6.11
A CRC model index card



Quote:

"Objects can be classified scientifically into three major categories: those that don't work, those that break down, and those that get lost."

Russell Baker

classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- *Controller classes* manage a "unit of work" [UML03] from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

? What guidelines can be applied for allocating responsibilities to classes?

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 6.5.2 and 6.5.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of

different ways. "Dumb" classes (those that have few responsibilities) can be modeled to act as servants to a few "smart" classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort.

If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.[17] This enhances the maintainability of the software and reduces the impact of side effects due to change.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.[18] In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks.* The first operation (responsibility) implies a complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry,** a class that is encompassed by the aggregate class **CheckingAccount.**

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.

17 Cohesiveness is a design concept that is discussed in Chapter 8.
18 In such cases, it may be necessary to spit the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.** Each of these classes has its own attributes (e.g., **position, orientation, color, speed**) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update()* and *display()* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update()* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes. Wirfs-Brock and her colleagues [Wir90] define collaborations in the following way:

> Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

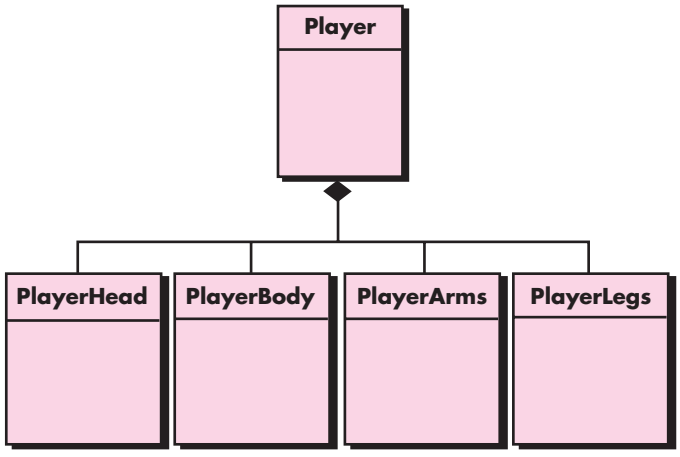
Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a **status** attribute to "not ready." Sensor information can be acquired from each **Sensor** object. Therefore, the responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor.**

To help in the identification of collaborators, you can examine three different generic relationships between classes [Wir90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player,** as are **PlayerArms, PlayerLegs,** and **PlayerHead.** In UML, these relationships are represented as the aggregation shown in Figure 6.12.

FIGURE 6.12
A composite aggregate class



When one class must acquire information from another class, the *has-knowledge-of* relationship is established. The *determine-sensor-status()* responsibility noted earlier is an example of a *has-knowledge-of* relationship.

The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of.* For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **PlayerHead** object called **center-position** is determined from the center position of **PlayerBody.** This information is obtained via a third object, **Player,** that acquires it from **PlayerBody.** Hence, **PlayerHead** *depends-upon* **PlayerBody.**

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 6.11).

When a complete CRC model has been developed, stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use case for *SafeHome* contains the following narrative:

   The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close

windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

When the review leader comes to "control panel," in the use case narrative, the token is passed to the person holding the **ControlPanel** index card. The phrase "implies that a sensor is open" requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status()* accomplishes this). Next to the responsibility on the index card is the collaborator **Sensor.** The token is then passed to the **Sensor** object.

4. When the token is passed, the holder of the **Sensor** card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use case is finished. When all use cases have been reviewed, requirements modeling continues.

SafeHome

CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

Vinod: While you've been working on surveillance and Jamie has been tied up with security, I've been working on the home management function.

Ed: What's the status of that? Marketing kept changing its mind.

Vinod: Here's the first-cut use case for the whole function . . . we've refined it a bit, but it should give you an overall view . . .

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a PC or an Internet connection to control electronic devices that have wireless interface controllers. The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, to set my heating and air conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is *home,* another is *away,* a third is *overnight travel,* and a fourth is *extended travel.* All of these situations will have settings that will be applied to all devices. In the *overnight travel* and *extended travel* states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

Ed: The hardware guys have got all the wireless interfacing figured out?

Vinod (smiling): They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

Ed: Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

Vinod: I thought you didn't understand CRC.

Ed: Maybe a little, but go ahead.

Vinod: So here's my class definition for **HomeManagementInterface.**

Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

Operations:

displayControl(), selectControl(), displaySituation(), select situation(), accessFloorplan(), selectDeviceIcon(), displayDevicePanel(), accessDevicePanel(), . . .

Class: HomeManagementInterface

Responsibility	**Collaborator**
displayControl()	**OptionsPanel** (class)
selectControl()	**OptionsPanel** (class)
displaySituation()	**SituationPanel** (class)
selectSituation()	**SituationPanel** (class)
accessFloorplan()	**FloorPlan** (class) . . .

. . .

Ed: So when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 6.10.)

Vinod: Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

Ed: Good way to find omissions or errors.

Vinod: Yep.

6.5.5 Associations and Dependencies

KEY POINT

An association defines a relationship between classes. Multiplicity defines how many of one class are related to how many of another class.

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another (Section 6.4.3). In UML these relationships are called *associations.* Referring back to Figure 6.10, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall.** The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment, Window,** and **Door.**

In some cases, an association may be further defined by indicating *multiplicity*. Referring to Figure 6.10, a **Wall** object is constructed from one or more **WallSegment** objects. In addition, the **Wall** object may contain 0 or more **Window** objects and 0 or more **Door** objects. These multiplicity constraints are illustrated in Figure 6.13, where "one or more" is represented using 1. .*, and "0 or more" by 0 . .*. In UML, the asterisk indicates an unlimited upper bound on the range.[19]

19 Other multiplicity relations—one to one, one to many, many to many, one to a specified range with lower and upper limits, and others—may be indicated as part of an association.

FIGURE 6.13
Multiplicity

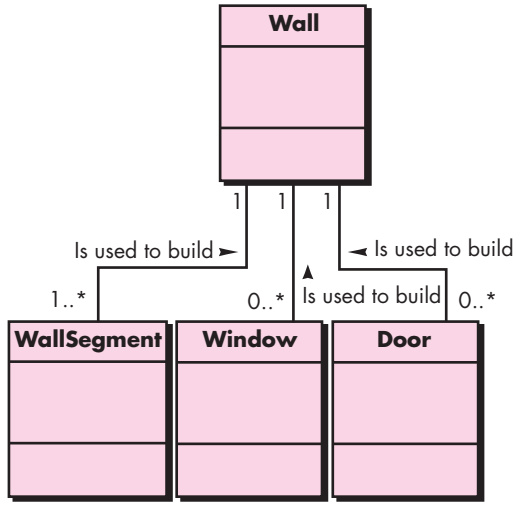
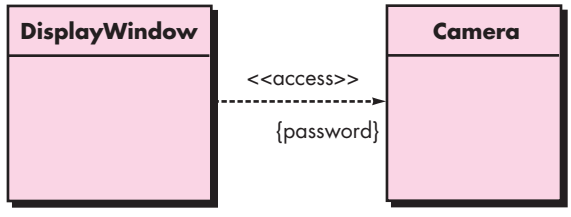


FIGURE 6.14
Dependencies



What is a stereotype?

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a *dependency relationship* is established. Dependencies are defined by a stereotype. A *stereotype* is an "extensibility mechanism" [Arl02] within UML that allows you to define a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., <<stereotype>>).

As an illustration of a simple dependency within the *SafeHome* surveillance system, a **Camera** object (in this case, the server class) provides a video image to a **DisplayWindow** object (in this case, the client class). The relationship between these two objects is not a simple association, yet a dependency association does exist. In a use case written for surveillance (not shown), you learn that a special password must be provided in order to view specific camera locations. One way to achieve this is to have **Camera** request a password and then grant permission to the **DisplayWindow** to produce the video display. This can be represented as shown in Figure 6.14 where <<access>> implies that the use of the camera output is controlled by a special password.

KEY POINT
A package is used to assemble a collection of related classes.

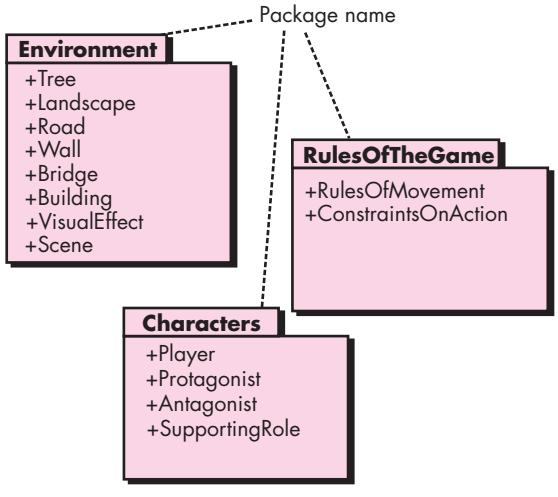
6.5.6 Analysis Packages

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that I introduced earlier. As the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as **Tree, Landscape, Road, Wall, Bridge, Building,** and **VisualEffect** might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. Classes such as **Player** (described earlier), **Protagonist, Antagonist,** and **SupportingRoles** might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **RulesOfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be grouped in analysis packages as shown in Figure 6.15.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

FIGURE 6.15
Packages



6.6 SUMMARY

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level representation that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the keys steps for a specific function or interaction. The degree of use-case formality and detail varies, but the end result provides necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a flowchart-like graphical representation that depicts the processing flow within a specific scenario. Swimlane diagrams illustrate how the processing flow is allocated to various actors or classes.

Data modeling is used to describe the information space that will be constructed or manipulated by the software. Data modeling begins by representing data objects—composite information that must be understood by the software. The attributes of each data object are identified and relationships between data objects are described.

Class-based modeling uses information derived from scenario-based and data modeling elements to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined. A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations, and dependencies among classes. Analysis packages are used to categorize and group classes in a manner that makes them more manageable for large systems.

PROBLEMS AND POINTS TO PONDER

6.1. Is it possible to begin coding immediately after an analysis model has been created? Explain your answer and then argue the counterpoint.

6.2. An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.

6.3. What is the purpose of domain analysis? How is it related to the concept of requirements patterns?

6.4. Is it possible to develop an effective analysis model without developing all four elements shown in Figure 6.3? Explain.

6.5. You have been asked to build one of the following systems:

a. a network-based course registration system for your university.
b. a Web-based order-processing system for a computer store.
c. a simple invoicing system for a small business.
d. an Internet-based cookbook that is built into an electric range or microwave.

Select the system that is of interest to you and develop an entity-relationship diagram that describes data objects, relationships, and attributes.

6.6. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

a. Draw a UML use case diagram for the PHTRS system. You'll have to make a number of assumptions about the manner in which a user interacts with this system.
b. Develop a class model for the PHTRS system.

6.7. Write a template-based use case for the *SafeHome* home management system described informally in the sidebar following Section 6.5.4.

6.8. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 6.5.

6.9. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

6.10. What is an analysis package and how might it be used?

Further Readings and Information Sources

Use cases can serve as the foundation for all requirements modeling approaches. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01b], and other references noted in both Chapters 5 and 6.

Data modeling presents a useful method for examining the information space. Books by Hoberman [Hob06] and Simsion and Witt [Sim05] provide reasonably comprehensive treatments. In addition, Allen and Terry (*Beginning Relational Data Modeling,* 2d ed., Apress, 2005), Allen (*Data Modeling for Everyone,* Wrox Press, 2002), Teorey and his colleagues (*Database Modeling and Design: Logical Design,* 4th ed., Morgan Kaufmann, 2005), and Carlis and Maguire (*Mastering Data Modeling,* Addison-Wesley, 2000) present detailed tutorials for creating

CHAPTER

17

SOFTWARE TESTING STRATEGIES

KEY CONCEPTS

A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses. Shooman [Sho83] discusses these issues:

> In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our

QUICK LOOK

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed. But how do you conduct the tests? Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you've already conducted as you add new components to a large system? When should you involve the customer? These and many other questions are answered when you develop a software testing strategy.

Who does it? A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering action. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps? Testing begins "in the small" and progresses "to the large." By this I mean that early testing focuses on a single component or a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

What is the work product? A *Test Specification* documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the types of tests that will be conducted.

How do I ensure that I've done it right? By reviewing the *Test Specification* prior to testing, you can assess the completeness of test cases and testing tasks. An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

integration testing 459
regression testing 467
system testing . . .470
unit testing 456
validation testing 467
V&V 450

> only defense against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and technical reviews] are helping us to reduce the number of initial errors that are inherent in the code. Similarly, different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

These "approaches and philosophies" are what I call *strategy*—the topic to be presented in this chapter. In Chapters 18 through 20, the testing methods and techniques that implement the strategy are presented.

17.1 A Strategic Approach to Software Testing

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which you can place specific test case design techniques and testing methods—should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews (Chapter 15). By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

WebRef
Useful resources for software testing can be found at **www.mtsu.edu/~storm/**.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

17.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of tasks that ensure that

software correctly implements a specific function. *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

> Verification: "Are we building the product right?"
>
> Validation: "Are we building the right product?"

The definition of V&V encompasses many software quality assurance activities (Chapter 16).[1]

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Quote:

"Testing is the unavoidable part of any responsible effort to develop a software system."

William Howden

ADVICE

Don't get sloppy and view testing as a "safety net" that will catch all errors that occurred because of weak software engineering practices. It won't. Stress quality and error detection throughout the software process.

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller [Mil77] relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

17.1.2 Organizing for Software Testing

Quote:

"Optimism is the occupational hazard of programming; testing is the treatment."

Kent Beck

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error-free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyzes, models, and then creates a computer program and its documentation. Like any builder, the software

1 It should be noted that there is a strong divergence of opinion about what types of testing constitute "validation." Some people believe that *all* testing is verification and that validation is conducted when requirements are reviewed and approved, and later, by the user when the system is operational. Other people view unit and integration testing (Sections 17.3.1 and 17.3.2) as verification and higher-order testing (Sections 17.6 and 17.7) as validation.

engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) destructive. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that you might infer erroneously from the preceding discussion: (1) that the developer of software should do no testing at all, (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly, (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

KEY POINT

An independent test group does not have the "conflict of interest" that builders of the software might experience.

The role of an *independent test group* (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

However, you don't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

Quote:

"The first mistake that people make is thinking that the testing team is responsible for assuring quality."

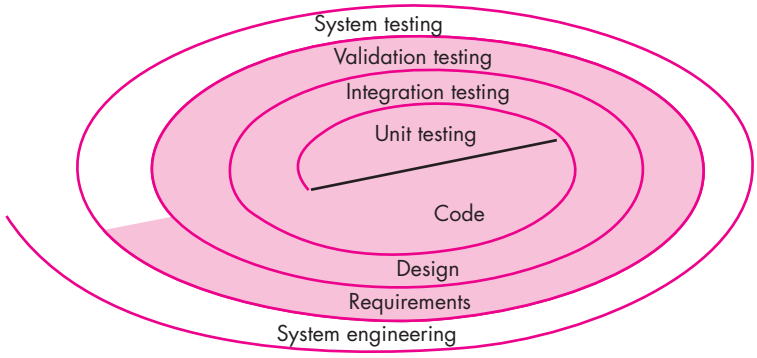
Brian Marick

The ITG is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering organization.

17.1.3 Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in Figure 17.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.

FIGURE 17.1
Testing strategy



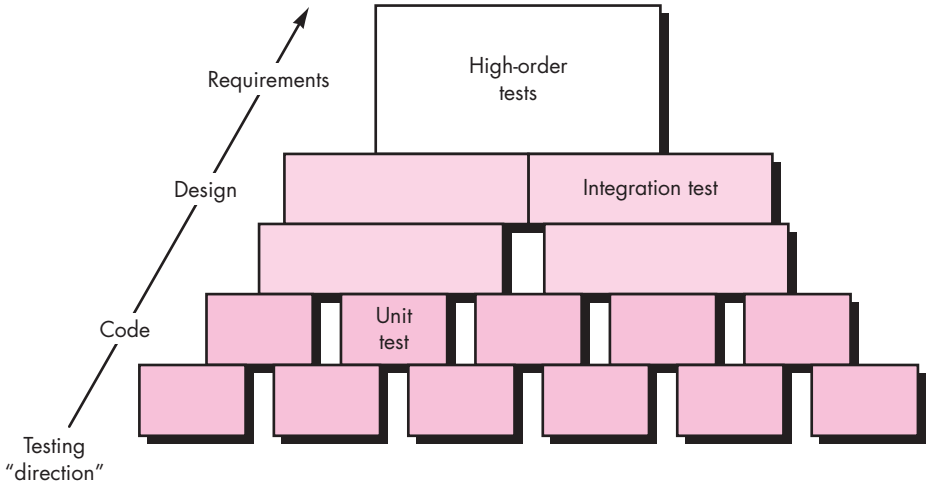
? What is the overall strategy for software testing?

A strategy for software testing may also be viewed in the context of the spiral (Figure 17.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter *validation testing,* where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

WebRef
Useful resources for software testers can be found at **www.SQAtester.com**.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 17.2. Initially, tests focus on each

FIGURE 17.2
Software testing steps



component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

SafeHome

Preparing for Testing

The scene: Doug Miller's office, as component-level design continues and construction of certain components begins.

The players: Doug Miller, software engineering manager, Vinod, Jamie, Ed, and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Doug: It seems to me that we haven't spent enough time talking about testing.

Vinod: True, but we've all been just a little busy. And besides, we have been thinking about it . . . in fact, more than thinking.

Doug (smiling): I know . . . we're all overloaded, but we've still got to think down the line.

Shakira: I like the idea of designing unit tests before I begin coding any of my components, so that's what I've been trying to do. I have a pretty big file of tests to run once code for my components is complete.

Doug: That's an Extreme Programming [an agile software development process, see Chapter 3] concept, no?

Ed: It is. Even though we're not using Extreme Programming per se, we decided that it'd be a good idea to design unit tests before we build the component—the design gives us all of the information we need.

Jamie: I've been doing the same thing.

Vinod: And I've taken on the role of the integrator, so every time one of the guys passes a component to me, I'll integrate it and run a series of regression tests on the partially integrated program. I've been working to design a set of appropriate tests for each function in the system.

Doug (to Vinod): How often will you run the tests?

Vinod: Every day . . . until the system is integrated . . . well, I mean until the software increment we plan to deliver is integrated.

Doug: You guys are way ahead of me!

Vinod (laughing): Anticipation is everything in the software biz, Boss.

17.1.4 Criteria for Completion of Testing

A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

? When are we finished testing?

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The *cleanroom software engineering* approach (Chapter 21) suggests statistical use techniques [Kel00] that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. Others (e.g., [Sin99]) advocate using statistical modeling and software reliability theory to predict the completeness of testing.

WebRef

A comprehensive glossary of testing terms can be found at **www.testingstandards.co.uk/living_glossary.htm**.

By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?" There is little debate that further work remains to be done before quantitative rules for testing can be established, but the empirical approaches that currently exist are considerably better than raw intuition.

17.2 Strategic Issues

Later in this chapter, I present a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [Gil95] argues that a software testing strategy will succeed when software testers:

? What guidelines lead to a successful software testing strategy?

Specify product requirements in a quantifiable manner long before testing commences. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability (Chapter 14). These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, mean-time-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.

WebRef

An excellent list of testing resources can be found at **www.io.com/~wazmo/qa/**.

Understand the users of the software and develop a profile for each user category. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Develop a testing plan that emphasizes "rapid cycle testing." Gilb [Gil95] recommends that a software team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Build "robust" software that is designed to test itself. Software should be designed in a manner that uses antibugging (Section 17.3.1) techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

Quote:

"Testing only to end-user requirements is like inspecting a building based on the work done by the interior designer at the expense of the foundations, girders, and plumbing."

Boris Beizer

Use effective technical reviews as a filter prior to testing. Technical reviews (Chapter 15) can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.

Conduct technical reviews to assess the test strategy and test cases themselves. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

17.3 Test Strategies for Conventional Software[2]

There are many strategies that can be used to test software. At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints all stakeholders. At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed. This approach, although less appealing to many, can be very effective. Unfortunately, some software developers hesitate to use it. What to do?

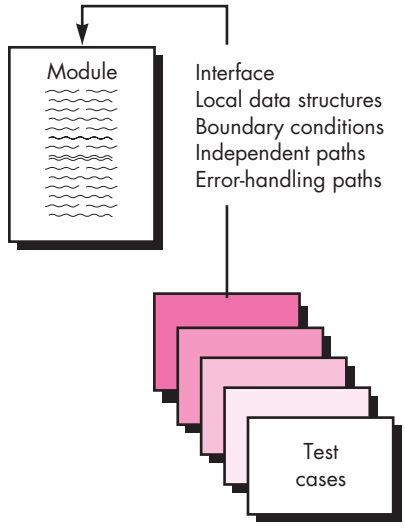
A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

17.3.1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a

2 Throughout this book, I use the terms *conventional software* or *traditional software* to refer to common hierarchical or call-and-return software architectures that are frequently encountered in a variety of application domains. Traditional software architectures are *not* object-oriented and do not encompass WebApps.

FIGURE 17.3
Unit test



guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

ADVICE

It's not a bad idea to design unit test cases before *you develop code for a component. It helps ensure that you'll develop code that will pass the tests.*

Unit-test considerations. Unit tests are illustrated schematically in Figure 17.3. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

? What errors are commonly found during unit testing?

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

WebRef

Useful information on a wide variety of articles and resources for "agile testing" can be found at **testing.com/agile/**.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon [You75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

> A computer-aided design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This "error message" was uncovered by a customer during user training!

Be sure that you design tests to execute every error-handling path. If you don't, the path may fail when it is invoked, exacerbating an already dicey situation.

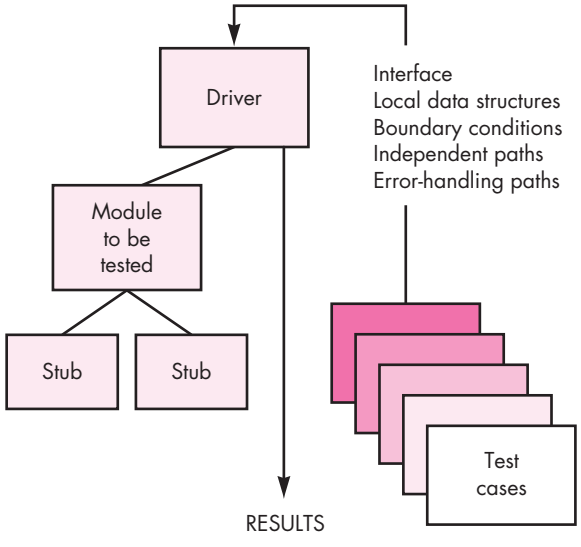
Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in Figure 17.4. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints

FIGURE 17.4

Unit-test environment



relevant results. *Stubs* serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

There are some situations in which you will not have the resources to do comprehensive unit testing. Select critical or complex modules and unit test only those.

Drivers and stubs represent testing "overhead." That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

17.3.2 Integration Testing

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

Taking the "big bang" approach to integration is a lazy strategy that is doomed to failure. Integrate incrementally, testing as you go.

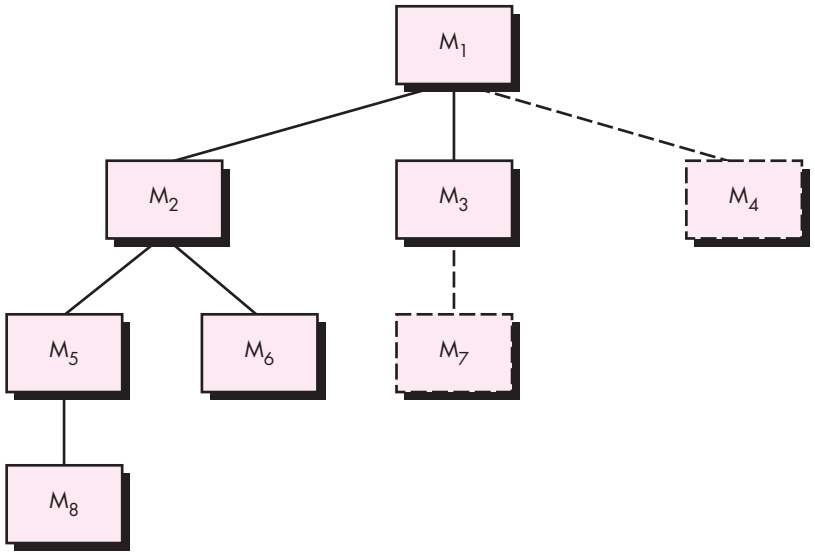
There is often a tendency to attempt nonincremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

When you develop a project schedule, you'll have to consider the manner in which integration will occur so that components will be available when needed.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed.

Top-down integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module

FIGURE 17.5
Top-down integration



(main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Referring to Figure 17.5, *depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M_1, M_2 , M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2, M_3, and M_4 would be integrated first. The next control level, M_5, M_6, and so on, follows. The integration process is performed in a series of five steps:

? What are the steps for top-down integration?

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a "well-factored" program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

? What problems may be encountered when top-down integration is chosen?

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. As a tester, you are left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) can cause you to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up integration is discussed in the paragraphs that follow.

Bottom-up integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

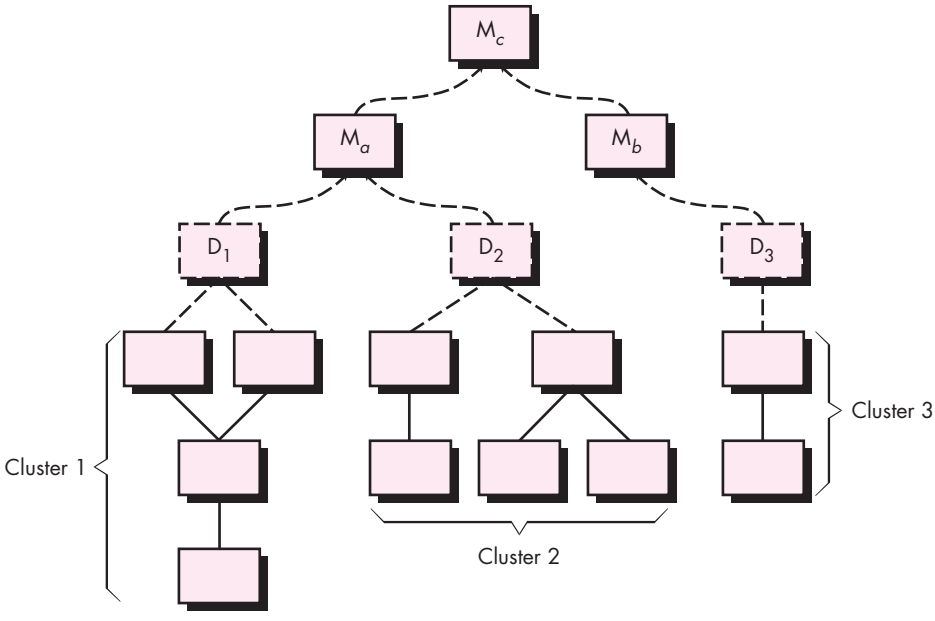
? What are the steps for bottom-up integration?

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.
2. A *driver* (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

KEY POINT
Bottom-up integration eliminates the need for complex stubs.

Integration follows the pattern illustrated in Figure 17.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

FIGURE 17.6
Bottom-up integration



As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression testing. Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

ADVICE

Regression testing is an important strategy for reducing "side effects." Run regression tests every time a major change is made to the software (including the integration of new components).

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent playback and comparison. The *regression test suite* (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to reexecute every test for every program function once a change has occurred.

Smoke testing. *Smoke testing* is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke-testing approach encompasses the following activities:

KEY POINT
Smoke testing might be characterized as a rolling integration strategy. The software is rebuilt (with new components added) and smoke tested every day.

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [McC96] describes the smoke test in the following manner:

> The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Quote:
"Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead."
Jim McCarthy

Smoke testing provides a number of benefits when it is applied on complex, time-critical software projects:

? What benefits can be derived from smoke testing?

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- *The quality of the end product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.

- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

WebRef

Pointers to commentary on testing strategies can be found at **www.qalinks.com**.

Strategic options. There has been much discussion (e.g., [Bei84]) about the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, the advantages of one strategy tend to result in disadvantages for the other strategy. The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the advantage of testing major control functions early. The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added" [Mye79]. This drawback is tempered by easier test case design and a lack of stubs.

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

? What is a "critical module" and why should we identify it?

As integration testing is conducted, the tester should identify critical modules. A *critical module* has one or more of the following characteristics: (1) addresses several software requirements, (2) has a high level of control (resides relatively high in the program structure), (3) is complex or error prone, or (4) has definite performance requirements. Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

Integration test work products. An overall plan for integration of the software and a description of specific tests is documented in a *Test Specification*. This work product incorporates a test plan and a test procedure and becomes part of the software configuration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for the *SafeHome* security system might be divided into the following test phases:

- *User interaction* (command input and output, display representation, error processing and representation)
- *Sensor processing* (acquisition of sensor output, determination of sensor conditions, actions required as a consequence of conditions)
- *Communications functions* (ability to communicate with central monitoring station)
- *Alarm processing* (tests of software actions that occur when an alarm is encountered)

Each of these integration test phases delineates a broad functional category within the software and generally can be related to a specific domain within the software architecture. Therefore, program builds (groups of modules) are created to correspond to each phase. The following criteria and corresponding tests are applied for all test phases:

? What criteria should be used to design integration tests?

Interface integrity. Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

Functional validity. Tests designed to uncover functional errors are conducted.

Information content. Tests designed to uncover errors associated with local or global data structures are conducted.

Performance. Tests designed to verify performance bounds established during software design are conducted.

A schedule for integration, the development of overhead software, and related topics are also discussed as part of the test plan. Start and end dates for each phase are established and "availability windows" for unit-tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results are also included.

A history of actual test results, problems, or peculiarities is recorded in a *Test Report* that can be appended to the *Test Specification,* if desired. Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

Like all other elements of a software configuration, the test specification format may be tailored to the local needs of a software engineering organization. It is important to note, however, that an integration strategy (contained in a test plan) and testing details (described in a test procedure) are essential ingredients and must appear.

17.4 Test Strategies for Object-Oriented Software[3]

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics (Chapter 19).

3 Basic object-oriented concepts are presented in Appendix 2.

17.4.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

Class testing for OO software is analogous to module testing for conventional software. It is not advisable to test operations in isolation.

You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation *X* is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation *X*, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a stand-alone fashion (the conventional unit-testing approach) is usually ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

17.4.2 Integration Testing in the OO Context

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies (Section 17.3.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [Ber93].

An important strategy for integration testing of OO software is thread-based testing. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

There are two different strategies for integration testing of OO systems [Bin94b]. The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation

of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

17.5 Test Strategies for WebApps

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

The overall strategy for WebApp testing can be summarized in the 10 steps noted here.

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

WebRef

Excellent articles on WebApp testing can be found at **www.stickyminds.com/testing.asp**.

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by support staff who use regression tests derived from the tests developed when the WebApp was first engineered. Methods for WebApp testing are considered in Chapter 20.

17.6 Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional software, object-oriented

software, and WebApps disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a *Software Requirements Specification* has been developed, it describes all user-visible attributes of the software and contains a *Validation Criteria* section that forms the basis for a validation-testing approach.

KEY POINT

Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user.

17.6.1 Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

17.6.2 Configuration Review

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit, is discussed in more detail in Chapter 22.

Quote:

"Given enough eyeballs, all bugs are shallow (e.g., given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone)."

E. Raymond

17.6.3 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

? What is the difference between an alpha test and a beta test?

The *alpha test* is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

A variation on beta testing, called *customer acceptance testing,* is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SafeHome

Preparing for Validation

The scene: Doug Miller's office, as component-level design continues and construction of certain components continues.

The players: Doug Miller, software engineering manager, Vinod, Jamie, Ed, and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Doug: The first increment will be ready for validation in what . . . about three weeks?

Vinod: That's about right. Integration is going well. We're smoke testing daily, finding some bugs but nothing we can't handle. So far, so good.

Doug: Talk to me about validation.

Shakira: Well, we'll use all of the use cases as the basis for our test design. I haven't started yet, but I'll be developing tests for all of the use cases that I've been responsible for.

Ed: Same here.

Jamie: Me too, but we've got to get our act together for acceptance test and also for alpha and beta testing, no?

Doug: Yes. In fact I've been thinking; we could bring in an outside contractor to help us with validation. I have the money in the budget . . . and it'd give us a new point of view.

Vinod: I think we've got it under control.

Doug: I'm sure you do, but an ITG gives us an independent look at the software.

Jamie: We're tight on time here, Doug. I for one don't have the time to babysit anybody you bring in to do the job.

Doug: I know, I know. But if an ITG works from requirements and use cases, not too much babysitting will be required.

Vinod: I still think we've got it under control.

Doug: I hear you, Vinod, but I going to overrule on this one. Let's plan to meet with the ITG rep later this week. Get 'em started and see what they come up with.

Vinod: Okay, maybe it'll lighten the load a bit.

17.7 System Testing

Quote:

"Like death and taxes, testing is both unpleasant and inevitable."

Ed Yourdon

At the beginning of this book, I stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, I discuss the types of system tests that are worthwhile for software-based systems.

17.7.1 Recovery Testing

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

17.7.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [Bei84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

17.7.3 Stress Testing

Earlier software testing steps resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

Quote:

"If you're trying to find true system bugs and you haven't subjected your software to a real stress test, then it's high time you started."

Boris Beizer

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing*. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

17.7.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be

assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

17.7.5 Deployment Testing

In many cases, software must execute on a variety of platforms and under more than one operating system environment. *Deployment testing,* sometimes called *configuration testing,* exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

As an example, consider the Internet-accessible version of *SafeHome* software that would allow a customer to monitor the security system from remote locations. The *SafeHome* WebApp must be tested using all Web browsers that are likely to be encountered. A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows). Because security is a major issue, a complete set of security tests would be integrated with the deployment test.

SOFTWARE TOOLS

Test Planning and Management

Objective: These tools assist a software team in planning the testing strategy that is chosen and managing the testing process as it is conducted.

Mechanics: Tools in this category address test planning, test storage, management and control, requirements traceability, integration, error tracking, and report generation. Project managers use them to supplement project scheduling tools. Testers use these tools to plan testing activities and to control the flow of information as the testing process proceeds.

Representative Tools:[4]

QaTraq Test Case Management Tool, developed by Traq Software (**www.testmanagement.com**), "encourages a structured approach to test management."

QADirector, developed by Compuware Corp. (**www.compuware.com/qacenter**), provides a single point of control for managing all phases of the testing process.

TestWorks, developed by Software Research, Inc. (**www.soft.com/Products/index.html**), contains a fully integrated suite of testing tools including tools for test management and reporting.

OpensourceTesting.org (**www.opensourcetesting.org/testmgt.php**) lists a variety of open-source test management and planning tools.

NI TestStand, developed by National Instruments Corp. (**www.ni.com**), allows you to "develop, manage, and execute test sequences written in any programming language."

4 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.

17.8 The Art of Debugging

Quote:

"We found to our surprise that it wasn't as easy to get programs right as we had thought. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

Maurice Wilkes, discovers debugging, 1949

Software testing is a process that can be systematically planned and specified. Test-case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. As a software engineer, you are often confronted with a "symptomatic" indication of a software problem as you evaluate the results of a test. That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

17.8.1 The Debugging Process

Debugging is not testing but often occurs as a consequence of testing.[5] Referring to Figure 17.7, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

ADVICE

Be certain to avoid a third outcome: The cause is found, but the "correction" does not solve the problem or introduces still another error.

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

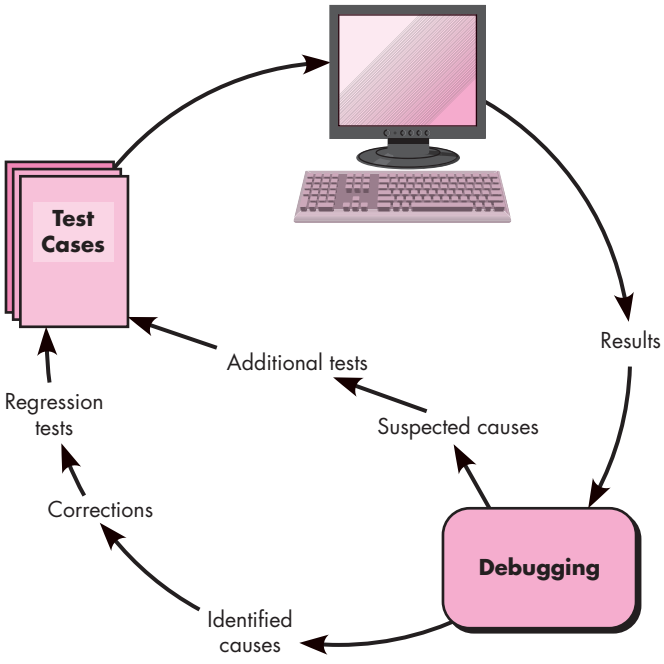
? Why is debugging so difficult?

Why is debugging so difficult? In all likelihood, human psychology (see Section 17.8.2) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 8) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.

5 In making the statement, we take the broadest possible view of testing. Not only does the developer test software prior to release, but the customer/user tests software every time it is used!

FIGURE 17.7
The debugging process



? "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"

Brian Kernighan

5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, you'll encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces some software developers to fix one error and at the same time introduce two more.

17.8.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging

ability have been reported for programmers with the same education and experience. Commenting on the human aspects of debugging, Shneiderman [Shn80] states:

> Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected.

Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. I examine them in Section 17.8.3.

SAFEHOME

Debugging

The scene: Ed's cubical as code and unit testing is conducted.

The players: Ed and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Shakira (looking in through the entrance to the cubical): Hey . . . where were you at lunchtime?

Ed: Right here . . . working.

Shakira: You look miserable . . . what's the matter?

Ed (sighing audibly): I've been working on this . . . bug since I discovered it at 9:30 this morning and it's what, 2:45 . . . I'm clueless.

Shakira: I thought we all agreed to spend no more than one hour debugging stuff on our own; then we get help, right?

Ed: Yeah, but . . .

Shakira (walking into the cubical): So what's the problem?

Ed: It's complicated, and besides, I've been looking at this for, what, 5 hours. You're not going to see it in 5 minutes.

Shakira: Indulge me . . . what's the problem?

[Ed explains the problem to Shakira, who looks at it for about 30 seconds without speaking, then . . .]

Shakira (a smile is gathering on her face): Uh, right there, the variable named *setAlarmCondition*. Shouldn't it be set to "false" before the loop gets started?

[Ed stares at the screen in disbelief, bends forward, and begins to bang his head gently against the monitor. Shakira, smiling broadly now, stands and walks out.]

17.8.3 Debugging Strategies

Set a time limit, say two hours, on the amount of time you spend trying to debug a problem on your own. After that, get help!

Regardless of the approach that is taken, debugging has one overriding objective—to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck. Bradley [Bra85] describes the debugging approach in this way:

> Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem's source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined.
>
> Take a simple non-software example: A lamp in my house does not work. If nothing in the house works, the cause must be in the main circuit breaker or outside; I look around

to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test.

In general, three debugging strategies have been proposed [Mye79]: (1) brute force, (2) backtracking, and (3) cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Quote:

"The first step in fixing a broken program is getting it to fail repeatably (on the simplest example possible)."

T. Duff

Debugging tactics. The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you'll find a clue that can lead to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated debugging. Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted. Hailpern and Santhanam [Hai02] summarize the state of these tools when they note, ". . . many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TOOLS

Debugging

Objective: These tools provide automated assistance for those who must debug software problems. The intent is to provide insight that may be difficult to obtain if approaching the debugging process manually.

Mechanics: Most debugging tools are programming language and environment specific.

Representative Tools:[6]

Borland Gauntlet, distributed by Borland (**www.borland.com**), assists in both testing and debugging.

Coverty Prevent SQS, developed by Coverty (**www.coverty.com**), provides debugging assistance for both C++ and Java.

C++Test, developed by Parasoft (**www.parasoft.com**), is a unit-testing tool that supports a full range of tests on C and C++ code. Debugging features assist in the diagnosis of errors that are found.

CodeMedic, developed by NewPlanet Software (**www.newplanetsoftware.com/medic/**), provides a graphical interface for the standard UNIX debugger, *gdb,* and implements its most important features. *gdb* currently supports C/C++, Java, PalmOS, various embedded systems, assembly language, FORTRAN, and Modula-2.

GNATS, a freeware application (**www.gnu.org/software/gnats/**), is a set of tools for tracking bug reports.

The people factor. Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.[7] A final maxim for debugging might be: "When all else fails, get help!"

17.8.4 Correcting the Error

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [Van89] suggests three simple questions that you should ask before making the "correction" that removes the cause of a bug:

Quote:

"The best tester isn't the one who finds the most bugs … the best tester is the one who gets the most bugs fixed."

Cem Kaner et al.

1. *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.
2. *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.

6 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.

7 The concept of pair programming (recommended as part of the Extreme Programming model discussed in Chapter 3) provides a mechanism for "debugging" as the software is designed and coded.

3. *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach (Chapter 16). If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

17.9 Summary

Software testing accounts for the largest percentage of technical effort in the software process. Regardless of the type of software you build, a strategy for systematic test planning, execution, and control begins by considering small elements of the software and moves outward toward the program as a whole.

The objective of software testing is to uncover errors. For conventional software, this objective is achieved through a series of test steps. Unit and integration tests concentrate on functional verification of a component and incorporation of components into the software architecture. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

The strategy for testing object-oriented software begins with tests that exercise the operations within a class and then moves to thread-based testing for integration. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

WebApps are tested in much the same way as OO systems. However, tests are designed to exercise content, functionality, the interface, navigation, and aspects of WebApp performance and security.

Unlike testing (a systematic, planned activity), debugging can be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

Problems and Points to Ponder

17.1. Using your own words, describe the difference between verification and validation. Do both make use of test-case design methods and testing strategies?

17.2. List some problems that might be associated with the creation of an independent test group. Are an ITG and an SQA group made up of the same people?

17.3. Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 17.1.3? What possible complications might arise for embedded systems?

17.4. Why is a highly coupled module difficult to unit test?

17.5. The concept of "antibugging" (Section 17.2.1) is an extremely effective way to provide built-in debugging assistance when an error is uncovered:

a. Develop a set of guidelines for antibugging.
b. Discuss advantages of using the technique.
c. Discuss disadvantages.

17.6. How can project scheduling affect integration testing?

17.7. Is unit testing possible or even desirable in all circumstances? Provide examples to justify your answer.

17.8. Who should perform the validation test—the software developer or the software user? Justify your answer.

17.9. Develop a complete test strategy for the *SafeHome* system discussed earlier in this book. Document it in a *Test Specification.*

17.10. As a class project, develop a *Debugging Guide* for your installation. The guide should provide language and system-oriented hints that have been learned through the school of hard knocks! Begin with an outline of topics that will be reviewed by the class and your instructor. Publish the guide for others in your local environment.

Further Readings and Information Sources

Virtually every book on software testing discusses strategies along with methods for test-case design. Everett and Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller and his colleagues (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2d ed., Auerbach, 2004), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), and Kaner and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

For those readers with interest in agile software development methods, Crispin and House (*Testing Extreme Programming,* Addison-Wesley, 2002) and Beck (*Test Driven Development: By Example,* Addison-Wesley, 2002) present testing strategies and tactics for Extreme Programming. Kamer and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) present a collection of over 300 pragmatic "lessons" (guidelines) that every software tester should learn. Watkins (*Testing IT: An Off-the-Shelf Testing Process,* Cambridge University Press, 2001) establishes an effective testing framework for all types of developed and acquired software. Manges and O'Brien (*Agile Testing with Ruby and Rails,* Apress, 2008) addresses testing strategies and techniques for the Ruby programming language and Web framework.

Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir and Goel (*Testing Object-Oriented Software,* Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), Kung and his colleagues (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998), and Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) present strategies and methods for testing OO systems.

Guidelines for debugging are contained in books by Grötker and his colleagues (*The Developer's Guide to Debugging,* Springer, 2008), Agans (*Debugging,* Amacon, 2006), Zeller (*Why Programs Fail: A Guide to Systematic Debugging,* Morgan Kaufmann, 2005), Tells and Hsieh (*The Science of Debugging,* The Coreolis Group, 2001), and Robbins (*Debugging Applications,* Microsoft Press, 2000). Kaspersky (*Hacker Debugging Uncovered,* A-List Publishing, 2005) addresses the technology of debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice-Hall, 2002) presents techniques for managing defects that are encountered in

object-oriented systems. Beizer [Bei84] presents an interesting "taxonomy of bugs" that can lead to effective methods for test planning.

Books by Madisetti and Akgul (*Debugging Embedded Systems,* Springer, 2007), Robbins (*Debugging Microsoft .NET 2.0 Applications,* Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning,* Prentice-Hall, 2005), Ford and Teorey (*Practical Debugging in C++,* Prentice-Hall, 2002), Brown (*Debugging Perl,* McGraw-Hill, 2000), and Mitchell (*Debugging Java,* McGraw-Hill, 2000) address the special nature of debugging for the environments implied by their titles.

A wide variety of information sources on software testing strategies are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software testing strategies can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

18 TESTING CONVENTIONAL APPLICATIONS

KEY CONCEPTS

Testing presents an interesting anomaly for software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Beizer [Bei90] describes this situation effectively when he states:

> There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top-down design, . . . then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic? For not solving human communications problems that have been kicked around . . . for forty centuries?

Should testing instill guilt? Is testing really destructive? The answer to these questions is "No!"

In this chapter, I discuss techniques for software test-case design for conventional applications. Test-case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives and the testing strategies discussed in Chapter 17.

QUICK LOOK

What is it? Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic and interfaces of every software component and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Why is it important? Reviews and other SQA actions can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? For conventional applications, software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test-case design techniques and (2) software requirements are exercised using "black box" test-case design techniques. Use cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.

What is the work product? A set of test cases designed to exercise both internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness. In addition, you can evaluate test coverage and track error detection activities.

18.1 Software Testing Fundamentals

Quote:

"Every program does something right, it just may not be the thing we want it to do."

Author unknown

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach[1] provides the following definition for testability: "*Software testability* is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

? What are the characteristics of testability?

Operability. "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability. "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and

1 The paragraphs that follow are used with permission of James Bach (copyright 1994) and have been adapted from material that originally appeared in a posting in the newsgroup comp.software-eng.

variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

Quote:

"Errors are more common, more pervasive, and more troublesome in software than with other technologies."

David Parnas

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements); *structural simplicity* (e.g., architecture is modularized to limit the propagation of faults), and *code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.

Understandability. "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

You can use the attributes suggested by Bach to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing.

Test Characteristics. And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

? What is a "good" test?

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a graphical user interface is the failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

SafeHome

Designing Unique Tests

The scene: Vinod's cubical.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

Vinod: So these are the test cases you intend to run for the *passwordValidation* operation.

Ed: Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.

Vinod: So let's see . . . you note that the correct password will be 8080, right?

Ed: Uh huh.

Vinod: And you specify passwords 1234 and 6789 to test for error in recognizing invalid passwords?

Ed: Right, and I also test passwords that are close to the correct password, see . . . 8081 and 8180.

Vinod: Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

Ed: Well, they're different values.

Vinod: That's true, but if 1234 doesn't uncover an error . . . in other words . . . the *passwordValidation* operation notes that it's an invalid password, it's not likely that 6789 will show us anything new.

Ed: I see what you mean.

Vinod: I'm not trying to be picky here . . . it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.

Ed: Not a problem . . . I'll give this a bit more thought.

18.2 Internal and External Views of Testing

Quote:

"There is only one rule in designing test cases: cover all features, but do not make too many test cases."

Tsuneo Yamaura

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.[2]

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software. *White-box testing* of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

KEY POINT

White-box tests can be designed only after component-level design (or source code) exists. The logical details of the program must be available.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents

2 The terms *functional testing* and *structural testing* are sometimes used in place of black-box and white-box testing, respectively.

certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

INFO

Exhaustive Testing

Consider a 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules.

Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

18.3 White-Box Testing

Quote:

"Bugs lurk in corners and congregate at boundaries."

Boris Beizer

White-box testing, sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

18.4 Basis Path Testing

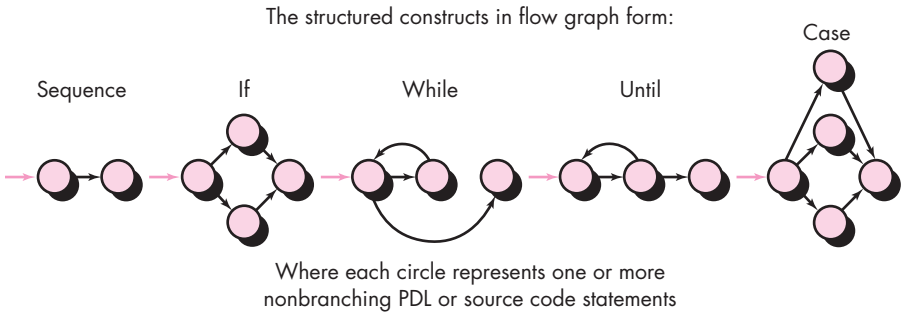
Basis path testing is a white-box testing technique first proposed by Tom McCabe [McC76]. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

18.4.1 Flow Graph Notation

Before we consider the basis path method, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.[3] The flow graph depicts logical control flow using the notation illustrated in Figure 18.1. Each structured construct (Chapter 10) has a corresponding flow graph symbol.

3 In actuality, the basis path method can be conducted without the use of flow graphs. However, they serve as a useful notation for understanding control flow and illustrating the approach.

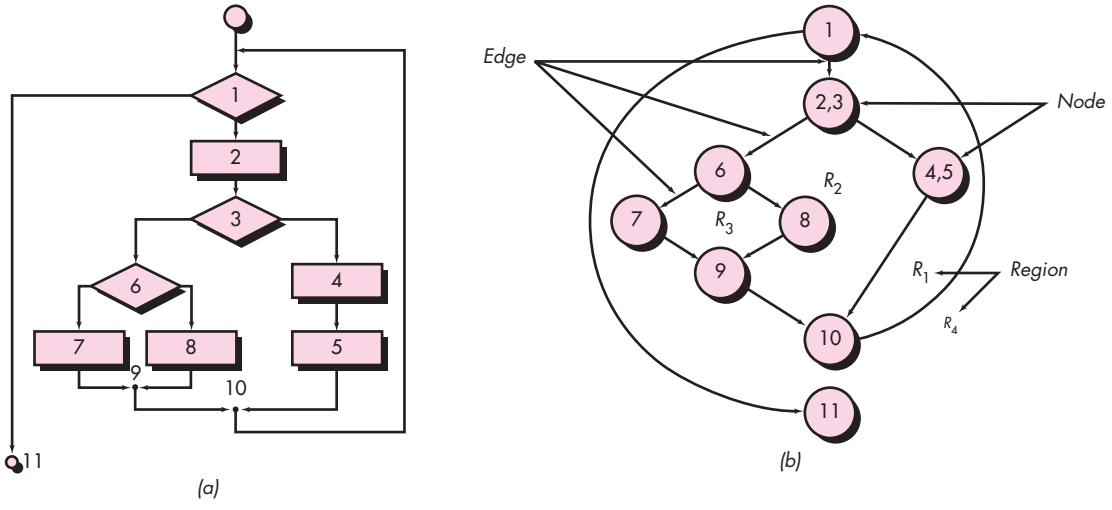
FIGURE 18.1 Flow graph notation



A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allows you to trace program paths more readily.

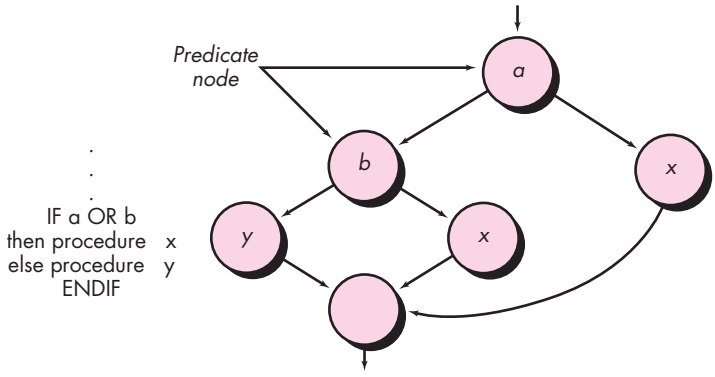
To illustrate the use of a flow graph, consider the procedural design representation in Figure 18.2a. Here, a flowchart is used to depict program control structure. Figure 18.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 18.2b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.[4]

FIGURE 18.2 (a) Flowchart and (b) flow graph



4 A more detailed discussion of graphs and their uses is presented in Section 18.6.1.

FIGURE 18.3
Compound logic



When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 18.3, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b.* Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

18.4.2 Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 18.2b is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 18.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that

the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do you know how many paths to look for? The computation of cyclomatic complexity provides the answer. *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

ADVICE

Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. Use it for test planning as well as test-case design.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

How do I compute cyclomatic complexity?

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

   $V(G) = E - N + 2$

   where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

   where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 18.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

KEY POINT

Cyclomatic complexity provides the upper bound on the number of test cases that will be required to guarantee that every statement in the program has been executed at least one time.

1. The flow graph has four regions.
2. $V(G)$ = 11 edges − 9 nodes + 2 = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 18.2b is 4.

More important, the value for $V(G)$ provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

SafeHome

Using Cyclomatic Complexity

The scene: Shakira's cubicle.

The players: Vinod and Shakira—members of the *SafeHome* software engineering team who are working on test planning for the security function.

The conversation:

Shakira: Look . . . I know that we should unit-test all the components for the security function, but there are a lot of 'em and if you consider the number of operations that

have to be exercised, I don't know . . . maybe we should forget white-box testing, integrate everything, and start running black-box tests.

Vinod: You figure we don't have enough time to do component tests, exercise the operations, and then integrate?

Shakira: The deadline for the first increment is getting closer than I'd like . . . yeah, I'm concerned.

Vinod: Why don't you at least run white-box tests on the operations that are likely to be the most error prone?

Shakira (exasperated): And exactly how do I know which are the most error prone?

Vinod: *V* of *G*.

Shakira: Huh?

Vinod: Cyclomatic complexity—*V* of *G*. Just compute *V*(*G*) for each of the operations within each of the components and see which have the highest values for *V*(*G*). They're the ones that are most likely to be error prone.

Shakira: And how do I compute *V* of *G*?

Vinod: It's really easy. Here's a book that describes how to do it.

Shakira (leafing through the pages): Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V*(*G*) will be the candidates for white-box tests.

Vinod: Just remember that there are no guarantees. A component with a low *V*(*G*) can still be error prone.

Shakira: Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

18.4.3 Deriving Test Cases

Quote:

"The Ariane 5 rocket blew up on lift-off due solely to a software defect (a bug) involving the conversion of a 64-bit floating point value into a 16-bit integer. The rocket and its four satellites were *uninsured* and worth $500 million. [Path tests that exercised the conversion path] would have found the bug but were vetoed for budgetary reasons."

A news report

The basis path testing method can be applied to a procedural design or to source code. In this section, I present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 18.4, will be used as an example to illustrate each step in the test-case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. **Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules presented in Section 18.4.1. Referring to the PDL for *average* in Figure 18.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is shown in Figure 18.5.
2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity *V*(*G*) is determined by applying the algorithms described in Section 18.4.2. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. Referring to Figure 18.5,

   V(*G*) = 6 regions
   V(*G*) = 17 edges − 13 nodes + 2 = 6
   V(*G*) = 5 predicate nodes + 1 = 6

FIGURE 18.4

PDL with nodes identified

```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;                    2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100   3
    4 increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum  6
5       THEN increment total.valid by 1;
7                 sum = s sum + value[i]
        ELSE skip
8     ENDIF
      increment i by 1;
9   ENDDO
    IF total.valid > 0   10
    11 THEN average = sum / total.valid;
12      ELSE average = -999;
    13 ENDIF
END average
```

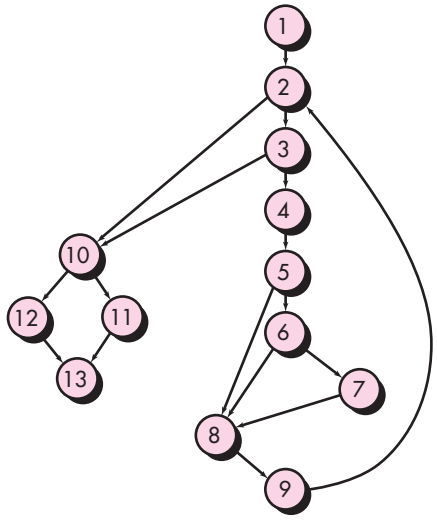
3. **Determine a basis set of linearly independent paths.** The value of *V*(*G*) provides the upper bound on the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

   Path 1: 1-2-10-11-13

   Path 2: 1-2-10-12-13

FIGURE 18.5

Flow graph for the procedure *average*



Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. **Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

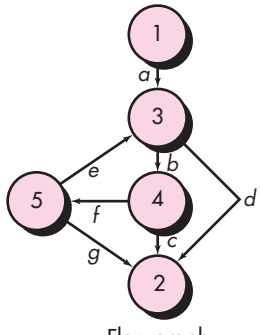
It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

18.4.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. A data structure, called a *graph matrix,* can be quite useful for developing a software tool that assists in basis path testing.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [Bei90] is shown in Figure 18.6.

FIGURE 18.6
Graph matrix



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*.

? What is a graph matrix and how do I extend it for use in testing?

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be execute.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

Beizer [Bei90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

18.5 Control Structure Testing

Quote:

"Paying more attention to running tests than to designing them is a classic mistake."

Brian Marick

The basis path testing technique described in Section 18.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

18.5.1 Condition Testing

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E_1 <relational-operator> E_2

KEY POINT

Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential processing statements.

where E_1 and E_2 are arithmetic expressions and <relational-operator> is one of the following: $<$, $\leq$, $=$, $\neq$ (nonequality), $>$, or $\geq$. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors

(incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

Quote:

"Good testers are masters at noticing 'something funny' and acting on it."

Brian Marick

18.5.2 Data Flow Testing

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number,

$$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$$
$$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$$

If statement *S* is an *if* or *loop statement,* its DEF set is empty and its USE set is based on the condition of statement *S.* The definition of variable *X* at statement *S* is said to be *live* at statement *S′* if there exists a path from statement *S* to statement *S′* that contains no other definition of *X.*

ADVICE

It is unrealistic to assume that data flow testing will be used extensively when testing a large system. However, it can be used in a targeted fashion for areas of software that are suspect.

A *definition-use (DU) chain* of variable *X* is of the form [*X, S, S′*], where *S* and *S′* are statement numbers, *X* is in DEF(*S*) and USE(*S′*), and the definition of *X* in statement *S* is live at statement *S′*.

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

18.5.3 Loop Testing

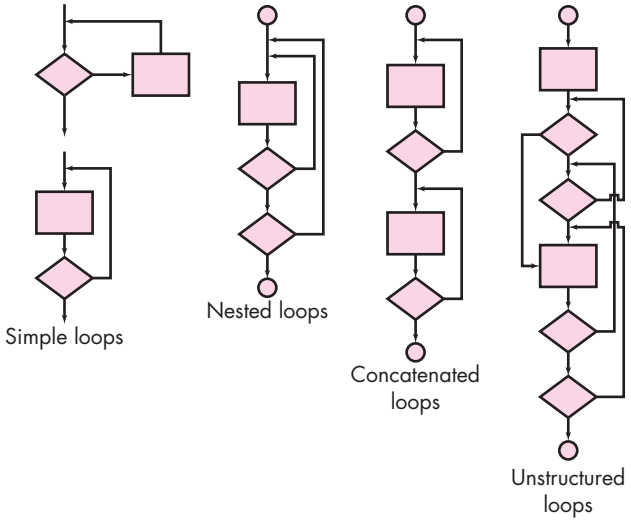
Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 18.7).

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.

FIGURE 18.7
Classes of Loops



4. m passes through the loop where $m < n$.
5. $n - 1$, n, $n + 1$ passes through the loop.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

You can't test unstructured loops effectively. Refactor them.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 10).

18.6 Black-Box Testing

Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Quote:

"To err is human, to find a bug is divine."

Robert Dunn

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing (see Chapter 17). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

? What questions do black-box tests answer?

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria [Mye79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

KEY POINT

A graph represents the relationships between data objects and program objects, enabling you to derive test cases that search for errors associated with these relationships.

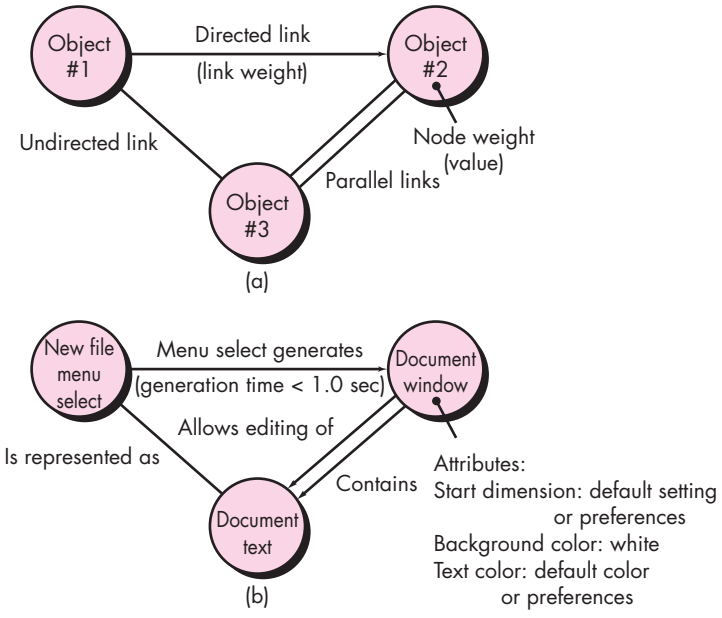
18.6.1 Graph-Based Testing Methods

The first step in black-box testing is to understand the objects[5] that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" [Bei95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and

5 In this context, you should consider the term *objects* in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

FIGURE 18.8

(a) Graph notation; (b) simple example



then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, you begin by creating a *graph*—a collection of *nodes* that represent objects, *links* that represent the relationships between objects, *node weights* that describe the properties of a node (e.g., a specific data value or state behavior), and *link weights* that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure 18.8a. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 18.8b) where

Object #1 = **newFile** (menu selection)

Object #2 = **documentWindow**

Object #3 = **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the

window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [Bei95] describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps (e.g., **flightInformationInput** is followed by *validationAvailabilityProcessing*). The data flow diagram (Chapter 7) can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **orderInformation** is verified during *inventoryAvailabilityLook-up* and is followed by **customerBillingInformation** input). The state diagram (Chapter 7) can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICA tax withheld (**FTW**) is computed from gross wages (**GW**) using the relationship, **FTW = 0.62 × GW.**

Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. If you have further interest, see [Bei95] for a comprehensive coverage.

18.6.2 Equivalence Partitioning

Input classes are known relatively early in the software process. For this reason, begin thinking about equivalence partitioning as the design is created.

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric,

transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

How do I define equivalence classes for testing?

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

18.6.3 Boundary Value Analysis

Quote:

"An effective way to test code is to exercise it at its natural boundaries."

Brian Kernighan

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

KEY POINT

BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class.

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b.*
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

18.6.4 Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

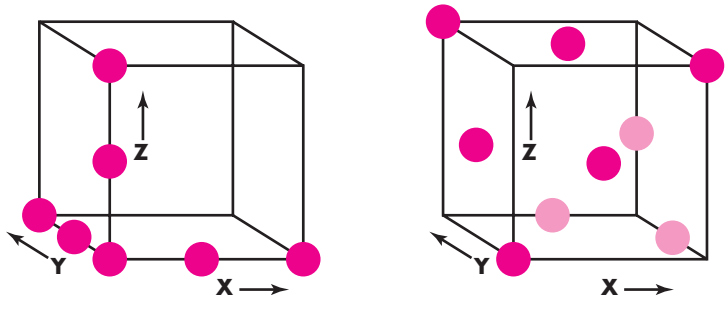
Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X, Y,* and *Z.* Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke [Pha97] suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property" [Pha97]. That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage across the input domain is more complete.

FIGURE 18.9
A geometric view of test cases
Source: [Pha97]



To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now
P1 = 2, send it one hour later
P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [Pha97] assesses these test cases by stating:

> Such test cases are useful only when one is certain that these test parameters do not interact. They can detect logic faults where a single parameter value makes the software malfunction. These faults are called *single mode faults.* This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values; that is, it cannot detect any interactions. Thus its ability to detect faults is limited.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is $3^4 = 81$, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax *send* function is illustrated in Figure 18.10.

FIGURE 18.10
An L9 orthogonal array

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke [Pha97] assesses the result of tests using the L9 orthogonal array in the following manner:

> **Detect and isolate all single mode faults.** A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 18.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.
>
> **Detect all double mode faults.** If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
>
> **Multimode faults.** Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

You can find a detailed discussion of orthogonal array testing in [Pha89].

SOFTWARE TOOLS

Test-Case Design

Objective: To assist the software team in developing a complete set of test cases for both black-box and white-box testing.

Mechanics: These tools fall into two broad categories: static testing tools and dynamic testing tools. Three different types of static testing tools are used in the industry: code-based testing tools, specialized testing languages, and requirements-based testing tools. Code-based testing tools accept source code as input and perform a number of analyses that result in the generation of test cases. Specialized testing languages (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. Requirements-based testing tools isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements. Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program.

Representative Tools:[6]

McCabeTest, developed by McCabe & Associates (**www.mccabe.com**), implements a variety of path testing techniques derived from an assessment of cyclomatic complexity and other software metrics.

TestWorks, developed by Software Research, Inc. (**www.soft.com/Products**), is a complete set of automated testing tools that assists in the design of tests cases for software developed in C/C++ and Java and provides support for regression testing.

T-VEC Test Generation System, developed by T-VEC Technologies (**www.t-vec.com**), is a tool set that supports unit, integration, and validation testing by assisting in the design of test cases using information contained in an OO requirements specification.

e-TEST Suite, developed by Empirix, Inc. (**www.empirix.com**), encompasses a complete set of tools for testing WebApps, including tools that assist test-case design and test planning.

6 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.

18.7 Model-Based Testing

Quote:

"It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."

Steve McConnell

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases. In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model (Chapter 7), as the basis for the design of test cases.[7] The MBT technique requires five steps:

1. **Analyze an existing behavioral model for the software or create one.** Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the steps discussed in Chapter 7: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system (e.g., see Figure 7.6), and (5) review the behavioral model to verify accuracy and consistency.
2. **Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.** The inputs will trigger events that will cause the transition to occur.
3. **Review the behavioral model and note the expected outputs as the software makes the transition from state to state.** Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model. "A fundamental assumption of this testing is that there is some mechanism, a *test oracle*, that will determine whether or not the results of a test execution are correct" [DAC03]. In essence, a test oracle establishes the basis for any determination of the correctness of the output. In most cases, the oracle is the requirements model, but it could also be another document or application, data recorded elsewhere, or even a human expert.
4. **Execute the test cases.** Tests can be executed manually or a test script can be created and executed using a testing tool.
5. **Compare actual and expected results and take corrective action as required.**

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

7 Model-based testing can also be used when software requirements are represented with decision tables, grammars, or Markov chains [DAC03].

18.8 Testing for Specialized Environments, Architectures, and Applications

Unique guidelines and approaches to testing are sometimes warranted when specialized environments, architectures, and applications are considered. Although the testing techniques discussed earlier in this chapter and in Chapters 19 and 20 can often be adapted to specialized situations, it's worth considering their unique needs individually.

18.8.1 Testing GUIs

Graphical user interfaces (GUIs) will present you with interesting testing challenges. Because reusable components are now a common part of GUI development environments, the creation of the user interface has become less time consuming and more precise (Chapter 11). But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. This model-based testing technique was discussed in Section 18.7.

Because of the large number of permutations associated with GUI operations, GUI testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.[8]

18.8.2 Testing of Client-Server Architectures

Quote:

"The topic of testing is one area in which a good deal of commonality exists between traditional system and client/server systems."

Kelley Bourne

The distributed nature of client-server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of client-server architectures and the software that resides within them considerably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when client-server environments are developed.

In general, the testing of client-server software occurs at three different levels: (1) Individual client applications are tested in a "disconnected" mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested.

WebRef

Useful client-sever testing information and resources can be found at **www.csst-technologies.com**.

8 Hundreds, if not thousands, of GUI testing tool resources can be evaluated on the Web. A good starting point for open-source tools is **www.opensourcetesting.org/functional.php**.

Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client-server applications:

What types of tests are conducted for client-server systems?

- **Application function tests.** The functionality of client applications is tested using the methods discussed earlier in this chapter and in Chapters 19 and 20. In essence, the application is tested in stand-alone fashion in an attempt to uncover errors in its operation.
- **Server tests.** The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.
- **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
- **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g., transaction processing times and transaction volume).
- **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

To accomplish these testing approaches, Musa [Mus93] recommends the development of *operational profiles* derived from client-server usage scenarios.[9] An operational profile indicates how different types of users interoperate with the client-server system. That is, the profiles provide a "pattern of usage" that can be applied when tests are designed and executed. For example, for a particular type of user, what percentage of transactions will be inquiries? updates? orders?

To develop the operational profile, it is necessary to derive a set of scenarios that are similar to use cases (Chapters 5 and 6). Each scenario addresses who, where, what, and why. That is, who the user is, where (in the physical client-server architecture) the system interaction occurs, what the transaction is, and why it has occurred. Scenarios can be derived using requirements elicitation techniques (Chapter 5) or through less formal discussions with end users. The result, however, should be the same. Each scenario should provide an indication of the system functions that will be required to service a particular user, the order in which those functions are required, the timing and response that is expected, and the frequency with which each function is used. These data are then combined (for all users) to create the operational profile. In general, testing effort and the number of test cases to be executed are

9 It should be noted that operational profiles can be used in testing for all types of system architectures, not just client-server architecture.

allocated to each usage scenario based on frequency of usage and criticality of the functions performed.

18.8.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. Recalling the definition of software presented in Chapter 1, it is important to note that testing must also extend to the third element of the software configuration—documentation.

Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an online help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that that documentation testing should be a meaningful part of every software test plan.

Documentation testing can be approached in two phases. The first phase, technical review (Chapter 15), examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

INFO

Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document's table of contents and index robust, accurate, and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

18.8.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test-case designer have to consider conventional test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the "copying" state). These same operator interrupts, if input when the machine is in the "jammed" state, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Comprehensive test-case design methods for real-time systems continue to evolve. However, an overall four-step strategy can be proposed:

? What is an effective strategy for testing a real-time system?

- **Task testing.** The first step in the testing of real-time software is to test each task independently. That is, conventional tests are designed for each task and executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.
- **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to equivalence partitioning (Section 18.6.2), events (e.g., interrupts, control signals) are categorized for testing. For example, events for the photocopier might be user interrupts (e.g., reset counter), mechanical interrupts (e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated). Each of these events is tested individually, and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events. The behavior of the system model (developed during the analysis activity) and the executable software can be compared for conformance. Once each class of events has been tested, events are presented to the system in random order and with random frequency. The behavior of the software is examined to detect behavior errors.

- **Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
- **System testing.** Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software-hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state diagram (Chapter 7), the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts. Tests are then designed to assess the following system characteristics:
  - Are interrupt priorities properly assigned and properly handled?
  - Is processing for each interrupt handled correctly?
  - Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
  - Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

18.9 PATTERNS FOR SOFTWARE TESTING

WebRef

A software testing patterns catalog can be found at **www.rbsc.com/pages/TestPatternList.htm**.

The use of patterns as a mechanism for describing solutions to specific design problems was discussed in Chapter 12. But patterns can also be used to propose solutions to other software engineering situations—in this case, software testing. *Testing patterns* describe common testing problems and solutions that can assist you in dealing with them.

Not only do testing patterns provide you with useful guidance as testing activities commence, they also provide three additional benefits described by Marick [Mar02]:

1. They [patterns] provide a vocabulary for problem-solvers. "Hey, you know, we should use a Null Object."
2. They focus attention on the forces behind a problem. That allows [test case] designers to better understand when and why a solution applies.
3. They encourage iterative thinking. Each solution creates a new context in which new problems can be solved.

Testing patterns can help a software team communicate more effectively about testing and better understand the forces that lead to a specific testing approach.

Although these benefits are "soft," they should not be overlooked. Much of software testing, even during the past decade, has been an ad hoc activity. If testing patterns can help a software team to communicate about testing more effectively;

WebRef

Patterns that describe testing organization, efficiency, strategy, and problem resolution can be found at: **www.testing.com/test-patterns/patterns/**.

to understand the motivating forces that lead to a specific approach to testing, and to approach the design of tests as an evolutionary activity in which each iteration results in a more complete suite of test cases, then patterns have accomplished much.

Testing patterns are described in much the same way as design patterns (Chapter 12). Dozens of testing patterns have been proposed in the literature (e.g., [Mar02]). The following three testing patterns (presented in abstract form only) provide representative examples:

Pattern name: **PairTesting**

Abstract: A process-oriented pattern, **PairTesting** describes a technique that is analogous to pair programming (Chapter 3) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Pattern name: **SeparateTestInterface**

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The **SeparateTestInterface** pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component" [Lan01].

Pattern name: **ScenarioTesting**

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement [Kan01].

A comprehensive discussion of testing patterns is beyond the scope of this book. If you have further interest, see [Bin99] and [Mar02] for additional information on this important topic.

18.10 Summary

The primary objective for test-case design is to derive a set of tests that have the highest likelihood for uncovering errors in software. To accomplish this objective, two different categories of test-case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure statement coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Hetzel [Het84] describes white-box testing as "testing in the small." His implication is that the white-box tests that have been considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens your focus and might be called "testing in the large."

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters. Model-based testing uses elements of the requirements model to test the behavior of an application.

Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client-server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test-case design, you can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

Problems and Points to Ponder

18.1. Myers [Mye79] uses the following program as a self-assessment for your ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

18.2. Design and implement the program (with error handling where appropriate) specified in Problem 18.1. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.

18.3. Can you think of any additional testing objectives that are not discussed in Section 18.1.1?

18.4. Select a software component that you have designed and implemented recently. Design a set of test cases that will ensure that all statements have been executed using basis path testing.

18.5. Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language of your choice. Use the graph matrix as the operative data structure in your design.

18.6. Read Beizer [Bei95] or a related Web-based source (e.g., **www.laynetworks.com/Discrete%20Mathematics_1g.htm**) and determine how the program you have developed in Problem 18.5 can be extended to accommodate various link weights. Extend your tool to process execution probabilities or link processing times.

18.7. Design an automated tool that will recognize loops and categorize them as indicated in Section 18.5.3.

18.8. Extend the tool described in Problem 18.7 to generate test cases for each loop category, once encountered. It will be necessary to perform this function interactively with the tester.

18.9. Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.

18.10. Will exhaustive testing (even if it is possible for very small programs) guarantee that the program is 100 percent correct?

18.11. Test a user manual (or help facility) for an application that you use frequently. Find at least one error in the documentation.

Further Readings and Information Sources

Virtually all books dedicated to software testing consider both strategy and tactics. Therefore, further readings noted for Chapter 17 are equally applicable for this chapter. Everett and Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller and his colleagues (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2d ed., Auerbach, 2004), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), and Whittaker (*How to Break Software,* Addison-Wesley, 2002) are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

A second edition of Myers [Mye79] classic text has been produced by Myers and his colleagues (*The Art of Software Testing,* 2d ed., Wiley, 2004) and covers test-case design techniques in considerable detail. Pezze and Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), Hutcheson (*Software Testing Fundamentals,* Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach,* 2d ed., CRC Press, 2002) each provide useful presentations of test-case design methods and techniques. Beizer's [Bei90] classic text provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [Bei95] presents a concise treatment of important methods.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Li and Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely and Posey (*Just Enough Software Test Automation,* Prentice-Hall, 2002); Dustin, Rashka, and Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); Graham and her colleagues (*Software Test Automation*, Addison-Wesley, 1999); and Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing. Nquyen and his colleagues (*Global Software Test Automation,* Happy About Press, 2006) present an executive overview of testing automation.

Thomas and his colleagues (*Java Testing Patterns,* Wiley, 2004) and Binder [Bin99] describe testing patterns that cover testing of methods, classes/clusters, subsystems, reusable components, frameworks, and systems as well as test automation and specialized database testing.

A wide variety of information sources on test-case design methods is available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing techniques can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PART

Four

MANAGING SOFTWARE PROJECTS

In this part of *Software Engineering: A Practitioner's Approach* you'll learn the management techniques required to plan, organize, monitor, and control software projects. These questions are addressed in the chapters that follow:

- How must people, process, and problem be managed during a software project?
- How can software metrics be used to manage a software project and the software process?
- How does a software team generate reliable estimates of effort, cost, and project duration?
- What techniques can be used to assess the risks that can have an impact on project success?
- How does a software project manager select a set of software engineering work tasks?
- How is a project schedule created?
- Why are maintenance and reengineering so important for both software engineering managers and practitioners?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product.

CHAPTER

24 PROJECT MANAGEMENT CONCEPTS

KEY CONCEPTS

In the preface to his book on software project management, Meiler Page-Jones [Pag85] makes a statement that can be echoed by many software engineering consultants:

> I've visited dozens of commercial shops, both good and bad, and I've observed scores of data processing managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be conducted

QUICK LOOK

What is it? Although many of us (in our darker moments) take Dilbert's view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to full operational deployment.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies among people involved in a software project. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software professionals.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four P's—people, product, process, and project. People must be organized to perform software work effectively. Communication with the customer and other stakeholders must occur so that product scope and requirements are understood. A process that is appropriate for the people and the product should be selected. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and identifying mechanisms to monitor and control work defined by the plan.

What is the work product? A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until you've delivered a high-quality product on time and within budget. However, a project manager does it right when he encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and Chapters 25 through 29, I'll present the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 25 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost are discussed in Chapter 26. Chapter 27 will help you to define a realistic project schedule. The management activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 28. Finally, Chapter 29 considers maintenance and reengineering and discusses the management issues that you'll encounter when you must deal with legacy systems.

24.1 The Management Spectrum

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

24.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives" [Cur01].

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

The People-CMM is a companion to the *Software Capability Maturity Model–Integration* (Chapter 30) that guides organizations in the creation of a mature

software process. Issues associated with people management and structure for software projects are considered later in this chapter.

24.1.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering (Chapter 5). Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

24.1.3 The Process

Those who adhere to the agile process philosophy (Chapter 3) argue that their process is leaner than others. That may be true, but they still have a process, and agile software engineering still requires discipline.

A software process (Chapters 2 and 3) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

24.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below

35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.[1]

To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 24.5 and in the chapters that follow.

24.2 People

In a study published by the IEEE [Cur88], the engineering vice presidents of three major technology companies were asked what was the most important contributor to a successful software project. They answered in the following way:

> **VP 1:** I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.
>
> **VP 2:** The most important ingredient that was successful on this project was having smart people . . . very little else matters in my opinion. . . . The most important thing you do for a project is selecting the staff. . . . The success of the software development organization is very, very much associated with the ability to recruit good people.
>
> **VP 3:** The only rule I have in management is to ensure I have good people—real good people—and that I grow good people—and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section I examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering.

24.2.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. *Senior managers* who define the business issues that often have a significant influence on the project.

1 Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, I think, is that a substantial number of these "failed" projects are ill conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

2. *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
3. *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
4. *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. *End users* who interact with the software once it is released for production use.

Every software project is populated by people who fall within this taxonomy.[2] To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

24.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers" [Edg95].

In an excellent book of technical leadership, Jerry Weinberg [Wei86] suggests an MOI model of leadership:

? What do we look for when choosing someone to lead a software project?

> **Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.
>
> **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
>
> **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem-solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Quote:

"In simplest terms, a leader is one who knows where he wants to go, and gets up, and goes."

John Erskine

Another view [Edg95] of the characteristics that define an effective project manager emphasizes four key traits:

> **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and

2 When WebApps are developed, other nontechnical people may be involved in content creation.

remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. She must demonstrate through her own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

24.2.3 The Software Team

Quote:

"Not every group is a team, and not every team is effective."

Glenn Parker

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes seven project factors that should be considered when planning the structure of software engineering teams:

? What factors should be considered when the structure of a software team is chosen?

- Difficulty of the problem to be solved
- "Size" of the resultant program(s) in lines of code or function points
- Time that the team will stay together (team lifetime)
- Degree to which the problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of sociability (communication) required for the project

Constantine [Con93] suggests four "organizational paradigms" for software engineering teams:

? What options do we have when defining the structure of a software team?

1. A *closed paradigm* structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.

2. A *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. An *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. A *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

Quote:

"If you want to be incrementally better: Be competitive. If you want to be exponentially better: Be cooperative."

Author unknown

As an historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the *chief programmer team.* This structure was first proposed by Harlan Mills and described by Baker [Bak72]. The nucleus of the team was composed of a *senior engineer* (the chief programmer), who plans, coordinates, and reviews all technical activities of the team; *technical staff* (normally two to five people), who conduct analysis and development activities; and a *backup engineer,* who supports the senior engineer in his or her activities and can replace the senior engineer with minimum loss in project continuity. The chief programmer may be served by one or more specialists (e.g., telecommunications expert, database designer), support staff (e.g., technical writers, clerical personnel), and a software librarian.

As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed *innovative anarchy.* Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, *Peopleware,* DeMarco and Lister [DeM98] discuss this issue:

> We tend to use the word team fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like

What is a "jelled" team?

> teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success. . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

Why is it that teams fail to jell?

But not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

Quote:

"Do or do not. There is no try."

Yoda from *Star Wars*

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built, the people doing the work, and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[3] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverts. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are

3 Technical reviews are discussed in detail in Chapter 15.

energized by the rush to make a last-minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.[4] However, it is important to note that recognition of human differences is the first step toward creating teams that jell.

24.2.4 Agile Teams

Over the past decade, agile software development (Chapter 3) has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity.

The small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write:

> If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job.

KEY POINT

An agile team is a self-organizing team that has autonomy to plan and make technical decisions.

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing.* A self-organizing team does not necessarily maintain a single team structure but instead uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 24.2.3.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

---

4 An excellent introduction to these issues as they relate to software project teams can be found in [Fer98].

Quote:

"Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the [agile] team, not individuals who make up the team."

Jim Highsmith

24.2.5 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

SAFEHOME

Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold on that for a moment. I'd like to talk about how we're going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and that fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): "We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

Doug: And, we're going to use UML as our modeling approach.

Vinod: But keep extraneous documentation to an absolute minimum.

Doug: Who is the liaison with me?

Jamie: We decided that Vinod will be the tech lead—he's got the most experience, so Vinod is your liaison, but feel free to talk to any of us.

Doug (laughing): Don't worry, I will.

24.3 The Product

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or even months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"

Like it or not, you must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

24.3.1 Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

If you can't bound a characteristic of the software you intend to build, list the characteristic as a project risk (Chapter 25).

Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, target environment, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described.

24.3.2 Problem Decomposition

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapters 6 and 7). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality and content (information) that must be delivered and (2) the process that will be used to deliver it.

In order to develop a reasonable project plan, you must decompose the problem. This can be accomplished using a list of functions or with use cases.

Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 26). Because both cost and schedule estimates are functionally oriented,

some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software.

As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as virtual keyboard input via a multitouch screen, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, and document production). For example, will continuous voice input require that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be and will it encompass the capabilities implied by a multitouch screen?

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), (4) the implementation of a style sheet feature that imposed consistency across a document, and (5) section and chapter reference validation for large documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier.

24.4 The Process

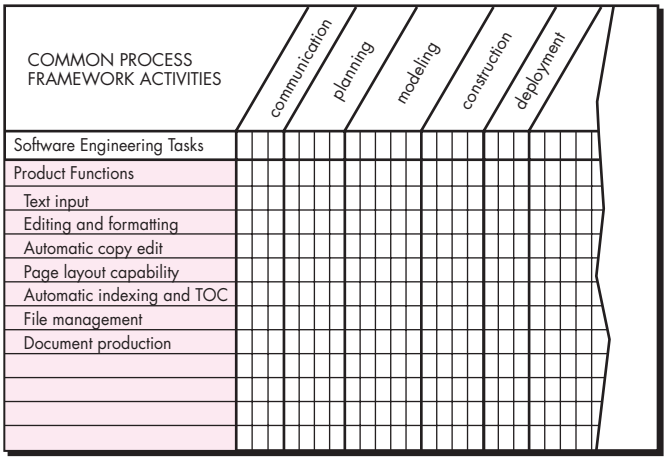
The framework activities (Chapter 2) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by your project team.

Your team must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 26.

24.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.

FIGURE 24.1
Melding the problem and the process



Assume that the organization has adopted the generic framework activities—**communication, planning, modeling, construction,** and **deployment**—discussed in Chapter 2. The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 24.1 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.[5] The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 26.

24.4.2 Process Decomposition

KEY POINT
The process framework establishes a skeleton for project planning. It is adapted by allocating a task set that is appropriate to the project.

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.[6]

5 It should be noted that work tasks must be adapted to the specific needs of the project based on a number of adaptation criteria.

6 Recall that project characteristics also have a strong bearing on the structure of the software team (Section 24.2.3).

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?" For example, a small, relatively simple project might require the following work tasks for the communication activity:

1. Develop list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the **communication**:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, functional, and behavioral features of the software. Alternatively, develop use cases that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use cases with all concerned.
10. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call **communication**, but the first project team performs half as many software engineering work tasks as the second.

24.5 The Project

In order to manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines 10 signs that indicate that an information systems project is in jeopardy:

? What are the signs that a software project is in jeopardy?

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Quote:

"We don't have time to stop for gas, we're already late."

M. Cleron

Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time [Zah94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [Ree99] suggests a five-part commonsense approach to software projects:

1. *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 24.2.3) and giving the team the autonomy, authority, and technology needed to do the job.
2. *Maintain momentum.* Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.[7]

7 The implication of this statement is that bureaucracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is deemphasized. The team should be self-organizing and autonomous.

Quote:

"A project is like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. But most projects worth doing are more like driving a truck off-road, in the mountains, at night."

Cem Kaner, James Bach, and Bret Pettichord

3. *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 25) can be collected and used to assess progress against averages developed for the software development organization.
4. *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).
5. *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

24.6 THE W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [Boe96] states: "you need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the *W^{5}HH Principle,* after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

? How do we define key project characteristics?

Why is the system being developed? All stakeholders should assess the validity of business reasons for the software work. Does the business purpose justify the expenditure of people, time, and money?

What will be done? The task set required for the project is defined.

When will it be done? The team establishes a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

Who is responsible for a function? The role and responsibility of each member of the software team is defined.

Where are they located organizationally? Not all roles and responsibilities reside within software practitioners. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 26) based on answers to earlier questions.

Boehm's W[5]HH Principle is applicable regardless of the size or complexity of a software project. The questions noted provide you and your team with an excellent planning outline.

24.7 Critical Practices

The Airlie Council[8] has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [Air99].

Critical practices[9] include: metric-based project management (Chapter 25), empirical cost and schedule estimation (Chapters 26 and 27), earned value tracking (Chapter 27), defect tracking against quality targets (Chapters 14 though 16), and people aware management (Section 24.2). Each of these critical practices is addressed throughout Parts 3 and 4 of this book.

Software Tools

Software Tools for Project Managers

The "tools" listed here are generic and apply to a broad range of activities performed by project managers. Specific project management tools (e.g., scheduling tools, estimating tools, risk analysis tools) are considered in later chapters.

Representative Tools:[10]

The Software Program Manager's Network (**www.spmn.com**) has developed a simple tool called *Project Control Panel*, which provides project managers with an direct indication of project status. The tool has "gauges" much like a dashboard and is implemented with Microsoft Excel. It is available for download at **www.spmn.com/products_software.html**.

Ganthead.com (**www.ganttthead.com/**) has developed a set of useful *checklists for project managers*.

Ittoolkit.com (**www.ittoolkit.com**) provides "a collection of planning guides, process templates and smart worksheets" available on CD-ROM.

8 The Airlie Council was comprised of a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering. For more on best practices, see **www.swqual.com/newsletter/vol1/no3/vol1no3.html**.

9 Only those critical practices associated with "project integrity" are noted here.

10 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.

24.8 Summary

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the modeling, construction, and deployment of computer software.

Four P's have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. Product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the product. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, technical reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. Each of these topics is considered in the chapters that follow.

Problems and Points to Ponder

24.1. Based on information contained in this chapter and your own experience, develop "ten commandments" for empowering software engineers. That is, make a list of 10 guidelines that will lead to software people who work to their full potential.

24.2. The Software Engineering Institute's People Capability Maturity Model (People-CMM) takes an organized look at "key practice areas" that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

24.3. Describe three real-life situations in which the customer and the end user are the same. Describe three situations in which they are different.

24.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

24.5. Review a copy of Weinberg's book [Wei86], and write a two- or three-page summary of the issues that should be considered in applying the MOI model.

24.6. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

24.7. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team structure would you choose and why? What software process model(s) would you choose and why?

24.8. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next-generation version of its widely used word-processing software. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

24.9. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal is to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

24.10. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

24.11. Do a first-level functional decomposition of the page layout function discussed briefly in Section 24.3.2.

Further Readings and Information Sources

The Project Management Institute (*Guide to the Project Management Body of Knowledge,* PMI, 2001) covers all important aspects of project management. Bechtold (*Essentials of Software Project Management,* 2d ed., Management Concepts, 2007), Wysocki (*Effective Software Project Management,* Wiley, 2006), Stellman and Greene (*Applied Software Project Management,* O'Reilly, 2005), and Berkun (*The Art of Project Management,* O'Reilly, 2005) teach basic skills and provide detailed guidance for all software project management tasks. McConnell (*Professional Software Development,* Addison-Wesley, 2004) offers pragmatic advice for achieving "shorter schedules, higher quality products, and more successful projects." Henry (*Software Project Management,* Addison-Wesley, 2003) offers real-world advice that is useful for all project managers.

Tom DeMarco and his colleagues (*Adrenaline Junkies and Template Zombies,* Dorset House, 2008) have written an insightful treatment of the human patterns that are encountered in every software project. An excellent four-volume series written by Weinberg (*Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts, explains how to use measurements effectively, and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Futrell and his colleagues (*Quality Software Project Management,* Prentice-Hall, 2002) present a voluminous treatment of project management. Brown and his colleagues (*Antipatterns in Project Management,* Wiley, 2000) discuss what not to do during the management of a software project.

Brooks (*The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. McConnell (*Software Project Survival Guide,* Microsoft Press, 1997) presents excellent pragmatic guidance for those who must manage software projects. Purba and Shah (*How to Manage a Successful Software Project,* 2d ed., Wiley, 2000) present a number of case studies that indicate why some projects succeed and others fail. Bennatan (*On Time Within Budget,* 3d ed., Wiley, 2000) presents useful tips and guidelines for software project managers. Weigers (*Practical Project Initiation,* Microsoft Press, 2007) provides practical guidelines for getting a software project off the ground successfully.

It can be argued that the most important aspect of software project management is people management. Cockburn (*Agile Software Development,* Addison-Wesley, 2002) presents one of

the best discussions of software people written to date. DeMarco and Lister [DeM98] have written the definitive book on software people and software projects. In addition, the following books on this subject have been published in recent years and are worth examining:

Cantor, M., *Software Leadership: A Guide to Successful Software Development,* Addison-Wesley, 2001.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software,* Prentice Hall, 2001.

Garton, C., and K. Wegryn, *Managing Without Walls,* McPress, 2006.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W. S., *TSP-Coaching Development Teams,* Addison-Wesley, 2006.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Peters, L., *Getting Results from Software Development Teams,* Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team,* Addison-Wesley, 2001.

Even though they do not relate specifically to the software world and sometimes suffer from oversimplification and broad generalization, best-selling "management" books by Kanter (*Confidence,* Three Rivers Press, 2006), Covy (*The 8th Habit,* Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done,* Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999), and Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy. Older titles such as *Who Moved My Cheese?*, *The One-Minute Manager,* and *In Search of Excellence* continue to provide valuable insights that can help you to manage people and projects more effectively.

A wide variety of information sources on the software project management are available on the Internet. An up-to-date list of World Wide Web references relevant to software project management can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

27 PROJECT SCHEDULING

In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking.

His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months.

He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going.

"Really great," said the young engineer with youthful enthusiasm. "This was much simpler than I thought. I'm probably close to 75 percent finished."

QUICK LOOK

What is it? You've selected an appropriate process model, you've identified the software engineering tasks that have to be performed, you estimated the amount of work and the number of people, you know the deadline, you've even considered the risks. Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time. Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality. In a nutshell, that's software project scheduling and tracking.

Who does it? At the project level, software project managers using information solicited from software engineers. At an individual level, software engineers themselves.

Why is it important? In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

What are the steps? The software engineering tasks dictated by the software process model are refined for the functionality to be built. Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established.

What is the work product? The project schedule and related information are produced.

How do I ensure that I've done it right? Proper scheduling requires that: (1) all tasks appear in the network, (2) effort and timing are intelligently allocated to each task, (3) interdependencies between tasks are properly indicated, (4) resources are allocated for the work to be done, and (5) closely spaced milestones are provided so that progress can be tracked.

time-line charts 732
tracking 734
work breakdown 732

The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time.

A week later the boss called the engineer into his office and asked, "Where are we?"

"Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon."

"How does the deadline look?" the boss asked.

"No problem," said the engineer. "I'm close to 90 percent complete."

If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer[1] stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late.

This story has been repeated tens of thousands of times by software developers during the past five decades. The big question is why?

27.1 Basic Concepts

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes:

- An unrealistic deadline established by someone outside the software team and forced on managers and practitioners.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Quote:

"Excessive or irrational schedules are probably the single most destructive influence in all of software."

Capers Jones

Aggressive (read "unrealistic") deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate, from the point of view of the person who sets the deadline. But common sense says that legitimacy must also be perceived by the people doing the work.

Napoleon once said: "Any commander-in-chief who undertakes to carry out a plan which he considers defective is at fault; he must put forth his reasons, insist on the plan being changed, and finally tender his resignation rather than be the instrument of his army's downfall." These are strong words that many software project managers should ponder.

1 In case you were wondering, this story is autobiographical.

The estimation activities discussed in Chapter 26 and the scheduling techniques described in this chapter are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should "protect his or her team from undue [schedule] pressure . . . [and] reflect the pressure back to its originators" [Pag85].

Quote:

"I love deadlines. I like the whooshing sound they make as they fly by."

Douglas Adams

To illustrate, assume that your software team has been asked to build a real-time controller for a medical diagnostic instrument that is to be introduced to the market in nine months. After careful estimation and risk analysis (Chapter 28), you come to the conclusion that the software, as requested, will require 14 calendar months to create with available staff. How should you proceed?

It is unrealistic to march into the customer's office (in this case the likely customer is marketing/sales) and demand that the delivery date be changed. External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work (from a career standpoint). So, what to do? I recommend the following steps in this situation:

? What should you do when management demands a deadline that is impossible?

1. Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model (Chapter 2), develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
3. Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percent improvement that would be required to achieve the deadline as it currently exists.[2] The following comment is appropriate:

   I think we may have a problem with the delivery date for the XYZ controller software. I've given each of you an abbreviated breakdown of development rates for past software projects and an estimate that we've done a number of different ways. You'll note that I've assumed a 20 percent improvement in past development rates, but we still get a delivery date that's 14 calendar months rather than 9 months away.

4. Offer the incremental development strategy as an alternative:

   We have a few options, and I'd like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we'll have a shot at getting this job done in nine months. But understand that this will increase the risk of poor quality due to the tight time line.[3] Second, we can remove a number of the software functions and capabilities that you're requesting. This will make the preliminary

2 If the required improvement is 10 to 25 percent, it may actually be possible to get the job done. But, more likely, the required improvement in team performance will be greater than 50 percent. This is an unrealistic expectation.

3 You might also add that increasing the number of people does not reduce calendar time proportionally.

version of the product somewhat less functional, but we can announce all functionality and then deliver over the 14-month period. Third, we can dispense with reality and wish the project complete in nine months. We'll wind up with nothing that can be delivered to a customer. The third option, I hope you'll agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster.

There will be some grumbling, but if a solid estimate based on good historical data is presented, it's likely that negotiated versions of option 1 or 2 will be chosen. The unrealistic deadline evaporates.

27.2 Project Scheduling

Fred Brooks was once asked how software projects fall behind schedule. His response was as simple as it was profound: "One day at a time."

The reality of a technical project (whether it involves building a hydroelectric plant or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the "critical path." If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

The tasks required to achieve a project manager's objective should not be performed manually. There are many excellent scheduling tools. Use them.

As a project manager, your objective is to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress to ensure that delay is recognized "one day at a time." To accomplish this, you must have a schedule that has been defined at a degree of resolution that allows progress to be monitored and the project to be controlled.

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled.

"Overly optimistic scheduling doesn't result in shorter actual schedules, it results in longer ones."

Steve McConnell

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

27.2.1 Basic Principles

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

When you develop a schedule, compartmentalize the work, note task interdependencies, allocate effort and time to each task, and define responsibilities, outcomes, and milestones.

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort[4]). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved.

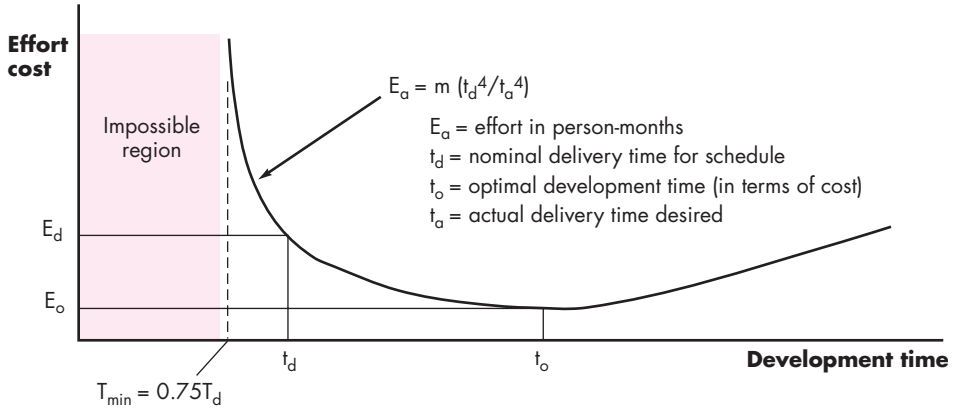
Each of these principles is applied as the project schedule evolves.

27.2.2 The Relationship Between People and Effort

In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved. (We can rarely afford the luxury of approaching a 10 person-year effort with one person working for 10 years!)

4 In reality, less than three person-days of effort are available because of unrelated meetings, sickness, vacation, and a variety of other reasons. For our purposes, however, we assume 100 percent availability.

FIGURE 27.1
The relationship between effort and delivery time



ADVICE

If you must add people to a late project, be sure that you've assigned them work that is highly compartmentalized.

There is a common myth that is still believed by many managers who are responsible for software development projects: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.

In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources).

The *Putnam-Norden-Rayleigh (PNR) Curve*[5] provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 27.1. The curve indicates a minimum value t_o that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of t_o (i.e., as we try to accelerate delivery), the curve rises nonlinearly.

KEY POINT

If delivery can be delayed, the PNR curve indicates that project costs can be reduced substantially.

As an example, we assume that a project team has estimated a level of effort E_d will be required to achieve a nominal delivery time t_d that is optimal in terms of

5 Original research can be found in [Nor70] and [Put78].

schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of t_d. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond $0.75t_d$. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option, $t_o = 2t_d$. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.

The software equation [Put92] introduced in Chapter 26 is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), *L,* is related to effort and development time by the equation:

$$L = P \times E^{1/3} t^{4/3}$$

where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for P range between 2000 and 12,000), and t is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort E:

$$E = \frac{L^3}{P^3 t^4} \tag{27.1}$$

where E is the effort expended (in person-years) over the entire life cycle for software development and maintenance and t is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

As the project deadline becomes tighter and tighter, you reach a point at which the work cannot be completed on schedule, regardless of the number of people doing the work. Face reality and define a new delivery date.

This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (27.1) yields:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ person-years}$$

This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

27.2.3 Effort Distribution

Each of the software project estimation techniques discussed in Chapter 26 leads to estimates of work units (e.g., person-months) required to complete software

development. A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule.* Forty percent of all effort is allocated to front-end analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.

? How should effort be distributed across the software process workflow?

This effort distribution should be used as a guideline only.[6] The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

27.3 Defining a Task Set for the Software Project

Regardless of the process model that is chosen, the work that a software team performs is achieved through a set of tasks that enable you to define, develop, and ultimately support computer software. No single task set is appropriate for all projects. The set of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product. Therefore, an effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

As I noted in Chapter 2, a task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project. The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.

In order to develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work. Although it is difficult

6 Today, the 40-20-40 rule is under attack. Some believe that more than 40 percent of overall effort should be expended during analysis and design. On the other hand, some proponents of agile development (Chapter 3) argue that less time should be expended "up front" and that a team should move quickly to construction.

to develop a comprehensive taxonomy of software project types, most software organizations encounter the following projects:

WebRef

An adaptable process model (APM) has been developed to assist in defining task sets for various software projects. A complete description of the APM can be found at **www.rspa.com/apm**.

1. *Concept development projects* that are initiated to explore some new business concept or application of some new technology.
2. *New application development* projects that are undertaken as a consequence of a specific customer request.
3. *Application enhancement* projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
4. *Application maintenance projects* that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
5. *Reengineering projects* that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Even within a single project type, many factors influence the task set to be chosen. These include [Pre05]: size of the project, number of potential users, mission criticality, application longevity, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, embedded and nonembedded characteristics, project staff, and reengineering factors. When taken in combination, these factors provide an indication of the *degree of rigor* with which the software process should be applied.

27.3.1 A Task Set Example

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following actions:

1.1 **Concept scoping** determines the overall scope of the project.

1.2 **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.

1.3 **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.

1.4 **Proof of concept** demonstrates the viability of a new technology in the software context.

1.5 **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

1.6 **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these actions should yield few surprises. In fact, the software engineering flow for concept development projects (and for all other types of projects as well) is little more than common sense.

27.3.2 Refinement of Software Engineering Actions

The software engineering actions described in the preceding section may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each action and decomposing it into a set of tasks (with related work products and milestones).

As an example of task decomposition, consider Action 1.1, Concept Scoping. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping action:

```
Task definition: Action 1.1 Concept Scoping
1.1.1   Identify need, benefits and potential customers;
1.1.2   Define desired output/control and input events that drive the application;
        Begin Task 1.1.2
        1.1.2.1 TR: Review written description of need[7]
        1.1.2.2 Derive a list of customer visible outputs/inputs
        1.1.2.3 TR: Review outputs/inputs with customer and revise as required; endtask
        Task 1.1.2
1.1.3   Define the functionality/behavior for each major function;
        Begin Task 1.1.3
        1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
        1.1.3.2 Derive a model of functions/behaviors;
        1.1.3.3 TR: Review functions/behaviors with customer and revise as required;
        endtask Task 1.1.3
1.1.4   Isolate those elements of the technology to be implemented in software;
1.1.5   Research availability of existing software;
1.1.6   Define technical feasibility;
1.1.7   Make quick estimate of size;
1.1.8   Create a scope definition;
endtask definition: Action 1.1
```

The tasks and subtasks noted in the process design language refinement form the basis for a detailed schedule for the concept scoping action.

7 TR indicates that a technical review (Chapter 15) is to be conducted.

27.4 Defining a Task Network

KEY POINT

The task network is a useful mechanism for depicting intertask dependencies and determining the critical path.

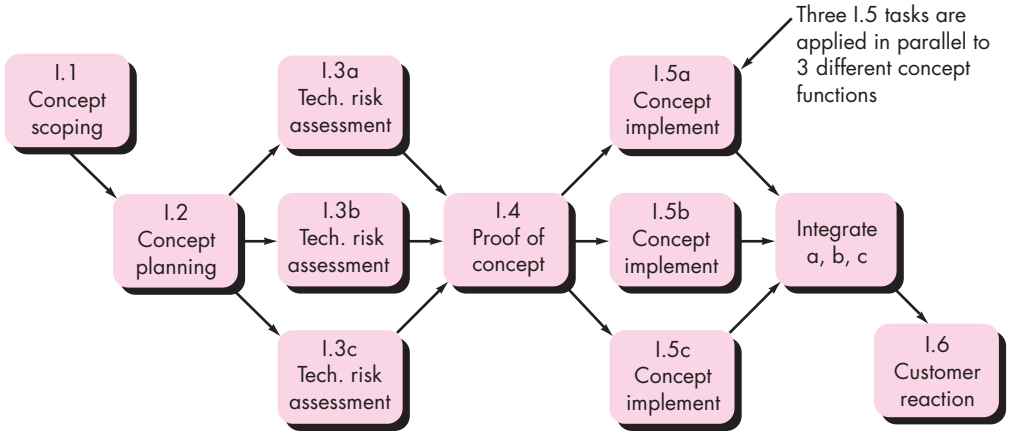
Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. Figure 27.2 shows a schematic task network for a concept development project.

The concurrent nature of software engineering actions leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, you should determine intertask dependencies to ensure continuous progress toward completion. In addition, you should be aware of those tasks that lie on the *critical path.* That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule. These issues are discussed in more detail later in this chapter.

It is important to note that the task network shown in Figure 27.2 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each action shown in the figure would be expanded. For example, Task 1.1 would be expanded to show all tasks detailed in the refinement of Actions 1.1 shown in Section 27.3.2.

FIGURE 27.2 A task network for concept development



27.5 SCHEDULING

Quote:

"*All* we have to decide is what to do with the time that is given to us."

Gandalf in *The Lord of the Rings: Fellowship of the Rings*

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

Program evaluation and review technique (PERT) and the *critical path method* (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected.

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish "most likely" time estimates for individual tasks by applying statistical models, and (3) calculate "boundary times" that define a time "window" for a particular task.

27.5.1 Time-Line Charts

When creating a software project schedule, you begin with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as

SOFTWARE TOOLS

Project Scheduling

Objective: The objective of project scheduling tools is to enable a project manager to define work tasks; establish their dependencies; assign human resources to tasks; and develop a variety of graphs, charts, and tables that aid in tracking and control of the software project.

Mechanics: In general, project scheduling tools require the specification of a work breakdown structure of tasks or the generation of a task network. Once the task breakdown (an outline) or network is defined, start and end dates, human resources, hard deadlines, and other data are attached to each. The tool then generates a variety of time-line charts and other tables that enable a manager to assess the task flow of a project. These data can be updated continually as the project is under way.

Representative Tools:[8]

AMS Realtime, developed by Advanced Management Systems (**www.amsusa.com**), provides scheduling capabilities for projects of all sizes and types.

Microsoft Project, developed by Microsoft (**www.microsoft.com**), is the most widely used PC-based project scheduling tool.

4C, developed by *4C Systems* (**www.4csys.com**), supports all aspects of project planning including scheduling.

A comprehensive list of project management software vendors and products can be found at **www.infogoal.com/pmc/pmcswr.htm**.

8 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.

a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.

KEY POINT

A time-line chart enables you to determine what tasks will be conducted at a given point in time.

As a consequence of this input, a *time-line chart,* also called a *Gantt chart,* is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Figure 27.3 illustrates the format of a time-line chart. It depicts a part of a software project schedule that emphasizes the concept scoping task for a word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information (Figure 27.4). Used in conjunction with the time-line chart, project tables enable you to track progress.

FIGURE 27.3 An example time-line chart

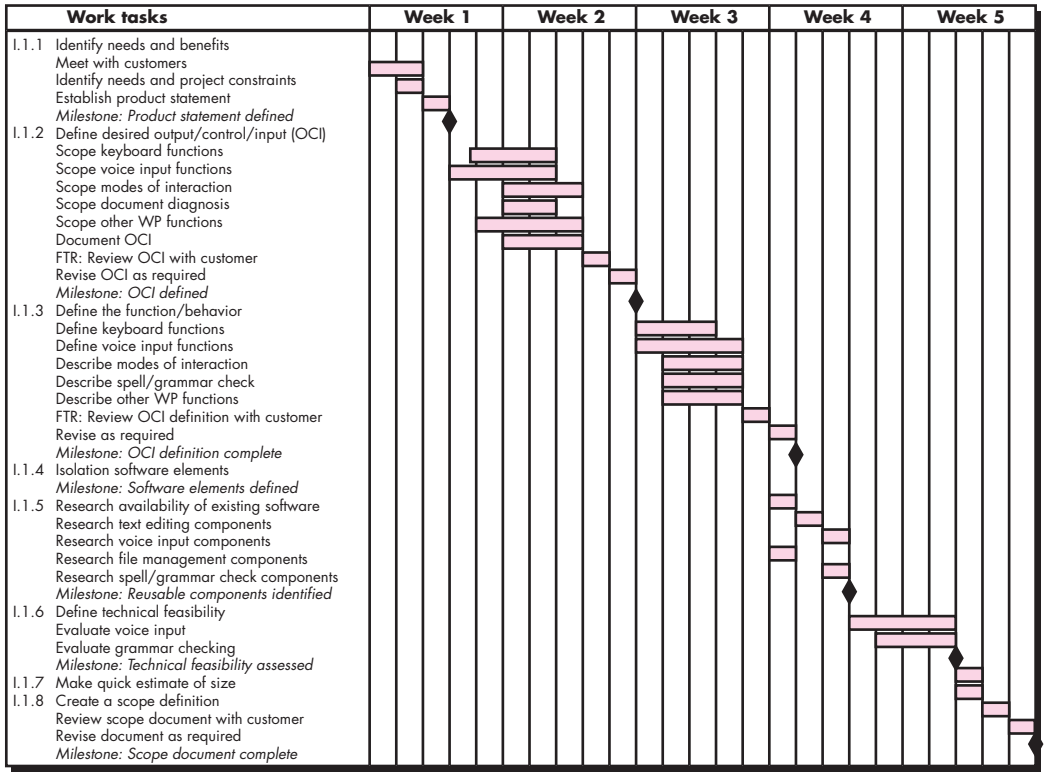


FIGURE 27.4 An example project table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

27.5.2 Tracking the Schedule

Quote:

"The basic rule of software status reporting can be summarized in a single phrase: 'No surprises'."

Capers Jones

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems
- Evaluating the results of all reviews conducted throughout the software engineering process
- Determining whether formal project milestones (the diamonds shown in Figure 27.3) have been accomplished by the scheduled date
- Comparing the actual start date to the planned start date for each project task listed in the resource table (Figure 27.4)
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon
- Using earned value analysis (Section 27.6) to assess progress quantitatively

ADVICE

The best indication of progress is the completion and successful review of a defined software work product.

In reality, all of these tracking techniques are used by experienced project managers.

Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems

occur, you must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [Jal04]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm (Chapter 2) is chosen, and a schedule is derived for each incremental delivery.

KEY POINT

When the defined completion date of a time-boxed task is reached, work ceases for that task and the next task begins.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative: "If the work isn't finished, how can we proceed?" The answer lies in the way work is accomplished. By the time the time-box boundary is encountered, it is likely that 90 percent of the task has been completed.[9] The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

27.5.3 Tracking Progress for an OO Project

Although an iterative model is the best framework for an OO project, task parallelism makes project tracking difficult. You may have difficulty establishing meaningful milestones for an OO project because a number of different things are happening at once. In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed

- All classes and the class hierarchy have been defined and reviewed.
- Class attributes and operations associated with a class have been defined and reviewed.
- Class relationships (Chapter 6) have been established and reviewed.
- A behavioral model (Chapter 7) has been created and reviewed.
- Reusable classes have been noted.

Technical milestone: OO design completed

- The set of subsystems has been defined and reviewed.
- Classes are allocated to subsystems and reviewed.
- Task allocation has been established and reviewed.

9 A cynic might recall the saying: "The first 90 percent of the system takes 90 percent of the time; the remaining 10 percent of the system takes 90 percent of the time."

- Responsibilities and collaborations have been identified.
- Attributes and operations have been designed and reviewed.
- The communication model has been created and reviewed.

Technical milestone: OO programming completed

- Each new class has been implemented in code from the design model.
- Extracted classes (from a reuse library) have been implemented.
- Prototype or increment has been built.

Technical milestone: OO testing

Debugging and testing occur in concert with one another. The status of debugging is often assessed by considering the type and number of "open" errors (bugs).

- The correctness and completeness of OO analysis and design models has been reviewed.
- A class-responsibility-collaboration network (Chapter 6) has been developed and reviewed.
- Test cases are designed, and class-level tests (Chapter 19) have been conducted for each class.
- Test cases are designed, and cluster testing (Chapter 19) is completed and the classes are integrated.
- System-level tests have been completed.

Recalling that the OO process model is iterative, each of these milestones may be revisited as different increments are delivered to the customer.

27.5.4 Scheduling for WebApp Projects

WebApp project scheduling distributes estimated effort across the planned time line (duration) for building each WebApp increment. This is accomplished by allocating the effort to specific tasks. It is important to note, however, that the overall WebApp schedule evolves over time. During the first iteration, a macroscopic schedule is developed. This type of schedule identifies all WebApp increments and projects the dates on which each will be deployed. As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule. Here, specific development tasks (required to accomplish an activity) are identified and scheduled.

As an example of macroscopic scheduling, consider the **SafeHomeAssured.com** WebApp. Recalling earlier discussions of **SafeHomeAssured.com**, seven increments can be identified for the Web-based component of the project:

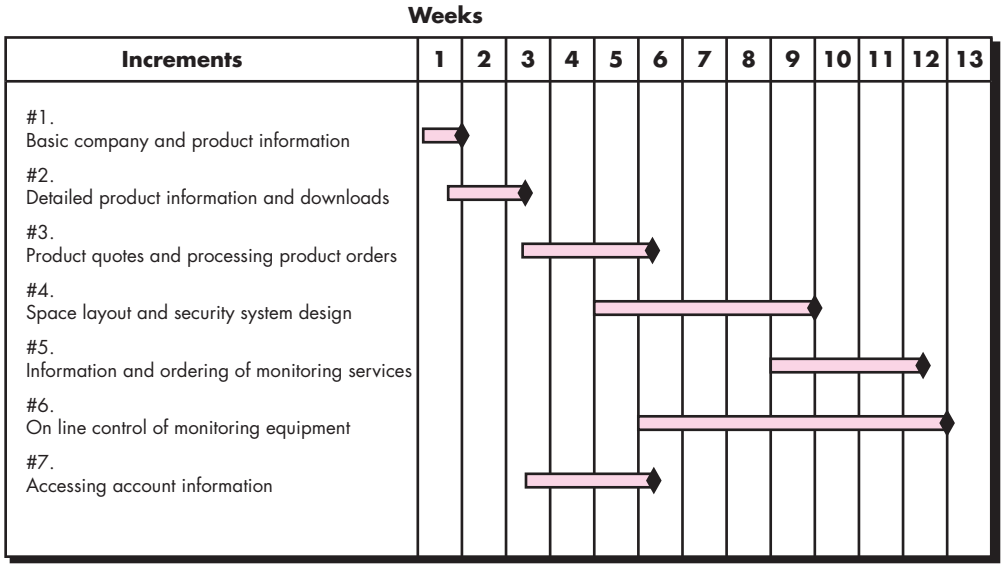
Increment 1: Basic company and product information

Increment 2: Detailed product information and downloads

Increment 3: Product quotes and processing product orders

Increment 4: Space layout and security system design

FIGURE 27.5 Time line for macroscopic project schedule



Increment 5: Information and ordering of monitoring services

Increment 6: Online control of monitoring equipment

Increment 7: Accessing account information

The team consults and negotiates with stakeholders and develops a *preliminary* deployment schedule for all seven increments. A time-line chart for this schedule is illustrated in Figure 27.5.

It is important to note that the deployment dates (represented by diamonds on the time-line chart) are preliminary and may change as more detailed scheduling of the increments occurs. However, this macroscopic schedule provides management with an indication of when content and functionality will be available and when the entire project will be completed. As a preliminary estimate, the team will work to deploy all increments with a 12-week time line. It's also worth noting that some of the increments will be developed in parallel (e.g., increments 3, 4, 6 and 7). This assumes that the team will have sufficient people to do this parallel work.

Once the macroscopic schedule has been developed, the team is ready to schedule work tasks for a specific increment. To accomplish this, you can use a generic process framework that is applicable for all WebApp increments. A *task list* is created by using the generic tasks derived as part of the framework as a starting point and then adapting these by considering the content and functions to be derived for a specific WebApp increment.

Each framework action (and its related tasks) can be adapted in one of four ways: (1) a task is applied as is, (2) a task is eliminated because it is not necessary for the

increment, (3) a new (custom) task is added, and (4) a task is refined (elaborated) into a number of named subtasks that each becomes part of the schedule.

To illustrate, consider a generic *design modeling* action for WebApps that can be accomplished by applying some or all of the following tasks:

- Design the aesthetic for the WebApp.
- Design the interface.
- Design the navigation scheme.
- Design the WebApp architecture.
- Design the content and the structure that supports it.
- Design functional components.
- Design appropriate security and privacy mechanisms.
- Review the design.

As an example, consider the generic task *Design the Interface* as it is applied to the fourth increment of **SafeHomeAssured.com**. Recall that the fourth increment implements the content and function for describing the living or business space to be secured by the *SafeHome* security system. Referring to Figure 27.5, the fourth increment commences at the beginning of the fifth week and terminates at the end of the ninth week.

There is little question that the *Design the Interface* task must be conducted. The team recognizes that the interface design is pivotal to the success of the increment and decides to refine (elaborate) the task. The following subtasks are derived for the *Design the Interface* task for the fourth increment:

- Develop a sketch of the page layout for the space design page.
- Review layout with stakeholders.
- Design space layout navigation mechanisms.
- Design "drawing board" layout.[10]
- Develop procedural details for the graphical wall layout function.
- Develop procedural details for the wall length computation and display function.
- Develop procedural details for the graphical window layout function.
- Develop procedural details for the graphical door layout function.
- Design mechanisms for selecting security system components (sensors, cameras, microphones, etc.).

10 At this stage, the team envisions creating the space by literally drawing the walls, windows, and doors using graphical functions. Wall lines will "snap" onto grip points. Dimensions of the wall will be displayed automatically. Windows and doors will be positioned graphically. The end user can also select specific sensors, cameras, etc., and position them once the space has been defined.

- Develop procedural details for the graphical layout of security system components.
- Conduct pair walkthroughs as required.

These tasks become part of the increment schedule for the fourth WebApp increment and are allocated over the increment development schedule. They can be input to scheduling software and used for tracking and control.

SAFEHOME

Tracking the Schedule

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug (glancing at a PowerPoint slide): The schedule for the first *SafeHome* increment seems reasonable, but we're going to have trouble tracking progress.

Vinod (a concerned look on his face): Why? We have tasks scheduled on a daily basis, plenty of work products, and we've been sure that we're not overallocating resources.

Doug: All good, but how do we know when the requirements model for the first increment is complete?

Jamie: Things are iterative, so that's difficult.

Doug: I understand that, but . . . well, for instance, take "analysis classes defined." You indicated that as a milestone.

Vinod: We have.

Doug: Who makes that determination?

Jamie (aggravated): They're done when they're done.

Doug: That's not good enough, Jamie. We have to schedule TRs [technical reviews, Chapter 15], and you haven't done that. The successful completion of a review on the analysis model, for instance, is a reasonable milestone. Understand?

Jamie (frowning): Okay, back to the drawing board.

Doug: It shouldn't take more than an hour to make the corrections . . . everyone else can get started now.

27.6 EARNED VALUE ANALYSIS

Earned value provides a quantitative indication of progress.

In Section 27.5, I discussed a number of qualitative approaches to project tracking. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA). Humphrey [Hum95] discusses earned value in the following manner:

> The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

Stated even more simply, earned value is a measure of progress. It enables you to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [Fle98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project." To determine the earned value, the following steps are performed:

? How do I compute earned value and use it to assess progress?

1. The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, $BCWS_i$ is the effort planned for work task *i*. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the $BCWS_i$ values for all work tasks that should have been completed by that point in time on the project schedule.
2. The BCWS values for all work tasks are summed to derive the *budget at completion* (BAC). Hence,

   $BAC = \Sigma (BCWS_k)$ for all tasks k
3. Next, the value for *budgeted cost of work performed* (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Wilkens [Wil99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed." Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

$$\text{Schedule performance index, SPI} = \frac{\text{BCWP}}{\text{BCWS}}$$

$$\text{Schedule variance, SV} = \text{BCWP} - \text{BCWS}$$

WebRef

A wide array of earned value analysis resources can be found at **www.acq.osd.mil/pm/**.

SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

$$\text{Percent scheduled for completion} = \frac{\text{BCWS}}{\text{BAC}}$$

provides an indication of the percentage of work that should have been completed by time *t*.

$$\text{Percent complete} = \frac{\text{BCWP}}{\text{BAC}}$$

provides a quantitative indication of the percent of completeness of the project at a given point in time *t*.

It is also possible to compute the *actual cost of work performed* (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have

been completed by a point in time on the project schedule. It is then possible to compute

$$\text{Cost performance index, CPI} = \frac{\text{BCWP}}{\text{ACWP}}$$

$$\text{Cost variance, CV} = \text{BCWP} - \text{ACWP}$$

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

27.7 Summary

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical path, a time-line chart, and a variety of project information. Using the schedule as a guide, you can track and control each step in the software process.

Problems and Points to Ponder

27.1. "Unreasonable" deadlines are a fact of life in the software business. How should you proceed if you're faced with one?

27.2. What is the difference between a macroscopic schedule and a detailed schedule? Is it possible to manage a project if only a macroscopic schedule is developed? Why?

27.3. Is there ever a case where a software project milestone is not tied to a review? If so, provide one or more examples.

27.4. "Communication overhead" can occur when multiple people work on a software project. The time spent communicating with others reduces individual productively (LOC/month), and the result can be less productivity for the team. Illustrate (quantitatively) how engineers who are well versed in good software engineering practices and use technical reviews can increase the production rate of a team (when compared to the sum of individual production rates). Hint: You can assume that reviews reduce rework and that rework can account for 20 to 40 percent of a person's time.

27.5. Although adding people to a late software project can make it later, there are circumstances in which this is not true. Describe them.

27.6. The relationship between people and time is highly nonlinear. Using Putnam's software equation (described in Section 27.2.2), develop a table that relates number of people to project

duration for a software project requiring 50,000 LOC and 15 person-years of effort (the productivity parameter is 5000 and $B = 0.37$). Assume that the software must be delivered in 24 months plus or minus 12 months.

27.7. Assume that you have been contracted by a university to develop an online course registration system (OLCRS). First, act as the customer (if you're a student, that should be easy!) and specify the characteristics of a good system. (Alternatively, your instructor will provide you with a set of preliminary requirements for the system.) Using the estimation methods discussed in Chapter 26, develop an effort and duration estimate for OLCRS. Suggest how you would:

a. Define parallel work activities during the OLCRS project.

b. Distribute effort throughout the project.

c. Establish milestones for the project.

27.8. Select an appropriate task set for the OLCRS project.

27.9. Define a task network for OLCRS described in Problem 27.7, or alternatively, for another software project that interests you. Be sure to show tasks and milestones and to attach effort and duration estimates to each task. If possible, use an automated scheduling tool to perform this work.

27.10. If an automated scheduling tool is available, determine the critical path for the network defined in Problem 27.9.

27.11. Using a scheduling tool (if available) or paper and pencil (if necessary), develop a timeline chart for the OLCRS project.

27.12. Assume you are a software project manager and that you've been asked to compute earned value statistics for a small software project. The project has 56 planned work tasks that are estimated to require 582 person-days to complete. At the time that you've been asked to do the earned value analysis, 12 tasks have been completed. However the project schedule indicates that 15 tasks should have been completed. The following scheduling data (in person-days) are available:

Task	Planned Effort	Actual Effort
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Compute the SPI, schedule variance, percent scheduled for completion, percent complete, CPI, and cost variance for the project.

Further Readings and Information Sources

Virtually every book written on software project management contains a discussion of scheduling. Wysoki (*Effective Project Management,* Wiley, 2006), Lewis (*Project Planning Scheduling and Control,* 4th ed., McGraw-Hill, 2006), Luckey and Phillips (*Software Project Management for Dummies,* For Dummies, 2006), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 9th ed., Wiley, 2005), Hughes (*Software Project Management,* McGraw-Hill, 2005), The Project Management Institute (*PMBOK Guide,* 3d ed., PMI, 2004), Lewin (*Better Software Project Management,* Wiley, 2001), and Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3d ed., Wiley, 2000) contain worthwhile discussions of the subject. Although application specific, Harris (*Planning and Scheduling Using Microsoft Office Project 2007,* Eastwood Harris Pty Ltd., 2007) provides a useful discussion of how scheduling tools can be used to successfully track and control a software project.

Fleming and Koppelman (*Earned Value Project Management,* 3d ed., Project Management Institute Publications, 2006), Budd (*A Practical Guide to Earned Value Project Management,* Management Concepts, 2005), and Webb and Wake (*Using Earned Value: A Project Manager's Guide,* Ashgate Publishing, 2003) discuss the use of earned value techniques for project planning, tracking, and control in considerable detail.

A wide variety of information sources on software project scheduling is available on the Internet. An up-to-date list of World Wide Web references relevant to software project scheduling can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.