CHAPTER

# 1 The Nature of Software

## Key Concepts



As he finished showing me the latest build of one of the world's most popular first-person shooter video games, the young developer laughed.

"You're not a gamer, are you?" he asked.

I smiled. "How'd you guess?"

The young man was dressed in shorts and a tee shirt. His leg bounced up and down like a piston, burning the nervous energy that seemed to be commonplace among his co-workers.

"Because if you were," he said, "you'd be a lot more excited. You've gotten a peek at our next generation product and that's something that our customers would kill for . . . no pun intended."

We sat in a development area at one of the most successful game developers on the planet. Over the years, earlier generations of the game he demoed sold over 50 million copies and generated billions of dollars in revenue.

"So, when will this version be on the market?" I asked.

He shrugged. "In about five months, and we've still got a lot of work to do."

He had responsibility for game play and artificial intelligence functionality in an application that encompassed more than three million lines of code.

"Do you guys use any software engineering techniques?" I asked, half-expecting that he'd laugh and shake his head.

## 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.

**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.

**What are the steps?** Customers and other stakeholders express the need for computer software, engineers build the software product, and end users apply the software to solve a specific problem or to address a specific need.

**What is the work product?** A computer program that runs in one or more specific environments and services the needs of one or more end users.

**How do I ensure that I've done it right?** If you're a software engineer, apply the ideas contained in the remainder of this book. If you're an end user, be sure you understand your need and your environment and then select an application that best meets them both.

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, a hand-held tablet, on the desktop, or within 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. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

**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 and protect 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 led 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*?**

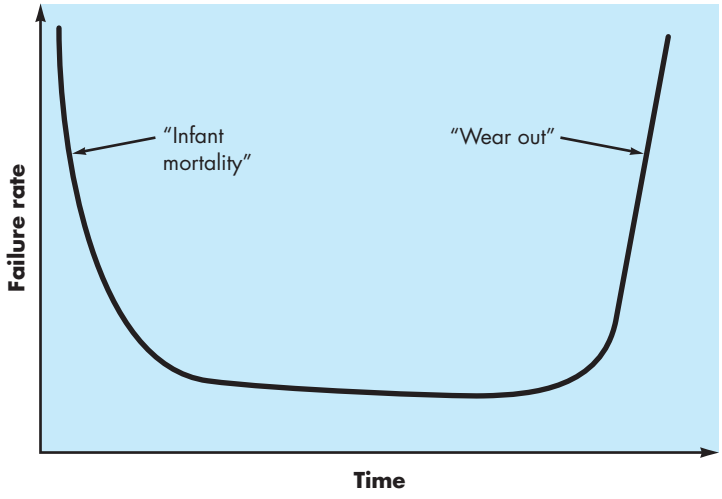
> Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation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

---

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**




ADVICE

*If you want to reduce software deterioration, you'll have to do better software design (Chapters 12 to 18).*

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 one fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*

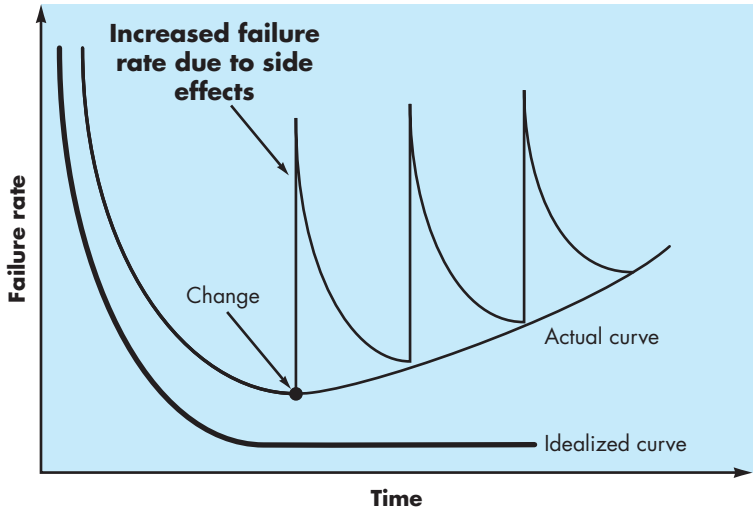
Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and 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!*

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

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.

**FIGURE 1.2**
Failure curves for software




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

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.

### 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,[3] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

3 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.

**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.

**Engineering/scientific software—**a broad array of "number-crunching programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

**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.

**Web/Mobile applications—**this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

**Quote:**

"What a computer is to me is the most remarkable tool that we have ever come up with. It's the equivalent of a bicycle for our minds."

Steve Jobs

**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.

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 on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.

### 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*.[4] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results 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 evolving computing environment.


*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 36) 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]

4 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.

Referring to the figure, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure. In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data structures. In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device.

The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services. The *front-end* includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed. The *back-end* includes servers and related computing resources, data storage systems (e.g., databases), server-resident applications, and administrative servers that use middleware to coordinate and monitor traffic by establishing a set of protocols for access to the cloud and its resident resources. [Str08]

The cloud architecture can be segmented to provide access at a variety of different levels from full public access to private cloud architectures accessible only to those with authorization.

### 1.2.4 Product Line Software

The Software Engineering Institute defines a *software product line* as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [SEI13] The concept of a line of software products that are related in some way is not new. But the idea that a line of software products, all developed using the same underlying application and data architectures, and all implemented using a set of reusable software components that can be reused across the product line provides significant engineering leverage.

A software product line shares a set of assets that include requirements (Chapter 8), architecture (Chapter 13), design patterns (Chapter 16), reusable components (Chapter 14), test cases (Chapters 22 and 23), and other software engineering work products. In essence, a software product line results in the development of many products that are engineered by capitalizing on the commonality among all the products within the product line.

## 1.3 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.

The nature of software is changing. 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. Mobile applications present new challenges as apps migrate to a wide array of platforms. Cloud computing will transform the way in which software is delivered and the environment in which it exists. Product line software offers potential efficiencies in the manner in which software is built.

## 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.

## Further Readings and Information Sources[6]

Literally thousands of books are 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, 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

6 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. We have created a comprehensive website to support *Software Engineering: A Practitioner's Approach* at **www.mhhe.com/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.

CHAPTER

# 2 SOFTWARE ENGINEERING

## KEY CONCEPTS



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

- 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[1] has grown dramatically. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*
- 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. *It follows that design becomes a pivotal activity.*

## QUICK LOOK

**What is it?** 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 apply the software engineering process.

**Why is it important?** Software engineering is important because it enables us to build complex systems in a timely manner and with high quality. It imposes discipline to work that can become quite chaotic, but it also allows the people who build computer software to adapt their approach in a manner that best suits their needs.

**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.

1 We will call these people "stakeholders" later in this book.

KEY POINT
Understand the problem before you build a solution.

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

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

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*.

## 2.1 Defining the Discipline

**? How do we define software engineering?**

The IEEE [IEE93a] has developed the following definition for 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.

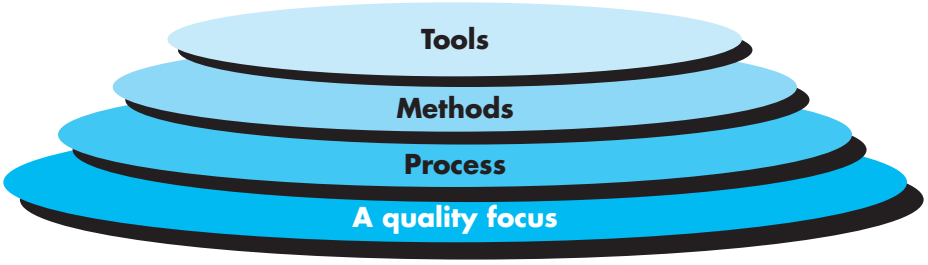
Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies[2] foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

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

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 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

2 Quality management and related approaches are discussed throughout Part 3 of this book.

**FIGURE 2.1** Software engineering layers




applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

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

**WebRef**

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

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

## 2.2 THE SOFTWARE PROCESS

**? What are the elements of a software process?**

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 model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

**Quote:**

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

**Ivar Jacobson, Grady Booch, and James Rumbaugh**

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.

### 2.2.1 The Process Framework

**What are the five generic process framework activities?**

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:

**Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders).[3] The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

**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**

**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.** What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

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

These five generic framework activities can be used during the development of small, simple programs, the creation of Web applications, and for the engineering

3 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."

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.

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 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.

### 2.2.2 Umbrella Activities

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:

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**—assess 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.

Software process adaptation is essential for project success.

**Work product preparation and production**—encompass 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.

### 2.2.3 Process Adaptation

Previously in this section, we 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

- 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.
- 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, we examine software process in considerable detail.

**Quote:**

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

**Madame Benoit**

## 2.3 Software Engineering Practice

In Section 2.2, we 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.[4]

**WebRef**

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

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

4 You should revisit relevant sections within this chapter as we discuss specific software engineering methods and umbrella activities later in this book.

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


*The most important element of problem understanding is* listening.

**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?

**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?


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

George Polya

**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?* Has 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.

### 2.3.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we'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.


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

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:[5]

#### 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.


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

**Alexander Pope (1688–1744)**

#### 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 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

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

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.[6] This could very possibly lead to the reuse of an entire system.

### The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[7] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code

6 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.

7 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.

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 conventionall 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.

## 2.4 Software Development Myths

Software development 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.

**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).

**WebRef**

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

**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?*

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.

## 2.6 Summary

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

**2.1.** Figure 2.1 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.

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

**2.3.** 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.

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

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

**2.6.** Add two additional myths to the list presented in Section 2.4. Also state the reality that accompanies the myth.

## Further Readings and Information Sources

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 IT Journal* often contain articles on software engineering topics. The discipline is "summarized" every year in the *Proceeding of the International Conference*

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

# 3 Software Process Structure

## 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, we 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.

## 3.1 A Generic Process Model

In Chapter 2, 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 resides within a framework or model that defines their relationship with the process and with one another.


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

The software process is represented schematically in Figure 3.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As we discussed in Chapter 2, 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.


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 3.2.

A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 3.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 3.2b). 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 3.2c). A *parallel process flow* (Figure 3.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

2. ***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]

**Quote:**

"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

**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.

3 These phase patterns are discussed in Chapter 4.

**WebRef**

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

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.

**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 4.1.3.

**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.

## 3.5 PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapter 19). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself can be assessed to

CHAPTER

# 4 PROCESS MODELS

## KEY CONCEPTS



Process models were originally proposed to bring order to the chaos of software development. History has indicated that these 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 products that are produced remain on "the edge of chaos."

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

## QUICK LOOK

**What is it?** A process model provides a specific roadmap for software engineering work. It defines the flow of all activities, actions and tasks, the degree of iteration, the work products, and the organization of the work that must be done.

**Who does it?** Software engineers and their managers adapt a process model 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 process 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?** The process model provides you with the "steps" you'll need to perform disciplined software engineering work.

**What is the work product?** From the point of view of a software engineer, the work product is a customized description 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.

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.

KEY POINT
The purpose of process models is to try to reduce the chaos present in developing new software products.

The philosophical implications of this argument are significant for software engineering. Each process model described in this chapter tries to strike a balance between the need to impart order in a chaotic world and the need to be adaptable when things change constantly.

## 4.1 Prescriptive Process Models

WebRef
An award-winning "process simulation game" that includes most important prescriptive process models can be found at: **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html**.

A *prescriptive process model*[1] strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we examine the prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapters 2 and 3, 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.

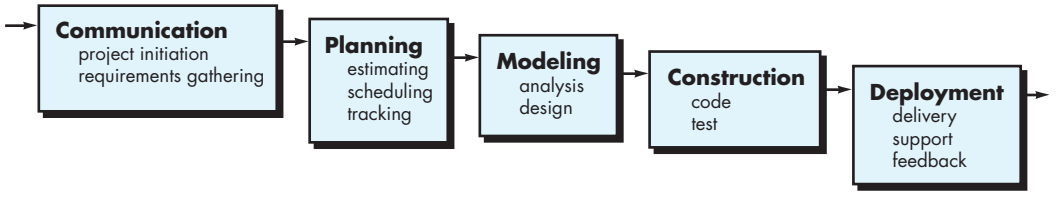
### 4.1.1 The Waterfall Model

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

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.

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

**FIGURE 4.1** The waterfall model




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

The *waterfall model*, sometimes called the *classic life cycle*, suggests a systematic, sequential approach[2] to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 4.1).

A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 4.2, the V-model [Buc99] depicts the relationship of quality 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 moves down the left side.[3] 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 four 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:

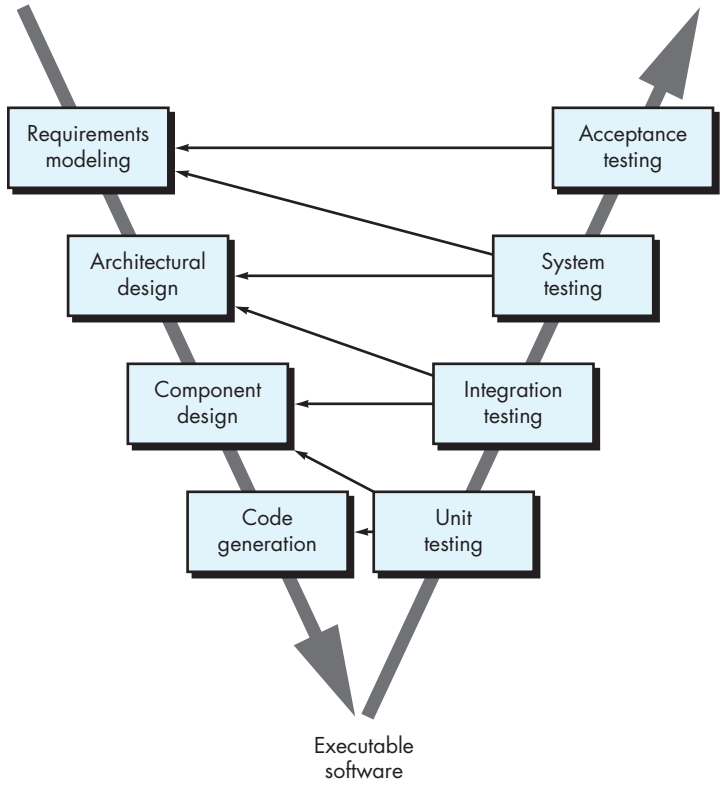
**? Why does the waterfall model sometimes fail?**

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.
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.

2 Although the original waterfall model proposed by Winston Royce [Roy70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

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

**FIGURE 4.2**
The V-model




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**

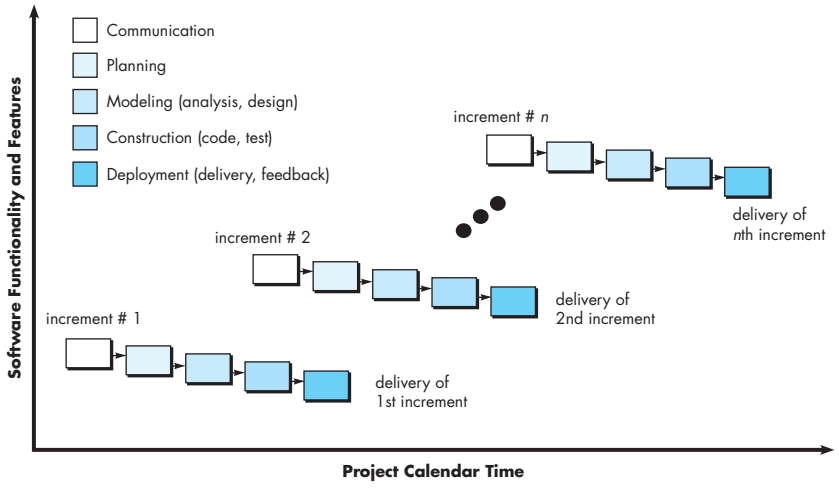
In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

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

### 4.1.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely

**FIGURE 4.3**
**The incremental model**




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 the elements' linear and parallel process flows discussed in Chapter 3. Referring to Figure 4.3, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93].

**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.

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 discussed in the next subsection.

**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.*

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 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.

### 4.1.3 Evolutionary Process Models


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

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

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, we present two common evolutionary process models.


"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.

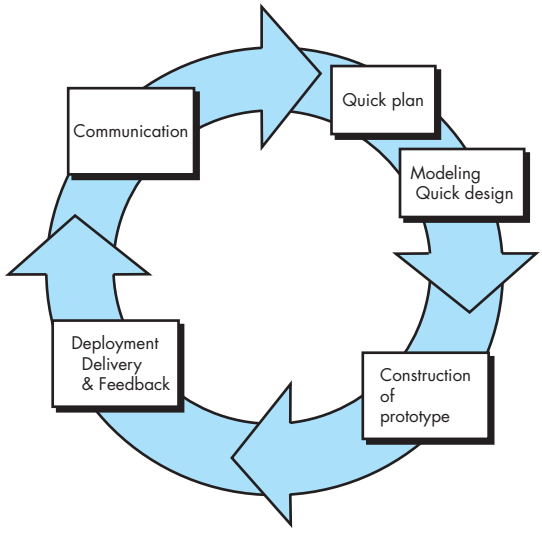
Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders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


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

The prototyping paradigm (Figure 4.4) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

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

**FIGURE 4.4**
The prototyping paradigm




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

## SafeHome

### *Selecting a Process Model, Part 1*

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

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

**The conversation:**

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

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

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

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

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

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

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

**Doug:** Give it a chance before you go negative on me. Here's what I mean. (Doug proceeds to describe the process framework described in Chapter 3 and the prescriptive process models presented to this point.)

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

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

**Doug:** I agree.

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

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

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

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.

**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:

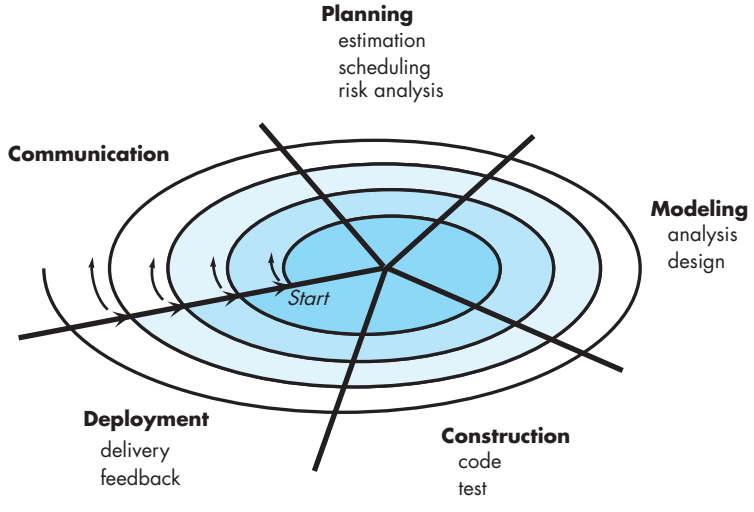
> The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation 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.

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

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

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.[4] Each of the framework activities represent one segment of the spiral path illustrated in Figure 4.5. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 35) 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.

**FIGURE 4.5**
**A typical spiral model**




4 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98].

**WebRef**

Useful information about the spiral model can be obtained at: **www.sei.cmu.edu/publications/documents/00.reports/00sr008.html**.

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.

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[5] until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).


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

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more 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.


"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.

### 4.1.4 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

5 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.

## SafeHome

### *Selecting a Process Model, Part 2*

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

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

**The conversation:** (Doug describes evolutionary process options.)

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

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

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

**Doug:** That's old-school thinking, Lee. Like the guys said, we've got to keep it real. I submit that it's better to tweak the plan as we learn more and as changes are requested. It's way more realistic. What's the point of a plan if it doesn't reflect reality?

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

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


Project plans must be viewed as living documents; progress must be assessed often and revised to take changes into account.


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


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

V. Daniel Hunt

defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.[6]

Figure 4.6 provides an example of the concurrent modeling approach. An activity—**modeling**—may be in any one of the states[7] 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.

For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **none** state while initial communication was completed) now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

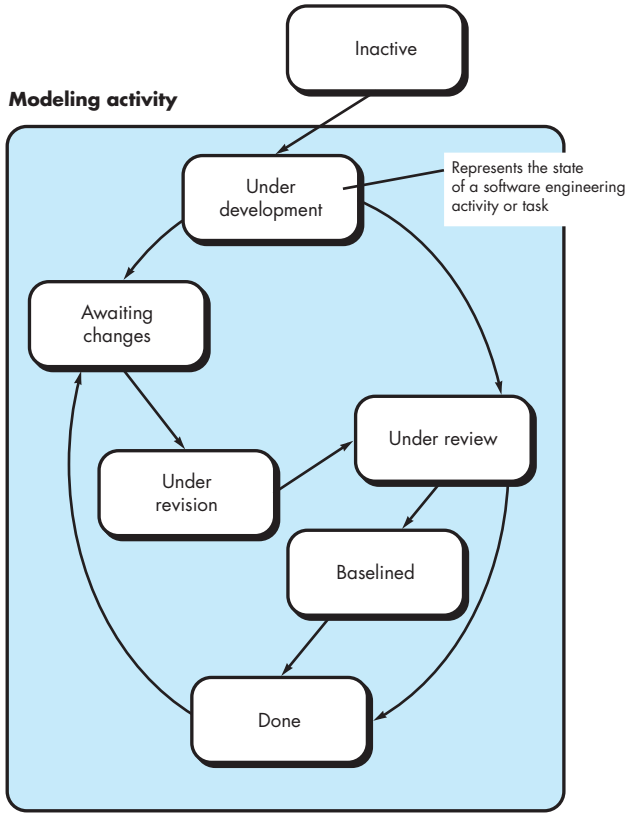
Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements

6 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.

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

**FIGURE 4.6**

One element of the concurrent process model




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.

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 associated with each activity.

### 4.1.5 A Final Word on Evolutionary Processes

We have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction. In many cases, time-to-market is the most important management

requirement. If a market window is missed, the software project itself may be meaningless.[8]

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 . . .
>
> Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected . . .
>
> Third, [evolutionary] software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary.

**? What are the potential weaknesses of evolutionary process models?**

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[9] 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).

## 4.2 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.[10]

---

8 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).

9 In this context software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Part 2 of this book.

10 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.

### 4.2.1 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 comprises 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[11] of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

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 including a reduction in development cycle time and a reduction in project cost if component reuse becomes part of your organization's culture. Component-based development is discussed in more detail in Chapter 14.

### 4.2.2 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.

11 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.

When formal methods (Appendix 3) 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.

### 4.2.3 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) or *aspect-oriented component engineering* (AOCE) [Gru02], 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].

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 [Ras11], [Saf08], [Cla05], [Fil05], [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:**[12]

*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.

*ALM Studio,* developed by Kovair Corporation (**http://www.kovair.com/**) encompasses a suite of tools for process definition, requirements management, issue resolution, project planning, and tracking.

*ProVision BPMx,* developed by OpenText (**http://bps.opentext.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.computer.org/portal/web/swebok/html/ch10**.

## 4.3 THE UNIFIED PROCESS

In their seminal book on the *Unified Process (UP),* 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

12 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.

> 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 (Chapter 5). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case).[13] 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.

### 4.3.1 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.

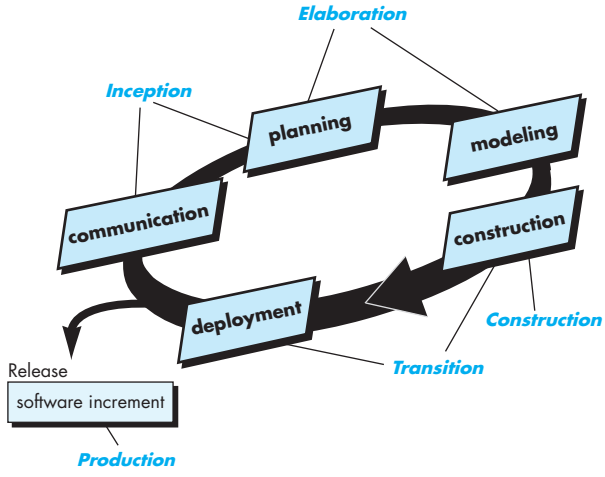
### 4.3.2 Phases of the Unified Process[14]

In Chapter 3, we discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process

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

14 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 4.7**
The Unified Process




is no exception. Figure 4.7 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.

**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 8) 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 functions 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 4.7). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [Arl02] that represents a "first cut" executable system.[15] The architectural baseline demonstrates the viability of the

15 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.

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.

**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, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests[16] 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 in Chapter 3). 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.

16 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 22 through 26).

## 4.4 Personal and Team Process Models

**Quote:**

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

**Dexter Yager**

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, 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 ([Hum05] 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.[17]

### 4.4.1 Personal Software Process

**WebRef**

A wide array of resources for PSP can be found at **http://www.sei.cmu.edu/tsp/tools/academic/**.

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 [Hum05] 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:

**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.

17 It's worth noting the proponents of agile software development (Chapter 5) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

**High-level design review.** Formal verification methods (Appendix 3) are applied to uncover errors in the design. Metrics are maintained for 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 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.

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 for you 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.

### 4.4.2 Team Software Process

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[18] level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.


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

A self-directed team has a consistent understanding of its overall goals and objectives; 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.

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

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 recognizes that the best software teams are self-directed.[19] 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.

## 4.5 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.

18 The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 37.

19 In Chapter 5 we discuss the importance of "self-organizing" teams as a key element in agile software development.

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Chapter 3. 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.

**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:**[20]

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

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

*ALM Studio Suite*—a collection of tools with a heavy emphasis on the management of communication and modeling activities (**http://www.kovair.com/**)

## 4.6 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

20 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.

## 4.7 Summary

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

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

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

**4.3.** What process adaptations are required if the prototype will evolve into a delivery system or product?

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

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

CHAPTER

# 5 AGILE DEVELOPMENT

**KEY CONCEPTS**



In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01] (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.

KEY POINT

Agile development does not mean no documents are created, it means only creating documents that will be referred to later in the development process.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a mobile 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 through 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 4 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*.

## 5.1 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.*

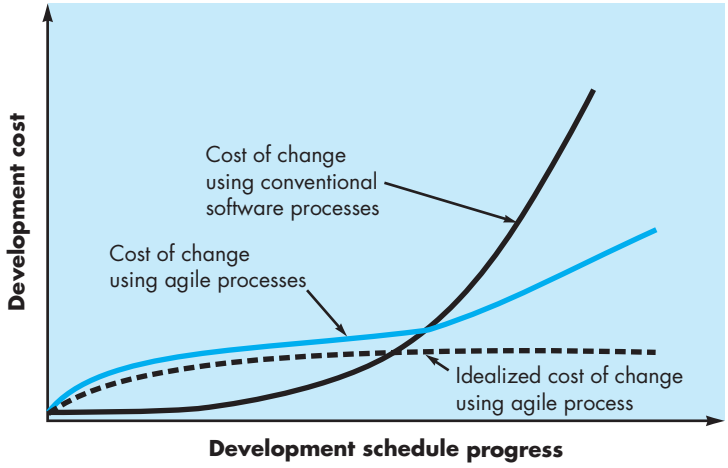
But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and deemphasizes 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.

## 5.2 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 5.1, 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

**FIGURE 5.1**
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.**

specification can be edited. The costs of doing this work are minimal, and the time required will 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.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 5.1, 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.

**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.

## 5.3 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 **http://www.agilemodeling.com/.**

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 we 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.

### 5.3.1 Agility Principles


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.


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

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
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.

### 5.3.2 The Politics of Agile Development


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

There has been 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."

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 5.4), 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.

## 5.4 Extreme Programming

**WebRef**

An award-winnng "process simulation game" that includes an XP process module can be found at **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html.**

In order to illustrate an agile process in a bit more detail, we'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]. A variant of XP, called *Industrial XP* (IXP), refines XP and targets the agile process specifically for use within large organizations [Ker05].

### 5.4.1 The XP Process

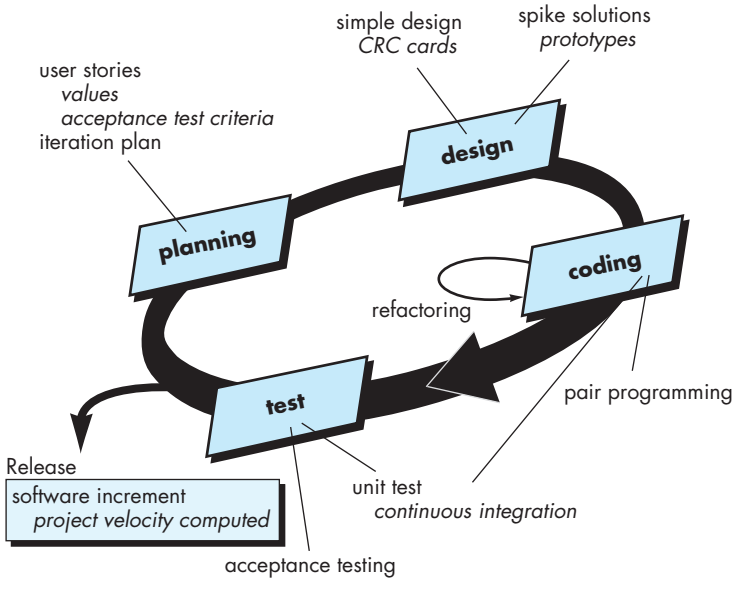
**What is an XP "story"?**

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 5.2 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

**Figure 5.2**

**The Extreme Programming process**



**WebRef**

A worthwhile XP "planning game" can be found at: **http://csis.pace.edu/~bergin/xp/planninggame.html.**

of the XP team to understand the business context for the software and to get a broad 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 8) 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.[2] 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.

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.


Project velocity is a subtle measure of team productivity.

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 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.


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

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.[3]

XP encourages the use of CRC cards (Chapter 10) as an effective mechanism for thinking about the software in an object-oriented context. CRC

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

3 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.

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

(class-responsibility-collaborator) cards identify and organize the object-oriented classes[4] 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 10. 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.

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

XP encourages *refactoring*—a construction technique that is also a design technique. 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 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.

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

**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).[5] Once the unit test[6] has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added

4 Object-oriented classes are discussed in Appendix 2, in Chapter10, and throughout Part 2 of this book.

5 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.

6 Unit testing, discussed in detail in Chapter 22, 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.

(KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

**What is pair programming?**

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.[7]


*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 22) that helps to uncover errors early.

**How are unit tests used in XP?**

**Testing.** 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 22) whenever code is modified (which is often, given the XP refactoring philosophy).

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 are derived from user stories.

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.

### 5.4.2 Industrial XP

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

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

7 Pair programming has become so widespread throughout the software community that *The Wall Street Journal* [Wal12] ran a front-page story about the subject.

are designed to help ensure that an XP project works successfully for significant projects within a large organization:

**Readiness assessment.** The IXP team ascertains whether all members of the project community (e.g., stakeholders, developers, management) are on board, have the proper environment established, and understand the skill levels involved.

**Project community.** The IXP team determines whether the right people, with the right skills and training have been staged for the project. The "community" encompasses technologists and other stakeholders.

**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.

**Test-driven management.** An IXP team establishes a series of measurable "destinations" [Ker05] that assess progress to date and then defines mechanisms for determining whether or not these destinations have been reached.

**Retrospectives.** An IXP team conducts a specialized technical review (Chapter 20) 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.

**Continuous learning.** The IXP team is encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

**Quote:**

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

**Lou Holtz**

In addition to the six new practices discussed, IXP modifies a number of existing 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.**

## 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?

**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.

## 5.5 Other Agile Process Models

**Quote:**

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

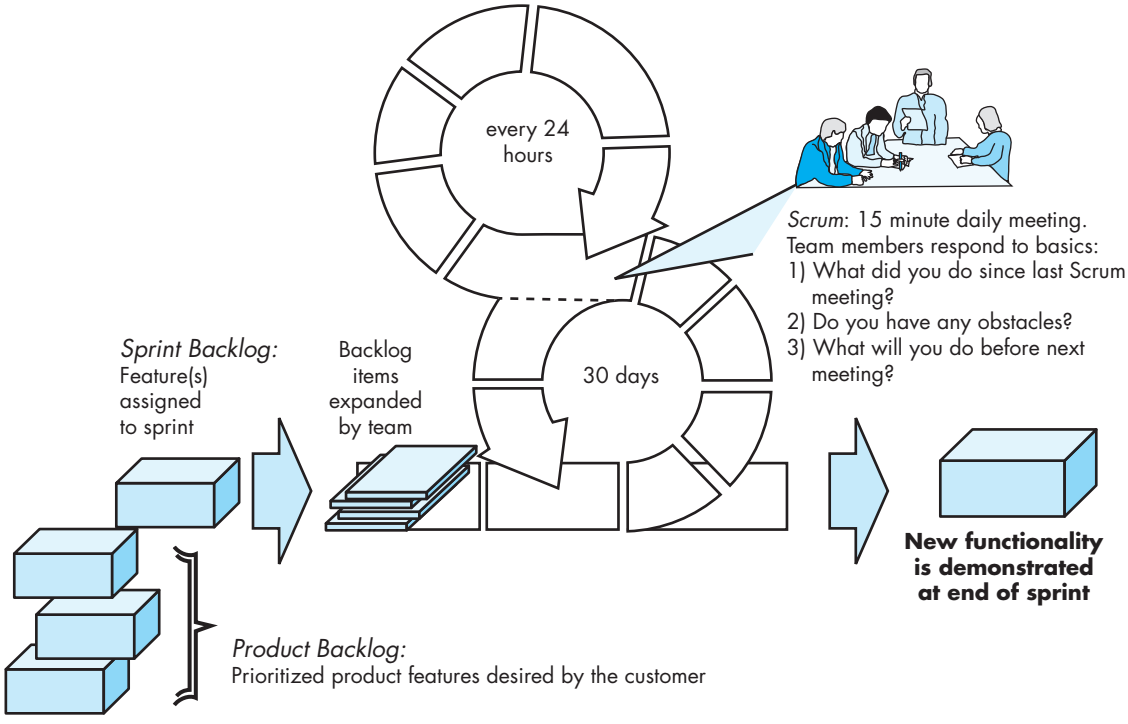
**Stephen Hawrysh and Jim Ruprecht**

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.[8]

As we 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. In this section, we present a brief overview of four common agile methods: Scrum, DSSD, Agile Modeling (AM), and Agile Unified Process (AUP).

8 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.

**FIGURE 5.3** Scrum process flow




### 5.5.1 Scrum

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

*Scrum* (the name is derived from an activity that occurs during a rugby match)[9] 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 [Sch01b].

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 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 5.3.

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

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 activities:


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

*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[10] (typically 30 days). 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-minute) 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.

### 5.5.2 Dynamic Systems Development Method

**WebRef**

Useful resources for DSDM 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

10 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.

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 begins with a *feasibility study* that establishes basic business requirements and constraints and is followed by a *business study* that identifies functional and information requirements. DSDM then defines three different iterative cycles:


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

*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 the 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, the functional model iteration and the 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 5.4) 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.

### 5.5.3 Agile Modeling

**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. But in some cases, it can be daunting to manage 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. Is there an agile approach to software engineering modeling that might provide some relief?

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.

**Quote:**

"I was in the drugstore 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**

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.

**ADVICE**

*"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)[11] as the preferred method for representing analysis and design models. The Unified Process (Chapter 4) 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.

### 5.5.4 Agile Unified Process

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.
- *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 3, 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 29) addresses change management, risk management, and the control of any persistent work products[12] that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

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

12 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.

- *Environment management.* Environmental 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 used in conjunction with any of the agile process models described in this chapter.

**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 4) 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:**[13]

*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 (**http://ideogramic-uml.software.informer.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.

## 5.6 A TOOL SET FOR THE AGILE PROCESS

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."

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

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04] or modern social networking techniques) that provides information and coordination among agile developers. Active communication is achieved via the team dynamics (e.g., pair programming), while

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