CHAPTER

1

SOFTWARE AND SOFTWARE ENGINEERING

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.

KEY CONCEPTS

QUICK LOOK

What is it? Computer software is a work product that software professionals build and then support over many years. These work products include programs that execute within computers of any size and architecture. Software engineering encompasses a process, a collection of methods (practice), and an array of tools that allow professionals to build high-quality computer software.

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

What is the work product? From the software engineer's point of view, the work product is the set of programs, content (data), and other work products that support computer software. But from the user's point of view, the work product is a tool or product 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.

"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.

He paused and thought for a moment. Then he slowly nodded. "We adapt them to our needs, but sure, we use them."

"Where?" I asked, probing.

"Our problem is often translating the requirements the creatives give us."

"The creatives?" I interrupted.

"You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game."

"And after the requirements are established?"

He shrugged. "We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends."

"You know my book?" I was honestly surprised.

"Sure, used it in school. There's a lot there."

"I've talked to some of your buddies here, and they're more skeptical about the stuff in my book."

He frowned. "Look, we're not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques."

"And how will your subset change as the years pass?"

He paused as if to ponder the future. "Games will become bigger and more complex, that's for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don't, we're dead."

*****

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Sixty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the media); that software would be the driving force behind the personal computer revolution; that software applications would be purchased by

consumers using their mobile devices; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than all industrial-era companies; or that a vast software-driven network would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

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 support 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.*

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. The number of people who have an interest in the features and functions provided by a specific application[1] has grown dramatically. *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 increasingly complex with each passing year. Large teams of people now create computer programs. 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 autonomous vehicles. *Design has become a pivotal activity.*
- 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 consequences. *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. *Software should be maintainable.*

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

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

1.1 THE NATURE OF SOFTWARE

Today, software takes on a dual role. It is a product, and 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 device, on the desktop, in the cloud, or within a mainframe computer or autonomous machine, 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 an augmented-reality representation derived from data acquired from dozens of independent sources and then overlaid on the real world. As the vehicle used to deliver a 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 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.

The role of computer software has undergone significant change over the last 60 years. 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:[2]

- 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?

2 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."

These, and many other questions, are a manifestation of the concern about software and how 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:

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

There is no question that other more complete definitions could be offered. But a more formal definition probably won't measurably improve your understanding. To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has 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,

FIGURE 1.1

Failure curve for hardware

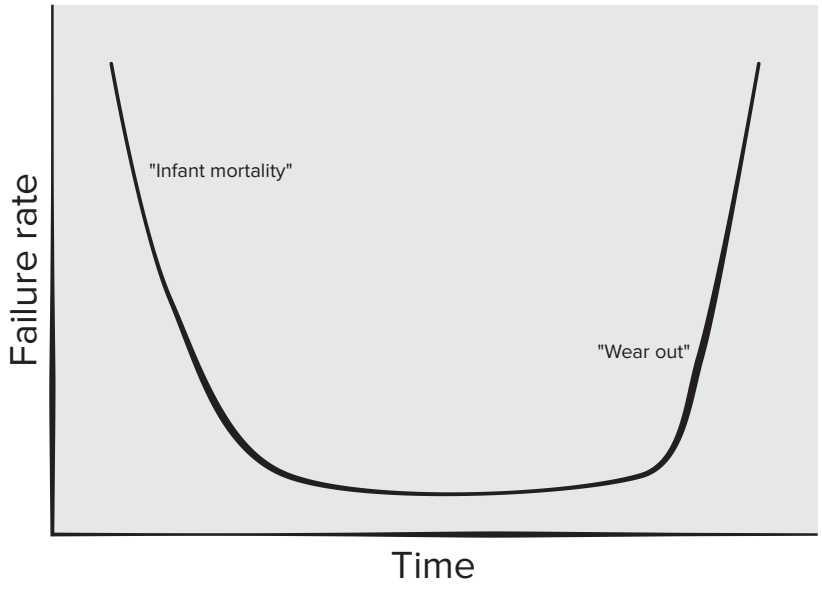
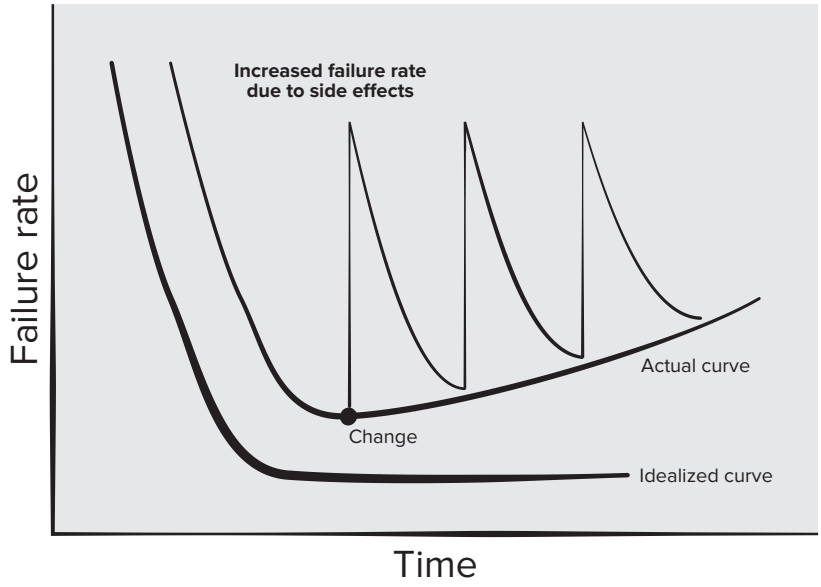


FIGURE 1.2

Failure curves for software



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

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

3 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.

1.1.2 Software Application Domains

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

System software. A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,[4] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

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" or data science programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, from computer-aided design to consumer spending habits, and 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. Composed of reusable components and designed to provide specific capabilities for use by many different customers. It may focus on a limited and esoteric marketplace (e.g., inventory control products) or attempt to address the mass consumer market.

Web/mobile applications. This network-centric software category spans a wide array of applications and encompasses browser-based apps, cloud computing, service-based computing, and software that resides on mobile devices.

Artificial intelligence software. Makes use of heuristics[5] to solve complex problems that are not amenable to regular computation or straightforward analysis. Applications within this area include robotics, decision-making systems, pattern recognition (image and voice), machine learning, 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.

---

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

5 The use of heuristics is an approach to problem solving that employs a practical method or "rule of thumb" not guaranteed to be perfect, but sufficient for the task at hand.

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. 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.

These changes may create an additional side effect that is often present in legacy software—*poor quality*.[6] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, and a poorly managed change history. The list can be quite long. And yet, these systems often support "core 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:

- 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 work with other more modern systems or databases.
- The software must be re-architected to make it viable within an evolving computing environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable in 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 change continually, new software systems can be built from the old ones, and . . . all must interact and cooperate with each other" [Day99].

1.2 Defining the Discipline

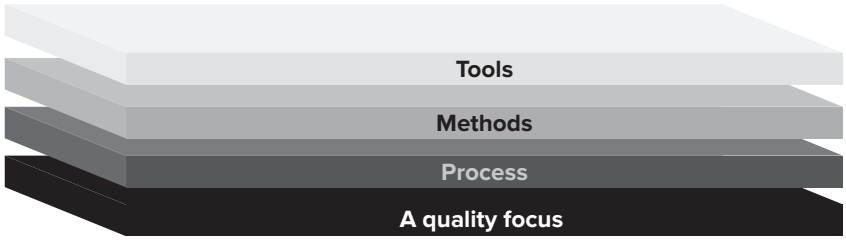
The IEEE [IEE17] has developed the following definition for software engineering:

> Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

6 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.

FIGURE 1.3

Software engineering layers



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 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. You may have heard of total quality management (TQM) or Six Sigma, and similar philosophies[7] that foster a culture of continuous process improvement. It is this culture that ultimately leads to more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

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

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

Software engineering *tools* provide automated or 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.

1.3 THE 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

7 Quality management and related approaches are discussed throughout Part Three of this book.

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.

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.

1.3.1 The Process Framework

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).[8] The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. 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.

---

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

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 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.

1.3.2 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:

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.

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

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

1.3.3 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 One of this book, we examine software process in considerable detail.

1.4 SOFTWARE ENGINEERING PRACTICE

In Section 1.3, 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.[9]

1.4.1 The Essence of Practice

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).

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

Understand the Problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and

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

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?

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.

1.4.2 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., creating a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:[10]

The First Principle: *The Reason It All Exists*

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

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

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 should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the simpler ones. Simple does not mean "quick and dirty." It often takes a lot of thought and work over multiple iterations to simplify the design. 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 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*

Always specify, design, document, 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.

---

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

The Fifth Principle: *Be Open to the Future*

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, systems must be ready to adapt to these and other changes. Systems that do this successfully 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.[11]

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[12] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh Principle: *Think!*

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

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

1.5 How It All Starts

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

11 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.

12 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 building targeted software. In some cases, the cost differential cannot be justified.

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

SAFEHOME[13]

How a Project Starts

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

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

The conversation:

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

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

Joe: You agree, Mal?

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

Joe: How big . . . bottom line big?

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

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

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

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

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

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

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

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

1.6 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 60 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.

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. As you learn more about software engineering, you'll begin to understand why these principles should be considered when beginning any software project.

PROBLEMS AND POINTS TO PONDER

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

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

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

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

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

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

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

1.8. 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?