CHAPTER 1

Introduction

1.1 THE NEED FOR SOFTWARE ENGINEERING

The first commonly known use of the term *software engineering* was in 1968 as a title for a NATO conference on software engineering. An article by A. J. S. Rayl in a 2008 NASA magazine commemorating NASA's fiftieth anniversary states that NASA scientist Margaret Hamilton had coined the term earlier (Rayl, 2008).

The nature of computer software has changed considerably in the last forty-five or so years, with accelerated changes in the last fifteen to twenty. Software engineering has matured to the extent that a common software engineering body of knowledge, known by the acronym SWEBOK, has been developed. See the article "Software Engineering Body of Knowledge" (SWEBOK, 2013) for details. Even with this fine collection of knowledge of the discipline of software engineering, the continuing rapid changes in the field make it essential for students and practitioners to understand the basic concepts of the subject, and to understand when certain technologies and methodologies are appropriate—and when they are not. Providing this foundational knowledge is the goal of this book. You will need this foundational knowledge to be able to adapt to the inevitable changes in the way that software will be developed, deployed, and used in the future.

We begin with a brief history. In the late 1970s and early 1980s, personal computers were just beginning to be available at reasonable cost. There were many computer magazines available at newsstands and bookstores; these magazines were filled with articles describing how to determine the contents of specific memory locations used by computer operating systems. Other articles described algorithms and their implementation in some dialect of the BASIC programming language. High school students sometimes made more money by programming computers for a few months than their parents made in a year. Media coverage suggested that the possibilities for a talented, solitary programmer were unlimited. It seemed likely that the computerization of society and the fundamental changes caused by this computerization were driven by the actions of a large number of independently operating programmers.

However, another trend was occurring, largely hidden from public view. Software was growing greatly in size and becoming extremely complex. The evolution of word processing software is a good illustration.

In the late 1970s, software such as Microsoft Word and WordStar ran successfully on small personal computers with as little as 64 kilobytes of user memory. The early versions of WordStar allowed the user to insert and delete text at will, to cut and paste blocks of text, use italics and boldface to set off text, change character size, and select from a limited set of fonts. A spelling checker was available. A small number of commands were allowed, and the user was expected to know the options available with each command. Lists of commands were available on plastic or cardboard templates that were placed over the keyboard to remind users of what keystroke combinations were needed for typical operations. The cardboard or plastic templates were generally sold separately from the software.

Microsoft Word and WordStar have evolved over time, as has most of their competition, including Apple's Pages word processing software. Nearly every modern word processing system includes all the functionality of the original word processors. In addition, modern word processing software usually has the following features:

- There is a graphical user interface that uses a mouse or other pointing device. (On tablets and smartphones, this interface is based on the touch and movement of one or more fingers.)
- There is a set of file formats in which a document can be opened.
- There is a set of file formats in which a document can be saved.
- There is a set of conversion routines to allow files to be transferred to and from different applications.
- There is a large set of allowable fonts.
- The software has the capability to cut and paste graphics.
- The software has the capability to insert, and perhaps edit, tables imported from a spreadsheet.
- The software has the capability to insert, and perhaps edit, other nontextual material.
- There are facilities for producing word counts and other statistics about the document.
- There are optional facilities for checking grammar.
- The software has the capability for automatic creation of tables of contents and indices.
- The software has the capability to format output for different paper sizes.
- The software has the capability to print envelopes.
- The software has the capability to compose and format pages well enough to be considered for "desktop publishing." Many inexpensive printers are capable of producing high-quality output.
- Automatic backups are made of documents, allowing recovery of a file if an unexpected error in the word processing software or a system crash occurs.

Because of the proliferation of printers, a word processing system must contain a large number of printer drivers. Most word processing systems often include an online help facility.

The added complexity does not come free, however. The late 1990s versions of most word processors required nine 1.44 MB floppy disks for its installation and the executable file itself was larger than four megabytes. Now such software is sold on CDs or DVDs, as downloads, or as a preinstalled option whose cost is included in the price of new computers.

Some are sold both as stand-alone systems and as part of "office suites" that are integrated with other applications, such as spreadsheets, presentation graphics software, and database management software. Microsoft Word is generally sold in such suites. Apple iWorks has recently been made available for free, and many open source packages that include many of the necessary capabilities are available also. Before Apple made the decision to make iWorks free, its three primary components, Pages, Numbers, and Keynote, were sold both separately and as an integrated suite.

The need to support many printers and to allow optional features to be installed increases word processing systems' complexity. Most printer drivers are available as downloads.

The latest versions of most word processing systems consist of many thousands of files. New releases of the word processing software must occur at frequent intervals. If there are no releases for a year or so, then many users who desire additional features may turn to a competitor's product.

Even if a single individual understood all the source code and related data files needed for a new release of the word processing software, there would not be enough time to make the necessary changes in a timely manner. Thus, the competitive nature of the market for word processing software and the complexity of the products themselves essentially force the employment of software development teams. This is typical of modern software development—it is generally done by teams rather than by individuals. The members of these teams are often referred to as "software engineers." Software engineers may work by themselves on particular projects, but the majority of them are likely to spend most of their careers working as part of software development teams. The teams themselves will change over time due to completion of old projects, the start of new projects, and other changes in individuals' careers and rapid technological changes.

The issues involved in the word processing software systems discussed in this section are typical of the problems faced by software engineers. The requirement of being able to cut and paste graphics and tables is essentially forced by the marketplace. This, in turn, requires that the cut-and-paste portion of the word processing software must interface with graphics and spreadsheet applications.

The interface can be created for each pair of possible interoperable applications (word processor and graphics package, word processor and spreadsheet, Internet browser, etc.). Alternately, there can be a single standard interface between each application (word processor, graphics package, spreadsheet, etc.) and the operating system, or some other common software. Figures 1.1 and 1.2 illustrate the two approaches. Note that Figure 1.1 indicates a system that has a central core that communicates between the various applications and the operating system.

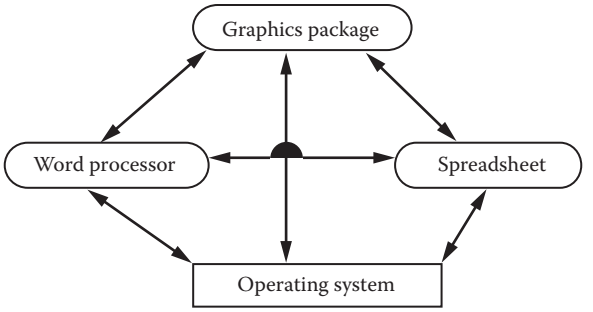

FIGURE 1.1 A complex interconnection system.

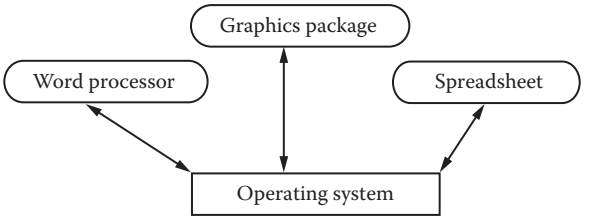

FIGURE 1.2 A conceptually simpler interconnection system.

The design illustrated in Figure 1.2 is conceptually simpler, with the major complications of device drivers, standards, and interfaces hidden in the interfaces. In either case, the word processing software must adhere to a previously specified interface. This is typical of the software industry; software that does not interface to some existing software is rare.

This simple model illustrates the need for a systematic approach to the issue of system complexity. Clearly, the complexity of a software system is affected by the complexity of its design.

There are also quality issues involved with software. Suppose that, in its rush to release software quickly and beat its competitors to market, a company releases a product that contains serious flaws. As a hypothetical example, suppose that a word processor removes all formatting information from a file, including margins, fonts, and styles, whenever the sequence of commands Save, Check spelling, Insert page break, and Save is entered. It is unlikely that any customer using this inadequately tested version of the word processing software would ever use this product again, even if it were free.

Sometimes the decision to add new features to a product may be based on technological factors such as the Internet or the cloud. At the time that this book is being written, several companies that produce word processing software are developing new applications that are network based and require subscriptions instead of an upfront cost. Several options are possible, regardless of whether data is stored either locally or remotely:

1. Have all new software reside on the user's local computer, as is presently the case for word processors for personal computers
2. Have a remote application be invoked over a network whenever the user selects a previously stored local document

3. Have the core of the application reside on the local computer, with specialized features invoked from a remote server only if needed
4. Have both the document and the remote application reside on the remote server

The advantage of the first alternative is that there is no change in the company's strategy or basic system design. The risk is a lack of ability to perform advanced operations such as having a document that can be easily shared by several users who are widely scattered, or that recovery might be compromised. There is also risk of the popular perception that the software is not up to date in its performance.

Using the second alternative means that the distribution costs are essentially reduced to zero and that there is a steady revenue stream automatically obtained by electronically billing users. The software must become more complex when issues such as security of data, security of billing information, and performance response become very important.

The third alternative has some of the best features of the first two. For example, distribution costs are reduced and the minimal core of the software, which resides on a local computer, can be smaller and simpler. Unfortunately, this alternative also shares the common disadvantages of the first two in terms of technology and complexity.

The fourth alternative is the natural extension of the second and third alternatives. There are some performance drawbacks to this approach, as any user of the Internet is aware.

Performance issues are critical because of potential delays in having, say, formatting changes appear quickly on a user's screen if remote storage is used.

There are obvious security issues if a user's critical or confidential data is stored on a remote server beyond the control of a user or the user's organization. The article by Richard Stallman of the Free Software Foundation (Stallman, 2010) is well worth reading on this subject.

Regardless of the choice made, it is clear that most word processing software will become more complex in the future. Word processing programs, including the one used to write this book, must interface with multiple applications to share printers and utility functions. They must also share data with other applications.

The problems just described for producers of word processing software are similar to those of software development organizations producing software for a single client or a small set of potential clients. For example, software used to control the movements of multiple trains sharing the same track system for one geographical area of a country's railway system must be able to interface with software used for similar purposes in another geographical area.

This is not an abstract problem. The coordination of independently developed local systems can be a problem. In Australia, the lack of coordination of even the size of railroad tracks in the different states and territories caused major problems in the development of an Australian national railroad system until national standards were developed and adhered to. There recently have been similar software problems with standardization of railroad speed monitoring in the United States. (Fortunately, in the United States, track sizes have been standardized for many years.)

Similar requirements hold for the computer software used to control aircraft traffic, the software used within the airplanes themselves, the central core of a nuclear power plant, and chemical processes in an oil company, or to monitor the dozens of medicines delivered to a patient in a hospital.

Such systems have an even higher degree of testing and analysis than does word processing software, because human life and safety are at stake. The term *safety-critical* is used to describe such systems. Obviously safety-critical software systems require more care than, for example, computer games.

Note, however, that many computer games are very complex and place many demands on a computer to achieve the level of realism and speed that many users now demand. This is the reason that many games are written for powerful game machines such as those found in the Microsoft Xbox and Nintendo 3DS families than for personal computers. In another direction, some relatively simple games written for smartphones, such as *Angry Birds* with its rudimentary graphics, have been wildly successful.

What will happen to the computer gaming industry? I mentioned the changes planned by game-producing company Zynga in the "Preface to the Second Edition" of this book. These were done in reaction to current and expected market trends.

At a 2008 talk to an overflow crowd at the Engineering Society of Baltimore located in the historic Mount Vernon area of the city, Sid Meier of Firaxis Games said something that amazed members of the audience not familiar with technical details of his career: He programmed in C! It was basically Objective C, but he felt he got the most control of the game-playing rules and logic doing this, leaving development of video, sound, music, and interaction with the object-oriented game engine to others. This division of labor largely continues at Firaxis even now. The point is, many older technologies continue in use for a long time, and are even critically important in what we might consider to be state-of-the-art systems.

Let us summarize what we have discussed so far. We talked about the explosive growth of personal computers in the 1980s and 1990s. The growth has continued to the present day, with computing now done on smartphones, tablets, and other devices. As indicated, many of the initial versions of some of the earlier software products have evolved into very large systems that require more effort than one individual can hope to devote to any project. What is different now?

A moment's thought might make you think that standards such as Hypertext Markup Language (HTML) and the Java programming language with its application programming interfaces have changed everything. There are more recent developments. Application frameworks and cloud computing have been major players, along with highly mobile devices such as smartphones and tablets with their specialized user interfaces.

There are also inexpensive, high-quality application development frameworks and software development kits. There are many sixteen-year-olds who are making a large amount of money as web page designers, although this is relatively less frequent in 2014, since many elementary school students in the United States are taught how to design a web page before they reach their teens. One of the job skills most in demand now is "web master"—a job title that did not even exist in 1993. A casual reading of online job listings might lead you to believe that we have gone back to the more freewheeling days of the 1980s.

Certainly, the effects of technology have been enormous. It is also true that nearly anyone who is so inclined can learn enough HTML in a few minutes to put together a flashy web page.

However, the problem is not so simple. Even the most casual user of the Internet has noticed major problems in system performance. Delays make waiting for glitzy pictures and online animations very unappealing if they slow down access to the information or services that the user desired. They are completely unacceptable if they cannot display properly on many portable devices with small screen "real estate."

Proper design of websites is not always a trivial exercise. As part of instruction in user interface design, a student of mine was asked to examine the main websites of a number of local universities to obtain the answer to a few simple questions. The number of selections (made by clicking a mouse button) ranged from five to eleven for these simple operations. Even more interaction was necessary in some cases because of the need to scroll through online documents that were more than one screen long, or even worse, more than one screen wide. Efficiency of design is often a virtue.

Several issues may not be transparent to the casual user of the Internet. Perhaps the most troublesome is the lack of systematic configuration management, with servers moving, software and data being reorganized dynamically, and clients not being informed. Who has not been annoyed by the following famous message that appears so often when attempting to connect to an interesting website?

> ERROR 404: File not found.

It is obvious what happened to the information that the user wanted, at least if there was no typing error. As we will see later in this book, maintenance of websites is often a form of "configuration management," which is the systematic treatment of software and related artifacts that change over time as a system evolves.

There are also major issues in ensuring the security of data on servers and preventing unwanted server interaction with the user's client computer. Finally, designing the decomposition of large systems into client and server subsystems is a nontrivial matter, with considerable consequences if the design is poor.

It is clear that software engineering is necessary to have modern software development done in an efficient manner. These new technologies have refocused software engineering to include the effects of market forces on software development. As we will see, these new technologies are amenable to good software engineering practice.

We will consider project size and its relationship to software teams in more detail in the next section.

1.2 ARE SOFTWARE TEAMS REALLY NECESSARY?

You might plan to be a developer of stand-alone applications for, say, smartphones, and wonder if all this formality is necessary. Here are some of the realities. There are well over one million apps for iPhones and iPads on the App Store and as a result, it can be hard to

have people locate your app. You may have heard about the programmer who developed an app that sold 17,000 units in one day and quit his job to be a full-time app developer. Perhaps you heard about Nick D'Aloisio, a British teenager who sold an application named *Summly* to Yahoo! The price was $30 million, which is a substantial amount of money for someone still in high school. (See the article at http://articles.latimes.com/2013/mar/26/business/la-fi-teen-millionaire-20130326.)

Some anecdotal information appears to suggest that teams of software engineers are not necessary for some systems that are larger than the typical smartphone app. For example, the initial version of the MS-DOS operating system was largely developed by two people, Bill Gates and Paul Allen of Microsoft. The Apple DOS disk operating system for the hugely successful Apple II family of computers was created by two people, Steve Jobs and Steve Wozniak, with Wozniak doing most of the development. The UNIX operating system was originally developed by Dennis Ritchie and Kenneth Thompson (Ritchie and Thompson, 1978). The first popular spreadsheet program, VisiCalc, was largely written by one person, Dan Bricklin. The list goes on and on.

However, times have changed considerably. The original PC-DOS operating system ran on a machine with 64K of random access, read-write memory. It now runs only as an application in a terminal window whose icon is found in the accessories folder on my PC. The Apple DOS operating system used even less memory; it ran on the Apple II machine with as little as 16K memory. UNIX was originally developed for the PDP-11 series of 16-bit minicomputers. Versions of VisiCalc ran on both the Apple II and the 8086-based IBM PC. The executable size of their successor programs is much, much larger.

Indeed, PC-DOS has long been supplanted to a great extent by several variants of Microsoft Windows such as Windows 8 and 10. Apple DOS was supplanted by Mac OS, which has been supplanted by OS X, Yosemite, and the like. UNIX still exists, but is much larger, with the UNIX kernel, which is the portion of the operating system that always remains in memory, being far larger than the entire memory space of the PDP-11 series computers for which it was originally written. The size of the UNIX kernel led Linus Torvalds to develop a smaller kernel, which has grown into the highly popular Linux operating system.

VisiCalc no longer exists as a commercial product, but two of its successors, Excel and Numbers, are several orders of magnitude larger in terms of the size of their executable files. The current version of Excel consists of millions of lines of source code written in C and related languages. See the book and article by Michael Cusumano and Richard Selby (Cusumano and Selby, 1995, 1997) for more information on Excel.

The additional effort to develop systems of this level of complexity does not come cheap. In his important book *The Mythical Man-Month*, Fred Brooks gives the rule of thumb that a software project that was created by one or two persons requires an additional eight or nine times the original development effort to change it from something that can be used only by its originators to something useful to others (Brooks, 1975). Brooks's rule of thumb appears to be valid today as well. It was interesting to have Brooks's book quoted by Hari Sreenivasan when discussing problems with the deployment of the healthcare.gov website

for enrollment in the insurance exchanges available under the Affordable Care Act (*PBS NewsHour*, October 23, 2013.)

It is important to understand the distinction between initial prototype versions of software, which are often by very small groups of people, and commercial software, which requires much larger organizational structures. Today, any kind of commercial software requires a rock-solid user interface, which is usually graphical, at least on desktop devices, laptops, and larger, full-screen tablets. User interfaces for smartphones may be much simpler in order to be useful on these devices' smaller screens. Heads-up and other wearable devices, such as Google Glass, have their own interface issues, which, unfortunately, are far beyond the scope of this book.

Testing is essential in modern software development, and a technical support organization to answer customers' questions is necessary. Documentation must be clear, complete, and easy to use for both the first-time user who is learning the software and the experienced user who wishes to improve his or her productivity by using the advanced features of the package. Thus, the sheer pressure of time and system size almost always requires multiple individuals in a software project. If development of a software product that fills a need requires six months, multiplication by Brooks's conservative factor of eight means that the software will take at least four years to be ready for release, assuming that the time needed and the effort needed in terms of person-hours scale up in the same way. Of course, the product may be irrelevant in four years, because the rest of the software industry is not standing still. (This is one reason that agile software processes have become more popular.)

Most successful projects that have given birth to software companies have gotten much larger in terms of the number of people that they employ. Clearly, even software that was originally developed by one or two entrepreneurs is now developed by teams. Even the wildly successful companies and organizations that make Internet browsers, the initial versions of which were largely due to the efforts of a small team, are now employers of many software engineers in order to increase the number of available features, to ensure portability to a number of environments, to improve usability by a wide class of users, and even to provide sufficient documentation.

How large should such a software team be? The most important factor seems to be the size of the project, with capability of the team members, schedule, and methodology also important. We will discuss estimation of software project size in Chapter 2. For now, just remember one thing: your software is likely to be developed within a team environment.

Modern teams may be highly distributed, and some major portions of a team project may be totally outsourced. Outsourcing brings its own set of complications, as the experience of two of my relatives, one with a major project in electronic healthcare records systems and the other with the interface from a website to a database, shows. A 2013 article by Lisa Kaczmarczyk has an interesting discussion of some of the cultural issues with coordination of outsourced software development (Kaczmarczyk, 2013). Just keep in mind that the problem of coordination with entities performing this outsourced work can be highly complex, and that you can learn a lot by observing how senior personnel or management in your organization handle the problems with outsourced software that will inevitably occur.

1.3 GOALS OF SOFTWARE ENGINEERING

Clearly organizations involved with producing software have a strong interest in making sure that the software is developed according to accepted industry practice, with good quality control, adherence to standards, and in an efficient and timely manner. For some organizations, it is literally a matter of life and death, both for the organization and for potential users of the software. *Software engineering* is the term used to describe software development that follows these principles.

Specifically, the term *software engineering* refers to a systematic procedure that is used in the context of a generally accepted set of goals for the analysis, design, implementation, testing, and maintenance of software. The software produced should be efficient, reliable, usable, modifiable, portable, testable, reusable, maintainable, interoperable, and correct. These terms refer both to systems and to their components. Many of the terms are self-explanatory; however, we include their definitions for completeness. You should refer to the recently withdrawn, but still available and useful, IEEE standard glossary of computer terms (IEEE, 1990) or to the "Software Engineering Body of Knowledge" (SWEBOK, 2013) for related definitions.

Efficiency—The software is produced in the expected time and within the limits of the available resources. The software that is produced runs within the time expected for various computations to be completed.

Reliability—The software performs as expected. In multiuser systems, the system performs its functions even with other load on the system.

Usability—The software can be used properly. This generally refers to the ease of use of the user interface but also concerns the applicability of the software to both the computer's operating system and utility functions and the application environment.

Modifiability—The software can be easily changed if the requirements of the system change.

Portability—The software system can be ported to other computers or systems without major rewriting of the software. Software that needs only to be recompiled in order to have a properly working system on the new machine is considered to be very portable.

Testability—The software can be easily tested. This generally means that the software is written in a modular manner.

Reusability—Some or all of the software can be used again in other projects. This means that the software is modular, that each individual software module has a well-defined interface, and that each individual module has a clearly defined outcome from its execution. This often means that there is a substantial level of abstraction and generality in the modules that will be reused most often.

Maintainability—The software can be easily understood and changed over time if problems occur. This term is often used to describe the lifetime of long-lived systems such as the air traffic control system that must operate for decades.

Interoperability—The software system can interact properly with other systems. This can apply to software on a single, stand-alone computer or to software that is used on a network.

Correctness—The program produces the correct output.

These goals, while noble, do not help with the development of software that meets these goals. This book will discuss systematic processes and techniques that aid in the efficient development of high-quality software. The software systems that we will use as the basis of our discussion in this book are generally much too large to be developed by a single person. Keep in mind that we are much more interested in the process of developing software, such as modern word processors that will be used by many people, than in writing small programs in perhaps obsolete languages that will be used only by their creators.

1.4 TYPICAL SOFTWARE ENGINEERING TASKS

There are several tasks that are part of every software engineering project:

- Analysis of the problem
- Determination of requirements
- Design of the software
- Coding of the software solution
- Testing and integration of the code
- Installation and delivery of the software
- Documentation
- Maintenance
- Quality assurance
- Training
- Resource estimation
- Project management

We will not describe either the analysis or training activities in this book in any detail. Analysis of a problem is very often undertaken by experts in the particular area of application, although, as we shall see later, the analysis can often benefit from input from software engineers and a variety of potential users. Think of the problem that Apple's software designers faced when Apple decided to create its iOS operating system for smartphones using as many existing software components from the OS X operating system as possible.

Apple replaced a Linux-based file system where *any* running process with proper permissions can access *any* file to one where files are in the province of the particular

application that created them. The changes in the user interface were also profound, using the "capacitive touch" where a swipe of a single finger and a multifinger touch are both part of the user interface.

Unfortunately, a discussion of software training is beyond the scope of this book.

We now briefly introduce the other activities necessary in software development. Most of these activities will be described in detail in a separate chapter. You should note that these tasks are generally not performed in a vacuum. Instead, they are performed as part of an organized sequence of activities that is known as the "software life cycle" of the organization. Several different approaches to the sequencing of software development activities will be described in the next section about software life cycles. For now, just assume that every software development organization has its own individual software life cycle in which the order of these activities is specified.

The requirements phase of an organization's software life cycle involves precisely determining what the functionality of the system will be. If there is a single customer, or set of customers, who is known in advance, then the requirements process will require considerable discussion between the customer and the requirements specialists on the software team. This scenario would apply to the process of developing software to control the flight of an airplane.

If there is no immediately identifiable customer but a potential set of individual customers, then other methods such as market analysis and preference testing might be used. This approach might be appropriate for the development of an app for a smartphone or tablet, or a simple software package for the Internet.

The process of developing software requirements is so important that we will devote all of Chapter 3 to this subject.

The design phase involves taking the requirements and devising a plan and a representation to allow the requirements to be translated into source code. A software designer must have considerable experience in design methodology and in estimating the trade-offs in the selection of alternative designs. The designer must know the characteristics of available software systems, such as databases, operating systems, graphical user interfaces, and utility programs that can aid in the eventual process of coding. Software design will be discussed in Chapter 4.

Coding activity is most familiar to students and needs not be discussed in any detail at this point. We note, however, that many decisions about coding in object-oriented or procedural languages might be deferred until this point in the software life cycle, or they might have been made at the design or even the requirements phase. Since coding standards are often neglected in many first- and second-year programming courses, and they are critical in both large, industrial-type systems and for small apps that are evaluated for quality in an app store before they are allowed to be placed on sale, coding standards will be discussed in some detail. Software coding will be covered in Chapter 5.

A beginning software engineer is very likely to be assigned to either software testing or software maintenance, at least in a large company. Software testing is an activity that often begins after the software has been created but well before it is judged ready to be released to its customer. It is often included with software integration, which is the process

of combining separate software modules into larger software systems. Software integration also often requires integration of preexisting software systems in order to make large ones. Software testing and integration will be discussed in Chapter 6.

Documentation includes much more than simply commenting the source code. It involves rationales for requirements and design, help files, user manuals, training manuals, technical guides such as programming reference manuals, and installation manuals. Even internal documentation of source code is much more elaborate than is apparent to a beginning programmer. It is not unusual for a source code file to have twice as many lines of documentation and comments as the number of lines that contain actual executable code.

After the software is designed, coded, tested, and documented, it must be delivered and installed. Software documentation, delivery, and installation will be discussed in Chapter 7.

The term *software maintenance* is used to describe those activities that are done after the software has been released. Maintenance includes correcting errors found in the software; moving the software to new environments, computers, and operating systems; and enhancing the software to increase functionality. For many software systems, maintenance is the most expensive and time-consuming task in the software development life cycle. We have already discussed maintenance of stand-alone apps. Software maintenance will be discussed in more detail in Chapter 8.

Of course, all these activities must be coordinated. Project management is perhaps the most critical activity in software engineering. Unfortunately, it is difficult to truly understand the details of project management activities without the experience of actually working on a large-scale software development project. Therefore, we will be content to present an overview of project management activities in Chapter 2. Each of the later chapters will also include a section presenting a typical managerial viewpoint on the technical activities discussed in the chapter.

Quality assurance, or QA, is concerned with making certain that both the software product that is produced and the process by which the software is developed meet the organization's standards for quality. The QA team is often responsible for setting the quality standards. In many organizations, the QA team is separate from the rest of the software development team. QA will be discussed throughout the book rather than being relegated to a separate chapter. We chose this approach to emphasize that quality cannot simply be added near the end of a software project's development. Instead, the quality must be a result of the engineering process used to create the software.

1.5 SOFTWARE LIFE CYCLES

In the previous section, we discussed the different activities that are typically part of the systematic process of software engineering. As we saw, there are many such activities. Knowing that these activities will occur during a software product's lifetime does not tell you anything about the timing in which these activities occur. The activities can occur in a rigid sequence, with each activity completed before the next one begins, or several of the activities can occur simultaneously. In some software development organizations,

most of these activities are performed only once. In other organizations, several of the activities may be iterated several times, with, for example, the requirements being finalized only after several designs and implementations of source code have been done. The timing of these activities and the choice between iterative and noniterative methods are often described by what are known as software development models.

We will briefly discuss six basic models of software development life cycles in this introductory section:

- Classical waterfall model
- Rapid prototyping model
- Spiral model
- Market-driven model
- Open source development model
- Agile development model

(Only the first four life cycle models listed were described in the first edition of this book. There is a seventh software development life cycle, called hardware and software concurrent development, or codevelopment, that is beyond the scope of this book.)

Each of these software life cycle models will be discussed in a separate section. For simplicity, we will leave documentation out of our discussion of the various models of software development. Keep in mind that the models discussed describe a "pure process"; there are likely to be many variations on the processes described here in almost any actual software development project.

1.5.1 Classical Waterfall Model

One of the most common models of the software development process is called the classical waterfall model and is illustrated in Figure 1.3. This model is used almost exclusively in situations where there is a customer for whom the software is being created. The model is essentially never applied in situations where there is no known customer, as would be the case of software designed for the commercial market to multiple retail buyers.

In the simplest classical waterfall model, the different activities in the software life cycle are considered to occur sequentially. The only exception is that two activities that are adjacent in Figure 1.3 are allowed to communicate via feedback. This artificiality is a major reason that this model is no longer used extensively in industry. A second reason is that technology changes can make the original set of requirements obsolete. Nonetheless, the model can be informative as an illustration of a software development life cycle model.

A life cycle model such as the one presented in Figure 1.3 illustrates the sequence of many of the major software development activities, at least at a high level. It is clear, for example, that specification of a system's requirements will occur before testing of individual modules. It is also clear from this abstraction of the software development life cycle

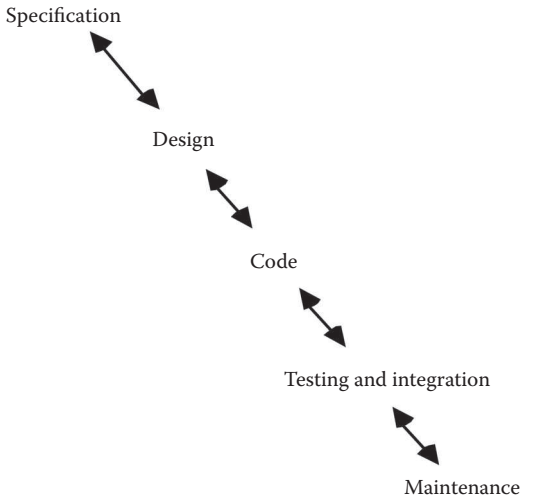

FIGURE 1.3 A simplified illustration of the classical waterfall life cycle model.

that there is feedback only between certain pairs of activities: requirements specification and software design; design and implementation of source code; coding and testing; and testing and maintenance.

What is not clear from this illustration is the relationship between the different activities and the time when one activity ends and the next one begins. This stylized description of the classical waterfall model is usually augmented by a sequence of milestones in which each two adjacent phases of the life cycle interface. A simple example of a milestone chart for a software development process that is based on the classical waterfall model is given in Figure 1.4.

Often, there are several opportunities for the software system's designers to interact with the software's requirements specification. The requirements for the system are typically presented in a series of requirements reviews, which might be known by names such as the preliminary requirements review, an intermediate requirements review, and the

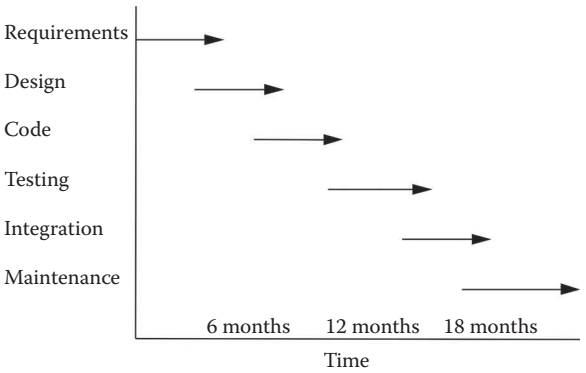

FIGURE 1.4 An example of a milestone chart for a classical waterfall software development process.

final (or critical) requirements review. In nearly all organizations that use the classical waterfall model of software development, the action of having the requirements presented at the critical requirements review accepted by the designers and customer will mark the "official" end of the requirements phase.

There is one major consequence of a decision to use the classical waterfall model of software development. Any work by the software's designers during the period between the preliminary and (accepted) final, or critical, requirements reviews may have to be redone if it is no longer consistent with any changes to the requirements. This forces the software development activities to essentially follow the sequence that was illustrated in Figure 1.3.

Note that there is nothing special about the boundary between the requirements specification and design phases of the classical waterfall model. The same holds true for the design and implementation phases, with system design being approved in a preliminary design review, an intermediate design review, and the final (or critical) design review. The pattern applies to the other interfaces between life cycle phases as well.

The classical waterfall model is very appealing for use in those software development projects in which the system requirements can be determined within a reasonable period and, in addition, these requirements are not likely to change very much during the development period. Of course, there are many situations in which these assumptions are not valid.

1.5.2 Rapid Prototyping Model

Due to the artificiality of the classical waterfall model and the long lead time between setting requirements and delivering a product, many organizations follow the rapid prototyping or spiral models of software development. These models are iterative and require the creation of one or more prototypes as part of the software development process. The model can be applied whether or not there is a known customer. The rapid prototyping is illustrated in Figure 1.5. (The spiral model will be discussed in Section 1.5.3.)

The primary assumption of the rapid prototyping model of software development is that the complete requirements specification of the software is not known before the software

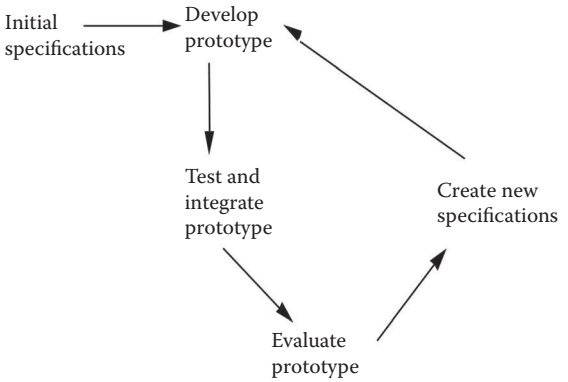

FIGURE 1.5 The rapid prototyping model of the software development life cycle.

is designed and implemented. Instead, the software is developed incrementally, with the specification developing along with the software itself. The central point of the rapid prototyping method is that the final software project produced is considered as the most acceptable of a sequence of software prototypes.

The rapid prototyping method will now be explained in more detail, illustrating the basic principles by the use of a system whose development history may be familiar to you: the initial development of the Microsoft Internet Explorer network browser. (The same issues apply to the development of the Mozilla Firefox, Apple Safari, and Google Chrome browsers, to a large extent.) Let us consider the environment in which Microsoft found itself. For strategic reasons, the company decided to enter the field of Internet browsers. The success of Mosaic and Netscape, among others, meant that the development path was constrained to some degree. There had to be support for HTML and most existing web pages had to be readable by the new browser. The user interface had to have many features with the same functionality as those of Netscape and Mosaic. However, the user interfaces had to be substantially different from those systems in order to avoid copyright issues.

Some of the new system could be specified in advance. For example, the new system had to include a parser for HTML documents. In addition, the user interface had to be mouse driven.

However, the system could not be specified fully in advance. The user interface would have to be carefully tested for usability. Several iterations of the software were expected before the user interface could be tested to the satisfaction of Microsoft. The users testing the software would be asked about the collection of features available with each version of Internet Explorer. The user interface not only had to work correctly, but the software had to have a set of features that was of sufficient perceived value to make other potential users change from their existing Internet browsers if they already used one or purchase Microsoft Internet Explorer if they had never used such a product before.

Thus, the process of development for this software had to be iterative. The software was developed as a series of prototypes, with the specifications of the individual prototypes changing in response to feedback from users. Many of the iterations reflected small changes, whereas others responded to disruptive changes such as the mobile browsers that work on smaller screens on smartphones and tablets with their own operating systems and the need to interoperate with many Adobe products, for example

Now let us consider an update to this browser. There have been changing standards for HTML, including frames, cascading style sheets, a preference for the HTML directive <p> instead of , and a much more semantically powerful version, HTML 6.

Here the browser competition is between IE and several browsers that became popular after IE became dominant, such as Mozilla's Firefox, Apple's Safari, and Google's Chrome. Of course, there are many software packages and entire suites of tools for website development, far too many to mention. Anyone creating complex, graphics-intensive websites is well aware of the subtle differences between how these browsers display some types of information.

Not surprisingly, there are many web page coding styles that once were popular but have become obsolete. Following is an example of the effect of coding styles, given in the context

of a software engineering project that illustrates the continued use of multiple prototypes. The project shows some of the steps needed for a prototype-based software project. You will recognize many of these steps that will appear in one form or another throughout this book.

As part of my volunteer activities for a nonprofit cultural organization located in Maryland, I had to reconstruct a complex website that had been taken over by criminal intruders who had turned it into a SPAM relay site. As a response to the SPAMming, the hosting company disallowed access to the website.

The problem was caused primarily by a combination of a weak password set by the website's initial creator and a plaintext file in a third-party application that did not protect the security of an essential mySQL database. I also thought that the website's hosting company had poor security, so I was happy to move the website to a new hosting company. The problem was exacerbated by the fact that the software originally designed to create the website was no longer available and, therefore, could not be placed on my personal computer that I used to recreate the website. (This unavailability of the software that was originally used to create a particular software artifact is all too common in the software engineering community, due to changes in companies and organizations that produce applications, operating systems, compilers, development environments, and so on.)

Here are the steps that had to be followed:

1. Move the HTML and other files from the organization's website at the original hosting company to my home computer.
2. Identify the directory structure used to organize the files that comprised the website.
3. Examine the HTML of the main pages in each directory to determine their function. This meant looking for files that were named something like index.htm or index.html.
4. Develop a naming standard for file names, and rename files as necessary to make sure that links worked in a consistent manner.
5. Keep a log of all changes, with the results of the changes. This was invaluable, since I could only work intermittently on this problem when I had free time for this volunteer activity.
6. Test the links in the pages to see how the directory structure was reflected in the portion of the website that was on my home computer. Fix them if possible, editing the raw HTML.
7. While the previous steps were occurring, evaluate new website creation tools to determine if it would be better to develop the website using a tool or raw HTML. Cost was a major issue for this cash-strapped organization. This evaluation of available tools is part of a systems engineering effort that we will meet again in Chapter 3.
8. Begin porting files to the hosting environment at the new hosting company. (I had already decided that hosting this service on my home computer was out of the question for security reasons and my own contract with my Internet service provider.)

9. Make the initial renewed website minimally functional, with only static web pages with working links.
10. Test the website iteratively as it is being created, making sure that all links work directly.
11. Remove unnecessary programming, such as JavaScript programming on otherwise simple pages for which there was little benefit to the programming. This was done for security purposes.
12. Release this portion of the website to the members and the entire outside world.
13. Replace the database software using a new version of mySQL, together with the updated version of our application software. This was all done using improved security access. Place all of this inside a newly created password-protected area of the website used exclusively for our members.
14. Give access to a select few individuals to test the security features throughout the reengineering process. Using the results of their tests, I would fix all problems as necessary.
15. Release a final version to the user community.

We now consider each of these steps in turn.

Fortunately, I was given permission from the website's original Internet hosting service to access the account long enough to move all the website's files onto my home computer. I was able to use a very fast freeware software utility, *Filezilla*, which used the FTP file transfer protocol for concurrent file transfer, which was much faster than using a batch transfer of each directory and each subdirectory.

Of course, the original website creator did not have a complete backup of all the files. The files were named without adherence to any consistent file naming standard, so there was a long learning curve to see just what each file contained, and what hyperlinks went with which file.

To make matter worse, Netscape *Communicator*, the software tool used to create the website, no longer worked on the new hosting company's system. In fact, Netscape as a company no longer existed.

Experimentation with the software tools available convinced me that I had to recreate most of the website's pages from scratch, editing the raw HTML. Clearly the only sensible approach was an iterative one, with the detailed requirements and the implementation evolving together.

I did a simple calculation of the likely amount of data that would be stored on the new website, and what was the likely amount of data that would be transferred each month. Armed with this information, I chose a particular package from the hosting company. (Since the cost was small, I decided to absorb the hosting cost myself to avoid getting permission from a committee for each expenditure. It was easier for me to get a letter at the end of the year thanking me for my expenses on behalf of the organization than to ask for approval

of each small expenditure. Armed with this letter, I could take the expense as a charitable contribution.) I then uploaded all the directories to the new web hosting company's site.

A portion of the original HTML created using the obsolete and unavailable website design software is shown next. It really looks ugly. I note that a few years ago, an undergraduate student named Howard Sueing, who is now a highly successful computer consultant, was working on a HTML translator that flagged oddities in HTML 4 that would cause problems if executed in HTML 5, and he described this code as "immature." It was immature, as you can see from the following HTML code fragment whose sole purpose is to insert a clickable tab labeled "Md-MAP" into a fixed position on a particular web page.

```
beginSTMB("auto","0","0","vertically","","0","0","2","3","#ff0000","","ti
   led",
"#000000","1","solid","0","Wipe right",
"15","0","0", "0","0","0", "0","2",
"#7f7f7f","false","#000000","#000000",
"#000000","simple");
appendSTMI("false","Md-MAP ","center","middle","","","-1",
   "-1","0","normal","#fbc33a","#ff0000","","1","-1","-1","blank.gif",
   "blank.gif","-1","-1","0","","mdmap01.html","_self","Arial","9pt","#0000
   00","normal","normal","none","Arial","9pt","#ffffff","normal","normal",
   "none","2","solid","#000000","#000000","#000000","#000000","#000000
   ","#000000","#000000","#000000","mdmap01.html","","","tiled","tiled");
…
endSTMB();
```

Note that there is little separation of objects and attributes, which is a common theme of object-oriented programming. What you do not see is that I had to manually enter carriage returns throughout the code, because these HTML statements had been produced without any breaks, so that what is shown in this book as two blocks that consisted of multiple lines of code was available only as two very long lines. Clearly, one of the first steps of the process of changing the code was to insert these hard carriage returns and to check that each insertion left the functionality of the relevant web page unchanged. (The three dots near the end of the code fragment reflect that many lines of HTML code have been omitted for reasons of clarity.)

Once I fixed the line breaks, so that I could intelligently read the code, I looked at the file structure and developed some naming conventions for file names. I then meticulously loaded the web pages one at a time, checking all hyperlinks and changing file names to the standard naming conventions as I went along. I created a simple spreadsheet indicating all the changes I made and whether they solved the problems I found.

I made the site operable as soon as possible, concentrating on the static web pages. I removed items such as pages that used JavaScript, but that did not, in my opinion, have enough functionality to justify the security problems in older versions of JavaScript. After this minimal website was up and running, and working properly, I could concentrate on the more dynamic and troublesome capabilities that I needed to include.

The last two steps were installation of a database system that was built on top of a mySQL database and the creation of a protected area for access by organization members only. Discussion of the database and security issues encountered would take us too far from our discussion of software engineering.

It was surprising to find that the primary problem with our members accessing the members-only area of the website was that their passwords often did not work because of the need to clear their browser's data cache. Many of the browsers available on user's computers had different ways of treating cached data and I had to respond to each user separately. It was clear to me that I hated doing customer service, but I gained an increased appreciation of the people who work in technical support at help desks. By this time, the project, which had been a combination of reengineering of the old website, reusing much of the code, and developing multiple prototypes (each updating of the website), had morphed into maintenance of the website.

Look over the description of the recreation of the website described earlier from the perspective of an iterative software engineering project. It is clear that several things were necessary for an efficient iterative software development life cycle. There must be an initial step to begin the process. In the rapid prototyping model described in Figure 1.5, an initial set of specifications is needed to start the prototyping process. In our example of reengineering a website, the specifications were to reproduce the functionality of the original website as much as possible.

There must also be an iterative cycle in order to update and revise prototypes. This is clear from the diagram in Figure 1.5. This iteration was the basic approach used to reengineer the aforementioned website.

Finally, there must be a way to exit from the iteration cycle. This step is indicated in Figure 1.5 by the "evaluate prototype" step. The basis for the evaluation is the perceived completeness, correctness, functionality, and usability of the prototype. This evaluation varies from organization to organization. In many cases, the final prototype is delivered as the final product; in others, the final prototype serves as the basis for the final delivered product. This was the case for the website reengineering project.

Was the project successful? The recreated website was up and running, it had the same functionality as the original website, the design of the individual webpages was made more consistent, and some security flaws were fixed. The recreated website was easy to modify. This project was certainly a success.

A milestone chart can also be developed for the rapid prototyping method of software development. A sample milestone chart for the rapid prototyping process model is illustrated in Figure 1.6. (The time periods shown are pure fiction.) Compare this chart to the milestone chart shown in Figure 1.4 for the classical waterfall software development process.

1.5.3 Spiral Model

Another iterative software development life cycle is the spiral model developed by Barry Boehm (Boehm, 1988). The spiral model differs from the rapid prototyping model primarily in the explicit emphasis on the assessment of software risk for each prototype during the evaluation period. The term *risk* is used to describe the potential for disaster in the

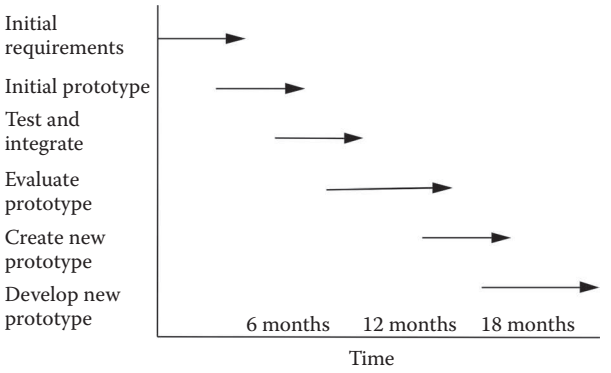

FIGURE 1.6 An example of a milestone chart for a rapid prototyping software development process.

software project. Note that this model is almost never applied to projects for which there is no known customer.

Clearly there are several levels of disasters, and different organizations and projects may view identical situations differently. Examples of commonly occurring disastrous situations in software development projects include the following:

- The software development team is not able to produce the software within the allotted budget and schedule.
- The software development team is able to produce the software and is either over budget or schedule but within an allowable overrun (this may not always be disastrous).
- The software development team is not able to produce the software within anything resembling the allotted budget.
- After considerable expenditure of time and resources, the software development team has determined that the software cannot be built to meet the presumed requirements at any cost.

Planning is also part of the iterative process used in the spiral development model.

The spiral model is illustrated in Figure 1.7.

As we saw before with the classical waterfall and rapid prototyping software development models, a milestone chart can be used for the spiral development process. This type of milestone chart is illustrated in Figure 1.8. Compare the illustration in Figure 1.8 to the milestone charts illustrated in Figures 1.4 and 1.6. For simplicity, only a small portion of a milestone chart is included.

Both the rapid prototyping and spiral models are classified as iterative because there are several instances of intermediate software that can be evaluated before a final product is produced.

It is easy to see the difference between iterative approaches and the classical waterfall model; the waterfall model has no provision for iteration and interaction with a potential

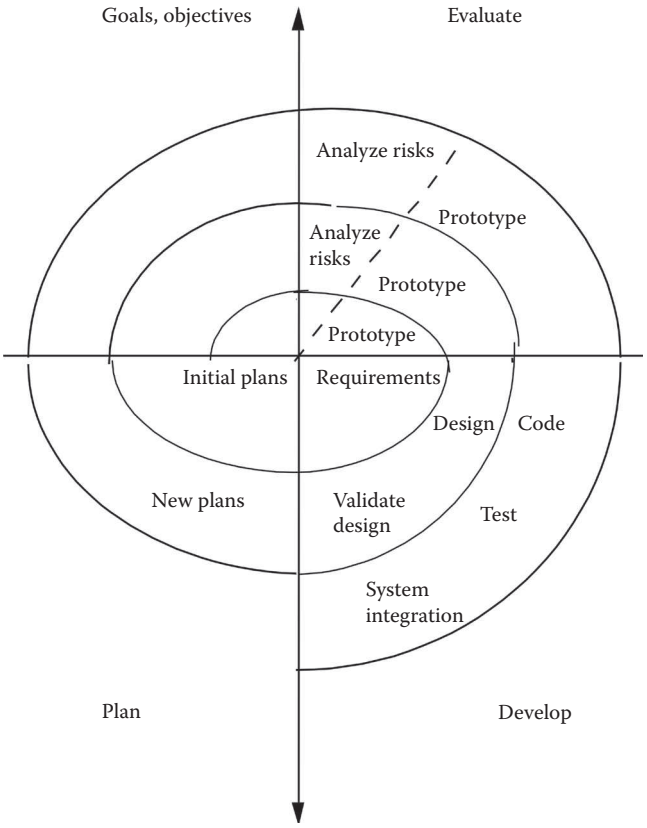

FIGURE 1.7 The spiral model of the software development life cycle.

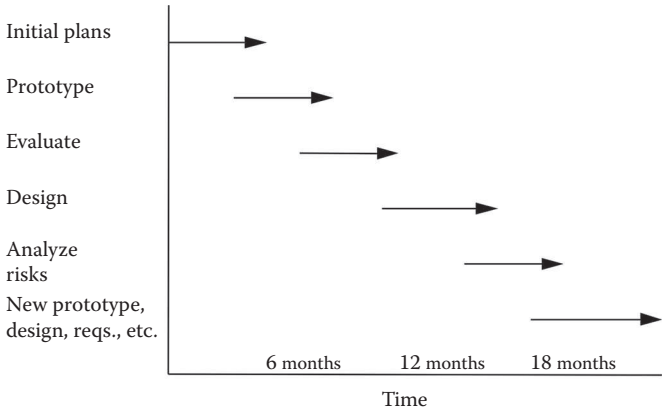

FIGURE 1.8 An example of a milestone chart for a spiral software development process.

customer after the requirements are complete. Instead, the customer must wait until final product delivery by the development organization.

Many, but not all, organizations that use the classical waterfall model allow the customer to participate in reviews of designs. Even if the customer is involved in evaluation of detailed designs, this is often the last formal interaction between the customer and the

developer, and the customer is often uneasy until the final delivery (and often even more uneasy after delivery). It is no wonder that iterative software development approaches are popular with customers.

1.5.4 Market-Driven Model of Software Development

It should be clear that none of the models of the software development life cycle previously described in this book are directly applicable to the modern development of software for the general consumer market. The models previously discussed assumed that there was time for relatively complete initial requirements analysis (as in the classical waterfall model) or for an iterative analysis (as in the rapid prototyping and spiral models with their risk assessment and discarding of unsatisfactory prototypes). These models do not, for example, address the realities of the development of software for personal computers and various mobile devices. Here, the primary pressure driving the development process is getting a product to market with sufficient quality to satisfy consumers and enough desirable new features to maintain, and even increase, market share.

This is a relatively new approach to software development and no precise, commonly accepted models of this type of software development process have been advanced as yet, at least not in the general literature. The reality is that the marketing arm of the organization often drives the process by demanding that new features be added, even as the product nears its target release date. Thus there is no concept of a "requirements freeze," which was a common, unwritten assumption of all the previous models at various points.

We indicate the issues with this type of market-driven "concurrent engineering" in the stylized milestone chart illustrated in Figure 1.9. We have reduced the time frame from the previous milestone charts to reflect the reality that many products have several releases each year.

We emphasize that even small applications written by a single programmer have an expected maintenance cycle, leading to new releases. Suppose, for example, that you have written an app for a mobile phone that has sold a reasonably large number of downloads. If an error is found or if the underlying technology changes, the app must be updated. If

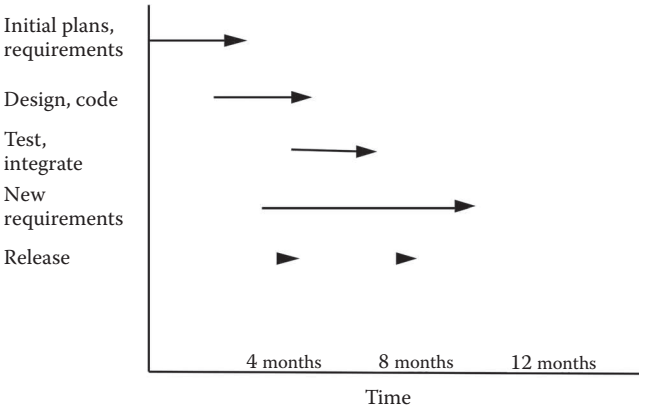

FIGURE 1.9 An example of a milestone chart for a market-based, concurrently engineered software development process.

the developer does not do so or is slow in making the changes, bad reviews result, and sales will plummet.

Even the most casual observer of the early twenty-first century will note that many of the most successful software companies were headed by visionaries who saw an opportunity for creating something that would be useful to huge numbers of people. I hope that some of the readers of this book have these insights and develop wildly successful software accordingly.

Fortunately, the software that anyone would now consider creating for a market of unknown customers is much easier to distribute than in the past. It will almost certainly be delivered by download, relieving the software's creator from the drudgery of creating a large number of copies of the software on CD or DVD, and arranging with stores for shelf space and maintenance of an inventory.

1.5.5 Open Source Model of Software Development

Open source software development is based on the fundamental belief that the intellectual content of software should be available to anyone for free. Hence, the source code for any project should be publicly available for anyone to make changes to it. As we will discuss later in this section, "free" does not mean "no cost," but refers to the lack of proprietary ownership so that the software can be easily modified for any new purpose if necessary.

Making source code available is the easy part. The difficulty is in getting various source code components developed by a wide variety of often geographically separated people to be organized, classified, maintainable, and of sufficient quality to be of use to themselves and others. How is this done?

In order to answer this question we will separate our discussion of open source projects into two primary categories: (1) projects such as new computer languages, compilers, software libraries, and utility functions that are created for the common good; and (2) reuse of the open source versions of these computer languages, compilers, software libraries, and utility functions to create a new software system for a specific purpose.

Here is a brief overview of how many open source projects involving computer languages, compilers, software libraries, and utility functions appear to be organized. The primary model for this discussion of open source software development is the work done by the Free Software Foundation, although there also are aspects of wiki-based software development in this discussion.

An authoritative person or group decides that a particular type of software application or utility should be created. Initial requirements are set and a relatively small number of software modules are developed. They are made publicly available to the open source community by means of publication on a website, possibly one separate from the one that contains relatively complete projects.

The idea is that there is a community of developers who wish to make contributions to the project. These developers work on one or more relevant source code modules and test the modules' performance to their own satisfaction, then submit the modules for approval by the authoritative person or group. If the modules seem satisfactory, they are uploaded to a website where anyone can download them, read the source code, and test the code. Anyone can make comments about the code and anyone can change it. The only rule is that the

changes must be made available to anyone, as long as the source code is always made available, without charge.

It is the use of the achievements and creations of this widespread community with its free access that is the basis for open source software development.

How do members of the large group of open source programmers make money if they are not allowed to charge for their software creations? There are two common ways. The most senior and authoritative people often have substantial contracts to develop systems, with the only proviso that the source code for the software produced by open source organization be made available publicly free of charge. These large contracts are a major part of the funding that keeps the Free Software Foundation's continued operation possible, in addition to donations from volunteers and members of the various user communities.

In addition to the aforementioned funding, members of the open source community often make money by doing software consulting, using their detailed knowledge of the internal structure of many important software products that were developed as open source systems. Being a developer of some of the major modules used in a system is rather strong evidence of just how the system works and what issues are involved in the system's installation, configuration, and operation.

Note that many of the programmers who volunteer their time and effort to create open source modules for various projects are often employed by companies that develop large software systems. This appears to be a way that these volunteers feel that they can be more creative, since they can decide which projects are most interesting to them, rather than some of the projects to which they have been assigned. There is anecdotal evidence to support this view. I heard a statement to this effect while interacting with colleagues at an academic software engineering conference. When Bill Gates visited Howard University in 2006 while I was department chair, I asked him if he was aware of this, and he agreed that it was a likely scenario. (Howard University had sent more students to work at Microsoft in the 2004–2005 academic year than any other college or university in the United States, and Gates came to see us as part of a six-university tour that was intended to increase enrollment in computer science. Microsoft is not typically thought of as a company known for development of open source software.)

We now consider the second category of open source development: the reuse of the open source versions of these computer languages, compilers, software libraries, and utility functions in order to create a new software system for a specific purpose.

We have not attempted to give a precise definition of open source software development because of the variations in the way that it is used in actual practice. Note that open source does not necessarily mean free.

A July 28, 2014, article by Quentin Hardy in the *New York Times* describes how a company, Big Switch Networks, is moving from a development model that was entirely open source to one that includes a parallel development of completely open source software together with one in which many of the more commercially desirable features are proprietary, and therefore, not free. This approach has some things in common with the approach of several resellers of Linux software, especially Red Hat (now Fedora), Ubuntu, and Debian, which provide many things for free but sell consulting and some other services. In

particular, some companies prefer to pay for the maintenance of open source software that they use rather than depend on volunteers.

For another perspective on the distinction between open source software and cost-free software, see the philosophy stated on the Free Software Foundation website at the URL https://www.gnu.org/philosophy/free-software-for-freedom.html. We will not discuss this issue of the precise definition of open source software any further in this book.

1.5.6 Agile Programming Model of Software Development

The roots of the concept of agile development can be traced back to at least as far as the "Skunk Works" efforts to engineer projects at the company now known as Lockheed Martin. (A *Wikipedia* article accessed May 5, 2015, states that the term *Skunk Works* "is an official alias for Lockheed Martin's Advanced Development Programs [ADP], formerly called Lockheed Advanced Development Projects.") Skunk Works came to the fore by the early 1940s, during World War II.

The term is still used to describe both highly autonomous teams and their projects, which are usually highly advanced. Many of the Skunk Works projects made use of existing technologies, a forerunner for component-based software development, with the efficient use of existing software components that can be as large as entire subsystems.

The principles of what is now known as agile programming are best described by the "Agile Manifesto," which can be found at http://agilemanifesto.org/principles.html and whose basic principles are listed here for convenience.

PRINCIPLES BEHIND THE AGILE MANIFESTO

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Some of these principles may seem obvious.

- Agile development is a form of prototyping.
- It depends on a large existing body of high-quality, working components that are especially well-understood by the development team.
- Due to the availability of toolkits, this model also works well for software in the form of apps that are intended for mobile devices. A primary example of this is the availability of free frameworks for the development of computer games.

The term *scrum* is often used in the context of agile development. Originally, the term was frequently used to describe the meetings at the end of each day to coordinate the ideas and accomplishments of the major agile stakeholders. The term was motivated by the sport of rugby. More often, the term *scrum* is used to describe the products of the important organization scrum.org. Often, the term is capitalized, and that is the pattern we will follow henceforth.

We will return to agile software development many times throughout this book.

1.5.7 Common Features of All Models of Software Development

Many organizations have their own written variants of these software development models, and recognize that any models are stylized approximations to actual software development practice in any case. Even a large defense contractor working on a multimillion dollar software project for the government might have to develop some prototypes in an iterative manner, due to disruptive changes in technology. A more market-driven life cycle might have many more things happening at the same time. We will not pursue these different nuances in life cycle models any further in this book.

There are many variants of these six life cycle models, but these are the most commonly used. Note that most of the activities in each of these life cycle models are similar. The primary differences are in the timing of activities and the expectation that some portions of the initial prototypes will be discarded if iterative approaches, such as the rapid prototyping or spiral development models, are used.

The point about timing of life cycle activities cannot be overemphasized. The various life cycle models are, at best, stylized descriptions of a process. Most organizations have their own set of procedures to be followed during software development. Of course, you should follow the guidelines set by your employer. Just be aware that these are the same sets of essential activities that must always occur in software development, regardless of the order in which they occur.

1.5.8 Software Evolution and the Decision to Buy versus Build versus Reuse versus Reengineer

Almost all software products, other than relatively small stand-alone programs written by a single programmer for his or her own use, will change over time. For example, in May 2014 the version of the iOS operating system for my iPhone was version 7.1.1, the operating system for my MacBook Air was 10.9.2, and the version of Word was 14.4.1.

My desktop PC was running a version of Windows 7 (Windows 7 Professional) and also can run an emulator for Windows XP that I needed for several years in order to run a particular version of publication software. (I no longer use the Windows XP emulator, because Microsoft has ended support for this very old operating system.) My desktop Mac uses the same operating system version as my MacBook Air.

The same situation held true for smaller devices. The app 2Screens Remote that allows my iPhone to control Apple Keynote presentations that had been loaded to the 2Screens app on my iPad was 1.5.2. I received notification that version 2.6.1 of the Fly Delta app is available and that I should upgrade to this latest release via the App Store.

You should think of all this software as evolving over time, whether with error fixes, increased features, or the need to have, say, applications software work with a new operating system. The term *software maintenance* is often used to describe this evolution. Clearly, this evolution of the software requires an effort in, at least, design, coding, testing, and integration. This effort obviously involves time, personnel, and other resources. With the typical movement of software engineers from one position to another, either inside or, most commonly, outside companies, there are real decisions to be made about maintaining this evolving software.

For long-lived systems, it is estimated that more than 80 percent of total costs during the software's useful life is during the maintenance phase after the initial delivery. Dan Galorath, the creator of the SEER software cost estimation tool, uses the figure 75 percent in his models. See the SEER website http://www.galorath.com/index.php/software_maintenance_cost. One reason these costs are so high is that it takes a long time for software maintenance engineers to learn the detailed structure of source code that is new to them.

If the expected costs to maintain the software for the rest of its expected life are too high, it may be more sensible to buy an existing software package to do many of the same things rather than incur high maintenance costs. Other alternatives may be to reengineer the software so that existing off-the-shelf software components can be used, or to have even greater percentages of software reused from other projects.

1.6 DIFFERENT VIEWS OF SOFTWARE ENGINEERING ACTIVITIES

As we have seen earlier in this chapter, most modern software development projects require teams. There are two obvious ways to organize teams: ensure that each person is a specialist with unique skills; have each person be a generalist, who is able to perform most, if not all, team responsibilities; or some combination of the two. Each of the organizational methods has unique advantages and disadvantages.

A team with specialists is likely to have individual tasks done more efficiently than if they were to be done by a generalist. Some technologies are so complex and are changing so rapidly that only a specialist can keep up with them.

On the other hand, a generalist is more able to fill in if there is a short-term emergency such as one of the team members being very sick or having a family emergency. Generalists often see connections between apparently unrelated subjects and can aid in identifying patterns of software development problems that have been solved before.

In any event, different people, both on and off particular software development teams, have different perspectives on the organization's software, the efficiency of the process of developing the software, and the particular project that they are working on.

A software project manager is interested in how the software development process is working, and whether the system will be produced on time and within budget. Decisions about the viability of the software may be made at a higher level, and many projects that are on time and under budget are canceled because of new releases by the competition, by corporate managers making some of the merged companies' systems redundant, or even by disruptive technologies.

The manager is especially concerned with the group he or she heads. It also is natural for a lower-level manager to be somewhat concerned with the work done by other groups on related projects. After all, an experienced manager is well aware that software projects can be canceled if they are expected to be delivered late or over budget. Higher-level management often has to make unpopular decisions based on whatever information is available. Thus, lack of performance of a team working on one subsystem can affect a project adversely, even if the other subsystems are likely to be produced on time by the teams responsible for their development.

Therefore, a manager will often require his or her team to produce measurements on the team's progress. These measurements could include the number of lines of code written, tested, or reused from other projects; the number and types of errors found in systems; and the number and size of software modules integrated into larger systems.

Software engineers often hate to fill out forms, because they feel that the time needed to do this takes away from their essential activities. It is hoped that understanding a manager's perspective will make this activity more palatable, even for the most harassed programmer.

1.7 SOFTWARE ENGINEERING AS AN ENGINEERING DISCIPLINE

The goal of this section is to convince you that software engineering is an engineering discipline. We will do this by presenting a collection of anecdotes relating to current software development practice. We will describe three classes of software application environments: HTML and other programming for the Internet, applications programs for personal computers, and applications that involve health and safety. In each case, we will illustrate the need for a disciplined approach to software engineering.

The current rush to create both mobile and traditional websites for the Internet, and the shortage of trained personnel to create them, means that jobs are readily available for people with minimal technical knowledge. However, at least for the case of traditional, nonmobile websites, this unusual situation is unlikely to continue far into the future, for several reasons.

Experimentation with leading edge technology may be appealing to many software engineers but will be less so to an organization's chief software officer if the future of the organization depends upon this technology working properly. This later situation is often called "bleeding edge technology." Management is much more likely to support small experimental projects than make major changes to its primary business practice, just to keep up with "leading edge technology."

You should note that when the rush to create the initial web pages is over, the web pages must be tested for accuracy of content, correctness of links, correctness of other programming using the CGI and related approaches, performance under load, appropriateness for both mobile devices and ones with larger screens, general usability, and configuration management. (Configuration management is a very general issue in software engineering. For now, think of it as the systematic approach used to make sure that changes in things such as Internet browser software standards do not make the pages unusable, at the same time making sure that the pages are still accessible by older versions of browsers.)

Here is an example of this problem: I recruited a research team to work on web page design. The team consisted of four freshmen majoring in computer science at Howard University. (The students were chosen because of the clever way they tore off portions of a cardboard pizza box when we ran out of paper plates at the departmental student–faculty party at the beginning of the academic year.) The goal of the project was to develop a website for an online archive of examples of source code with known logical software errors to be used by researchers in software testing to validate their theories and testing tools.

After three months of designing web pages, testing alternative designs, and writing HTML and CGI and Perl scripts, the web page was essentially produced in its final form. The remainder of the effort was devoted to the students learning software testing theory, porting examples written in one programming language to other standard languages, database design, and writing driver programs for procedures that were to be included in the software testing archive. Unfortunately, this website is no longer active.

In short, first-year college students did the web page work in a few months and then turned to standard software engineering activities. Writing web pages for small projects is too easy to serve as the basis for a career without considerable design training. The computer science content involved with the use of HTML appears to be very small. The computer science content is involved with the programming of servers, clients, and scripts in Java and other languages. Since HTML is a language, enhancing the language and developing HTML translators is also a computer science effort. It is in this programming portion that the software engineering activity occurs.

The experience of web design is reminiscent of the situation when personal computers became popular. One negative aspect of the tremendous publicity given to the early successful entrepreneurs in the software industry is the common perception among nonprofessionals that software engineering is primarily an art that is best left to a single talented programmer instead of going back to the origins of computers. The earliest computer programmers were scientists who wrote their own simple algorithms and implemented them on the computers available to them. The earliest computers were programmed by connecting wires in a wiring frame, although this arduous process was soon replaced by writing programs in the binary and octal code that was the machine language of the computer. Machine language programming was quickly replaced by assembly language programming where a small set of mnemonics replaced the numerical codes of machine language. All this encouraged the so-called ace programmer.

Finally, the first attempt at what we now call software engineering emerged in the development of subroutines, or blocks of code that perform one computational task and can be

reused again and again. (Admiral Grace Murray Hopper is generally credited with having written the first reusable subroutine.) Higher-level languages were then developed, with more expressive power and with extensive support for software developments. Indeed, every modern software development environment includes compilers, editors, debuggers, and extensive libraries.

The most modern software development environments include tools to build graphical user interfaces and special software to assist with the software development process itself. The software even has a special name, CASE tools, where the acronym stands for computer-aided software engineering.

Even with the improvement in software development environments and the increasing complexity of modern software, the image of the lone programmer persists. Little harm is done if the lone programmer develops software only for his or her use. The lone programmer also is useful when developing small prototype systems.

For larger software projects, an engineering discipline is necessary. This means that the entire software development process is guided by well-understood, agreed-upon principles, and these principles are based on documented experience. The software development process must:

- Allow for accurate estimates of the time and resources needed for completion of the project
- Have well-understood, quantifiable guidelines for all decisions made
- Have identifiable milestones
- Make efficient use of personnel and other resources
- Be able to deliver the products on time
- Be amenable to quality control

The resulting software product must be of high quality and meet the other goals of software engineering. Above all, as an engineering discipline, both the efficiency of the process and the quality of the product must be measurable.

The issue of quality brings us to the discussion of the other two software application environments that we will consider in this chapter: applications programs for personal computers, and applications that involve health and safety.

There is a tremendous amount of competitive pressure in the area of software application development for personal computers and portable devices. Many software development organizations attempt to release new versions of major applications every six months. For some applications, the interval between releases may be as short as four months. This rapid software development life cycle requires a degree of "concurrent engineering," in which several different life cycle activities are performed simultaneously. The driving factor in this process is the need to provide new features in products.

This efficient production of software comes at a price, however. There is no way that the quality, as measured by such things as ease of installation, interoperability with other applications, or robustness, can be as high as it would be if there were a longer period for testing and design reviews, among other things. Contrary to some views, such software is tested. The decision to release a software product is based on a careful assessment of the number and severity of errors remaining in the software and the relative gain in market share because of new features. Companies such as Microsoft use the technique of software reliability to base decisions on formal data analysis.

It is worthwhile at this point to briefly discuss the software development process used at Microsoft (Cusumano and Selby, 1995, 1997) for many large software products. Most new software development at Microsoft has three phases:

1. In the planning phase, a vision statement is developed for the software based on customer analysis. The specifications are written, together with a list of features. Finally, a project plan is set.
2. In the development phase, development is performed together with testing and evolution of the specifications. This is an iterative process.
3. In the third and final phase, called "stabilization," comprehensive testing occurs both within Microsoft and with selected "beta sites," for testing by external users.

The later portions of this process are often called "sync and stabilize" to reflect that they allow individual software modules to be placed into the overall product. Most systems under development have a daily "build," in which a clean compilation is done. This allows the synchronization of individual modules, regardless of where the modules were created, which is critical in view of the global nature of software development. On a personal note, it was always interesting to see the number of daily builds in large projects such as various releases of the Windows operating system.

The stabilization occurs when the number of changes appearing in the software and the number of errors has been reduced below a level that is considered acceptable. The number of detected software errors for the entire software system is computed each day. A typical, but hypothetical, graph of the daily number of software errors for a month is indicated in Figure 1.10.

Since the number of errors remaining in the software illustrated in Figure 1.10 appears to have stabilized below 7, the software development in this hypothetical example would probably be suspended and the software released, at least if the remaining errors were considered to be small and to be of minimal importance. The implicit assumption is that the software will be sufficiently defect-free once the number of remaining known software errors is below a predefined level and the statistical history suggests that this number will remain low. This assumption is based on the subject known as software reliability, which we will discuss in some detail when we study testing and quality control in Chapter 6.

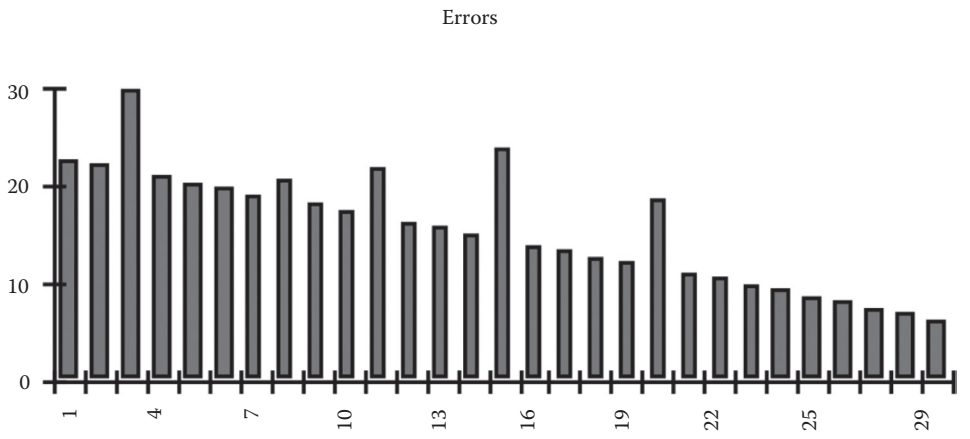

FIGURE 1.10 A typical graph of the daily number of software errors for a month.

You should be aware that each organization has its own standards for acceptable quality of the products it releases. However, no company could stay in business if the marketplace generally considered its products to be of extremely low quality.

Note that these systems are developed over a period of time, however short. Software such as Windows NT, which had a much longer development period, had over a thousand separate compilations (builds) before the latest version was released. Also, most Microsoft products now require large development teams. For example, the development team for Windows 95 had over 200 members (Cusumano and Selby, 1997). Newer versions of operating systems have had similar sizes.

It is clear that Microsoft has a software development process. This process has been tailored to the company's unique corporate culture, understanding of the marketplace, and its financial resources. Satya Nadella, who became Microsoft CEO in early 2014, has changed company practice to have a common software development environment across all platforms, aiming for greater efficiency of development and encouraging reuse.

Applications that involve health and safety generally have a more rigid software development process and a higher standard for logical software errors. The reasons are obvious: potential loss of life in a medical monitoring system, considerable hardship if airplanes are grounded because of an error in the air traffic control system, or major disruption to financial markets if database software used for the U.S. Social Security system fails. Considerable financial loss can occur if, for example, a hotel reservation system or a package tracking system has major failures. People could lose jobs. Clearly, there is an even higher need for a formal software development process. Systems in these areas are often much larger than in the areas previously discussed and, thus, in much more need of a careful software engineering process. The process must be carefully documented to meet the approval of agencies, such as the U.S. Food and Drug Administration for computer software that controls medical products.

We have illustrated that each type of software development area—Internet, personal computer applications, and safety-critical applications—involves some degree of software

engineering, although the processes may be different in different areas. There is one other advantage to treating software engineering as an engineering discipline: avoidance of future risk. Risk analysis is a major part of any engineering discipline.

If risk analysis had been applied to software development when programmers were using two digits instead of four to represent dates, the software development industry would not have faced the "Year 2000 problem," also known as the "Y2K problem," whose cost had been expected to be billions of dollars. (This problem occurred because programmers were under pressure to save space in their code and data sets, and because they did not expect their software to be used twenty or more years after it was developed.) The risk analysis would have cost money, but the resources applied to solve the problem could have been used in much more profitable endeavors with sufficient planning.

It is appropriate at this point to mention a major controversy in the software engineering community: certification of software professionals. In the United States, certification is most likely to take place at the state-government level.

The arguments for certification are that too much software is either produced inefficiently or else is of low quality. The need to ensure the correctness of safety-critical software that affects human life is adding to the pressure in favor of certification. Most engineering disciplines require practicing professionals, or at least senior professionals in each organization, to pass examinations and be licensed, leading to the title "Professional Engineer." For example, a mechanical or civil engineer with this certification is expected to know if a particular type and size of bolt assembly component used in construction has sufficient strength for a specified application.

The primary argument against certification of software professionals is the perception of the lack of a common core of engineering knowledge that is relevant to software development. Many opponents of certification of software professionals believe that the field is too new, with few software engineering practices being based on well-defined principles. Certifying software components for reuse in applications other than the one for which it was created is a major concern.

The Association for Computing Machinery (ACM) has officially opposed licensing for software engineers, whereas the IEEE has supported it. I will not take a position on this issue, although I expect a consensus to arise in the next few years. You should note, however, that the number of undergraduate computer science programs accredited by ABET is increasing rapidly. ABET was originally an acronym for the Accreditation Board for Engineering Technology, but now only the term ABET is in common use. ABET-accredited computer science programs are specifically evaluated by the Computing Science Commission (CAC), a part of ABET, under the auspices of CSAB, which was formerly known as the Computer Science Accrediting Board. A recent article by Phillip A. Laplante (2014) in *Communications of the ACM* provides an interesting perspective on licensing of software professionals (Laplante, 2014).

1.8 SOME TECHNIQUES OF SOFTWARE ENGINEERING

Efficiency is one of the goals of software engineering, both in the efficiency of the development process and the run-time efficiency of the resulting software system. We will not

discuss run-time efficiency of the software in this book, other than to note that it is often studied under the topics "analysis of algorithms" or "performance evaluation." Developing software efficiently often involves systematic efforts to reuse existing software, as we will see later in this section.

The other goals of software engineering that were presented in Section 1.3 are also important. However, focusing on efficiency of both the software product and the process used to develop the software can clarify the engineering perspective of the discipline known as "software engineering."

Following are five major techniques that are typically part of good software engineering practice and which may be familiar to you:

1. There must be a systematic treatment of software reuse.
2. There must be a systematic use of metrics to assess and improve both software quality and process efficiency.
3. There must be a systematic use of CASE tools that are appropriate for the project's development environment.
4. There must be a systematic use of valid software cost estimation models to assess the resources needed for a project and to allocate these resources efficiently.
5. There must be a systematic use of reviews and inspections to ensure that both the project's software and the process used to develop that software produce a high-quality solution to the problem that the software is supposed to solve.

Each of these techniques is briefly discussed next in a separate section. Note that these techniques are intended to support the efficient performance of the software engineering tasks listed in Section 1.4. Even if you are a lone programmer working on a small project, you need to work as efficiently as possible, so these guidelines apply to you, too.

1.8.1 Reuse

Software reuse refers to the development of software systems that use previously written component parts that have been tested and have well-defined, standard interfaces. Software reuse is a primary key to major improvements in software productivity. The productivity of programmers has remained amazingly constant over the last thirty-plus years, with the average programmer producing somewhere in the range of eight to twelve correct, fully documented and tested lines of code per day (Brooks, 1975). At first glance, this number seems appallingly low, but you should realize that this takes into account all meetings, design reviews, documentation checks, code walkthroughs, system tests, training classes, quality control, and so on. Keep in mind that although productivity in terms of lines of code has changed little, productivity in terms of problems solved has increased greatly due to the power of APIs and the ability to leverage efforts by reusing code and other artifacts as much as possible.

Software reuse has several different meanings in the software engineering community. Different individuals have viewpoints that depend upon their responsibilities. For example, a high-level software manager might view software reuse as a technique for improving the overall productivity and quality of his or her organization. As such, the focus would be on costs and benefits of organizational reuse plans and on schemes for implementing company-wide schemes.

Software developers who use reusable code written by others probably view software reuse as the efficient use of a collection of available assets. For these developers, who are also consumers of existing software, software reuse is considered a positive goal since it can improve productivity. A project manager for such development would probably view reuse as useful if the appropriate reused software were easy to obtain and were of high quality. The manager would have a different view if he or she were forced to use poorly tested code that caused many problems in system integration and maintenance because of its lack of modularity or adherence to standards.

On the other hand, developers who are producers of reusable code for use for their own projects and for reuse by others might view reuse as a drain on their limited resources. This is especially true if they are required to provide additional quality in their products or to collect and analyze additional metrics, all just to make source code more reusable by others.

A reuse librarian, who is responsible for operating and maintaining a library of reusable components, would have a different view. Each new reuse library component would have to be subjected to configuration management. (Configuration management is the systematic storage of both a newly edited document as well as the ability to revert back to any previous version. It is an essential part of every common software engineering process and is included in nearly all tools that support software engineering.) Configuration management is necessary if a file is being changed over a long period of time even if it is just being edited by a single person, and even more so if the file is being used and edited by several people. Some degree of cataloging would be necessary for future access. Software reuse makes the job of a reuse librarian necessary.

In general, the term *software reuse* refers to a situation in which some software is used in more than one project. Here "software" is defined loosely as one or more items that are considered part of an organization's standard software engineering process that produces some product. Thus, "software" could refer either to source code or to other products of the software life cycle, such as requirements, designs, test plans, test suites, or documentation. The term *software artifact* is frequently used in this context.

In informal terms, reuse can be defined as using what already exists to achieve what is desired. Reuse can be achieved with no special language, paradigm, library, operating system, or techniques. It has been practiced for many years in many different contexts. In the vast majority of projects, much of the necessary software has been already developed, even if not in-house.

Reusability is widely believed to be a key to improving software development productivity and quality. By reusing high-quality software components, software developers can

simplify the product and make it more reliable. Frequently, fewer total subsystems are used and less time is spent on organizing the subsystems.

If a software system is large enough, programmers often work on it during the day and wait until the next day to check the correctness of their code by running it. Many systems are so large that they are compiled only once a day, to reduce computer load and to provide consistency.

The major improvement in computer productivity is due to the improvement in programming languages. A single line of code in a modern programming language—say, C++ or Java—may express the same computational information that requires many lines of assembly language. A spreadsheet language, such as Microsoft Excel or Numbers, is even more powerful than most modern, general-purpose programming languages for special applications. Table 1.1 shows the effects of the choice of programming languages on the relative number of statements needed to perform a typical computation. Some related data can be found in Capers Jones's book (Jones, 1993).

Table 1.2 presents another view of how the expressive power of programming languages can be compared: the cost of a delivered line of code.

Note that there is a much more efficient way to improve software productivity: reuse code. The efficient reuse of source code (and other software engineering items such as

TABLE 1.1 Comparison of the Expressive Quality of Several Program Languages

Language	Lines of Code
Assembly	320
Ada	71
C	150
Smalltalk	21
COBOL	106
Spreadsheet Languages	6

Source: Reifer, D. J., *Crosstalk: The Journal of Defense Software Engineering*, vol. 9, no. 7, 28–30, July 1996.

Note: The term "spreadsheet language" refers to commonly available, higher-level programming applications such as spreadsheets or databases.

TABLE 1.2 Comparison of the Dollar Cost of Delivered Source Line of Code for Several Program Languages in Several Different Application Domains

Application Domain	Ada83	C	C++	3GL	Domain Norm
Commercial command and control	50	40	35	50	45
Military command and control	75	75	70	100	80
Commercial products	35	25	30	40	40
Commercial telecommunications	55	40	45	50	50
Military telecommunications	60	50	50	90	75

Source: Reifer, D. J., *Crosstalk: The Journal of Defense Software Engineering*, vol. 9, no. 7, 28–30, July 1996.

requirements, designs, test cases, documentation, and so on) can greatly improve the way in which software is produced.

At first glance, it might be surprising to a beginning software engineer that software reuse is relevant in an age of network-centric computing with many applications running in the cloud and many other applications running on mobile devices. However, a moment's thought leads to the observation that a cloud computing environment, with its hiding of details of large-scale subsystems, is an excellent place for reused software.

1.8.2 Metrics

Any engineering discipline has a quantifiable basis. Software engineering is no exception. Here, the relevant measurements include the size of a software system, the quality of a system, the system's performance, and its cost. The most commonly used measurement of the size of a system is the number of lines of source code, because this measurement is the easiest to explain to nonprogrammers. We will meet other measurements of software system size in this book, including measurements of the size of a set of requirements.

There are two measurements of software system quality that are in common use. The first is the number of defects, or deviations from the software's specifications. The quality of a system is often measured as the total number of defects, or the "defect ratio," which is the number of defects per thousand lines of code. The terms *fault* and *failure* are sometimes used in the software engineering literature. Unfortunately, the terms *defect*, *error*, *fault*, and *failure* do not always mean the same thing to different people. The term *bug* has been in common use since the time when Grace Murray Hopper recounted a story of an error in the Mark II computer traced to a moth trapped in a relay. The "Software Engineering Body of Knowledge" uses the term *bug* (SWEBOK, 2013). However, we will follow the recommendations of the IEEE (1988) and use the following terminology: A software *fault* is a deviation, however small, from the requirements of a system; and a software *failure* is an inability of a system to perform its essential duties.

A second measurement of software system quality is the number of faults per thousand hours of operation. This measurement may be more meaningful than the number of faults per thousand lines of code in practice.

One other much less common measurement is an assessment of the quality of the user interface of a software system. Although the quality of a user interface is hard to measure, good user interfaces are often the keys to the success or failure of a software application in a crowded market. We will discuss user interfaces extensively in Chapter 4 as part of our study of software design.

The term *metrics* is commonly used in the software engineering literature to describe those aspects of software that we wish to measure. There is a huge number of metrics that are currently in use by software organizations. Many of the metrics are collected, but the resulting values are not frequently used as a guide to improving an organization's software development process or toward improving the quality of its products. The use of metrics is absolutely essential for a systematic process of software engineering. Without metrics, there is no way to evaluate the status of software development, assess the quality of the result, or track the cost of development.

Metrics can be collected and analyzed at almost any phase of a software life cycle, regardless of the software development life cycle used. For example, a project could measure the number of requirements the software system is intended to satisfy, or, more likely, the number of critical requirements could be measured. During testing, the number of errors found could be measured, with the goal of seeing how this number changed over time. If more errors appear to grow near the scheduled date of release, there is clearly a problem. Ideally, other metrics would have pointed out these problems earlier in the development cycle.

Our view is that metrics should be collected systematically, but sparingly. Our approach is to use the goals, questions, metrics (GQM) paradigm of Basili and Rombach (Basili and Rombach, 1988). The GQM paradigm consists of three things: goals of the process and product, questions we wish to ask about the process and product, and methods of measuring the answers to these questions. The GQM paradigm suggests a systematic answer to the question "which metrics should we collect?"

Typical goals include:

- Quantify software-related costs
- Characterize software quality
- Characterize the languages used
- Characterize software volatility (volatility is the number of changes to the software component per unit time)

There are clearly many questions that can be asked about progress toward these goals. Typical questions include the following (we will only list two questions per goal):

- What are the costs per project? (This question is used for costs.)
- What are the costs for each life cycle activity? (This question is used for costs.)
- How many software defects are there? (This question is used for quality.)
- Is any portion of the software more defect-prone than others? (This question is used for quality.)
- What programming languages are used? (This question is used for languages.)
- What object-oriented programming language features are used? (This question is used for languages.)
- How many changes are made to requirements? (This question is used for volatility.)
- How many changes are made to source code during development? (This question is used for volatility.)

The clarity of these questions makes the choice of metrics easy in many cases. We note that there are several hidden issues that make metrics data collection complicated.

For example, if there is no tracking mechanism to determine the source of a software error, then it will be difficult to determine if some portion of the software is more defect-prone than others. However, if you believe that this question must be answered in order to meet your stated goals, then you must either collect the data (which will certainly cost money, time, and other resources) or else change your goals for information gathering.

There are a few essentials for collection and analysis of metrics data in support of a systematic process of software reuse:

- Metrics should be collected on the same basis as is typical for the organization, with extensions to be able to record and analyze reuse productivity and cost data.
- Predictive models should use the reuse data, and the observed resource and quality metrics must be compared with the ones that were estimated.
- Metrics that measure quality of the product, such as errors per 1,000 source lines of code, perceived readability of source code, and simplicity of control flow, should be computed for each module.
- Metrics that measure the process, such as resources expended, percentage of cost savings, and customer satisfaction, should be computed for each module and used as part of an assessment of reuse effectiveness.

1.8.3 Computer-Aided Software Engineering (CASE)

There are many vendors of CASE tools, far too many to discuss here in any detail. There are even more prototype CASE tools currently under development by academic and other research institutions. CASE tools can be complex to use and require a considerable amount of effort to develop. Commercial CASE tools are often expensive and require extensive training. I used one commercial product that cost $3,600, just for a single seat license at my university as part of a research project, with an annual software maintenance cost of nearly $500 per year. A second commercial product that I use often is much cheaper. (Many CASE tool companies give great discounts to academic institutions, thereby encouraging the development of a knowledgeable workforce and obtaining a tax deduction.) Why are these products so popular?

There are several reasons for this popularity. The most important is that software development is extremely expensive for most organizations and anything that improves efficiency and quality of the software process at reasonable cost is welcome. For many organizations, the cost of software development is so high that nearly any tool that improves productivity, however slight, is worth its price.

Organizations often wish to provide development environments that are considered cutting edge if they are to be competitive in attracting and retaining good personnel. An examination of the larger display ads for organizations hiring software engineers indicates the emphasis on providing good development environments.

Other reasons for the popularity of CASE tools include the need for a common, consistent view of artifacts at several different phases of the organization's software life cycle. Indeed, the requirements, design, and source code implementation of a project can be much more consistent if they are all stored in a common repository, which checks for such consistency. For the most part, this common repository of several different types of software artifacts is present only in expensive, high-end CASE tools.

There is a wide range of CASE tools. Some simple tools merely aid in the production of good quality diagrams to describe the flow of control of a software system or the flow of data through the software during its execution. Examples of control flow diagrams and data flow diagrams are shown in Figures 1.11 and 1.12, respectively.

The earliest popular graphical designs were called "flowcharts" and were control-flow oriented. The term *control flow* is a method of describing a system by means of the major blocks of code that control its operation. The nodes of a control flow graph are represented by boxes whose shape and orientation provides additional information about the program. For example, a rectangular box with horizontal and vertical sides means that a computational process occurs at this step in the program. Such boxes are often called "action boxes." A diamond-shaped box, with its sides at 45-degree angles with respect to the horizontal direction, is known as a "decision box." A decision box represents a branch in the control flow of a program. Other symbols are used to represent commonly occurring situations in program behavior.

Control flow diagrams indicate the structure of a program's control at the expense of ignoring the movement and transformation of data. This is not surprising, since control

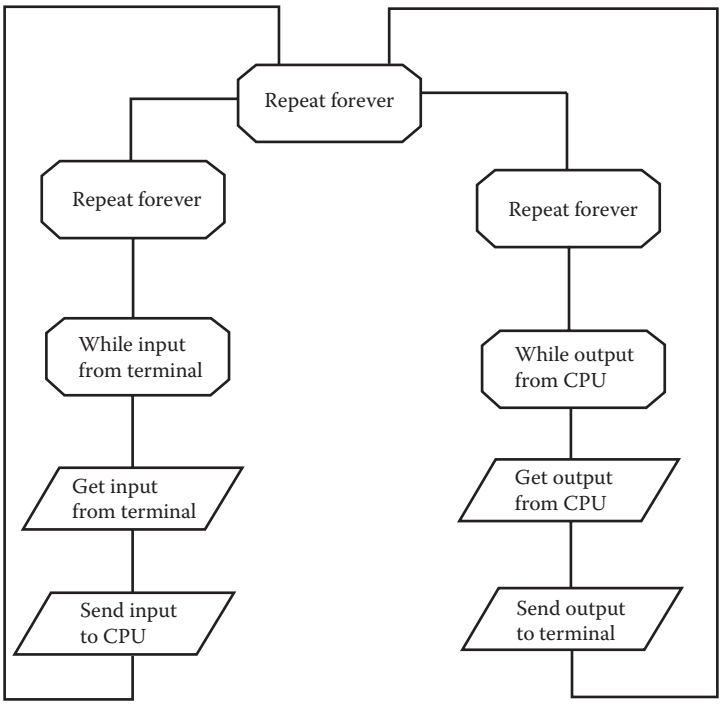

FIGURE 1.11 An example of a control flow diagram.

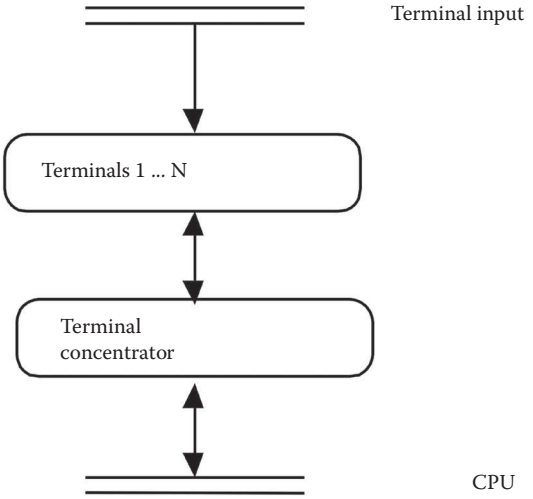

FIGURE 1.12 An example of a high-level data flow diagram with a data source and a data sink.

flow diagrams were developed initially for use in describing scientific programming applications. Programs whose primary function was to manage data, such as payroll applications, were often represented in a different way, using "data flow diagrams."

There is an often-voiced opinion that flowcharts are completely obsolete in modern software. This opinion is not completely correct. A United States patent named "Automatically Enabling Private Browsing of a Web Page, and Applications Thereof," invented by Michael David Smith and assigned to Google, uses diagrams that are related to flowcharts. One of the diagrams that comprises the application submitted to obtain this patent, number US 8,935,798 B1 and issued on January 13, 2015, is included in Appendix A.

Data flow representations of systems were developed somewhat later than control flow descriptions. The books by Yourdon (Yourdon, 1988) and DeMarco (DeMarco, 1979) are probably the most accessible basic sources for information on data flow design. Most software engineering books contain examples of the use of data flow diagrams in the design of software systems.

Since different data can move along different paths in the program, it is traditional for data flow design descriptions to include the name of the data along the arrows indicating the direction of data movement.

Data flow designs also depend on particular notations to represent different aspects of a system. Here, the arrows indicate a data movement. There are different notations used for different types of data treatment. For example, a node of the graph that represents a transformation of input data into output data according to some set of rules might be represented by a rectangular box in a data flow diagram.

A source of an input data stream, such as interactive terminal input, would be represented by another notation, indicating that it is a "data source." On the other hand, a repository from which data can never be recalled, such as a terminal screen, is described by another symbol, indicating that this is a "data sink."

You might ask at this point how these notations are used to describe larger systems. Certainly the systems you will create as practicing software engineers are much too large to be described on a single page. A flowchart will usually connect to another flowchart by having the action or decision box appear both on the initial page where it occurs and on the additional page where the box is needed.

Typical data flow descriptions of systems use several diagrams at different "levels." Each level of a data flow diagram represents a more detailed view of a portion of the system at a previously described, higher level.

Different data flow diagrams are used to reflect different amounts of detail. For example, a "level 0 data flow diagram" provides insight only into the highest level of the system. Each item in a data flow diagram that reflects a transformation of data or a transaction between system components can be expanded into additional data flow diagrams. Each of these data flow diagrams can, in turn, be expanded into data flow diagrams at different levels until the system description is considered sufficiently clear.

Of course, there are other ways to model the behavior of software systems. Many compilers for object-oriented languages such as C++, Objective C, Eiffel, or Java are part of software development that support object modeling. These environments generally allow the user to develop an object model of a system, with the major structure of the objects specified as to the public and private data areas and the operations that can be performed on instances of these objects.

The more elaborate environments include an editor, a drawing tool, and a windowing system that allows the user to view his or her software from many different perspectives. Such advanced environments certainly qualify as CASE tools. An example of a typical advanced software development environment is given in Figure 1.13.

The more advanced software development environments go one step further. An object model created by a user has a description of the public and private data used by an object and the operations that can be performed on the object. The additional step is to allow the generation of frameworks for source code by automatically transferring the information for each object that is stored in the model in a diagram to source code files that describe the object's structure and interfaces. This generated source code can be extended to the creation of complete programs. This code generation is an excellent example of a common CASE tool capability.

Many compilation environments include language-sensitive editors that encourage the use of correct language syntax, debuggers to help fix errors in source code, and profilers, which help improve the efficiency of programs by allowing the user to determine the routines that take the largest portion of the program's execution time. These editors, debuggers, and profilers are all CASE tools.

Other tools allow the management of libraries, where several people are working on source code development at the same time. These tools ensure that only one person is working on any particular source code file at the same time. If a software engineer is editing a source code file, any request from others to edit the same document is denied. As discussed in Section 1.8.1, this technique is called configuration management. It is an essential part of the software engineering process and is almost always supported by CASE tools.

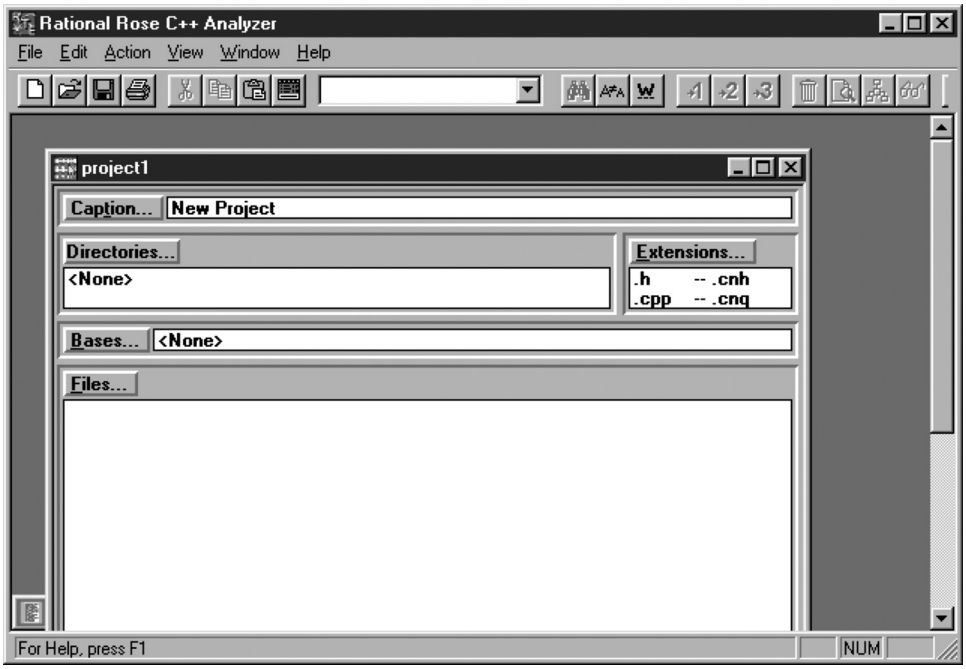

FIGURE 1.13 An example of a window-based software development environment for an object-oriented system. (From Rational Rose Software opening screen, Rational Software Corporation. Rational is now a subsidiary of IBM.)

Clearly, CASE tools are pervasive in software engineering. Software development tools have become so popular and inexpensive that many software engineers do not refer to a development tool as a CASE tool unless it is elaborate, expensive, and supports the entire life cycle. I choose to use the term more broadly, in the spirit of the examples described in this section.

1.8.4 Cost Estimation

Software cost estimation is both an art and a unique subfield of software engineering. It is far too complex to discuss in detail in any general-purpose book on software engineering. We will be content to provide simple rules of thumb and first approximations to determine the cost-benefit ratio of different system design and programming alternatives. The classic book by Boehm (Boehm, 1981) is still the best introductory reference to software cost estimation.

The basic idea is to estimate the cost of a software project by comparison with other, similar projects. This in turn requires both a set of "similar" software projects and a method of determining which projects are "similar" to the given one.

In order to determine the cost of projects, a baseline must be established. By the term *baseline*, we mean a set of previous projects for which both measurements of size and cost are available. The information obtained from these previously completed projects is stored in a database for future analysis. The baseline information should include the application domain; the size of the project, according to commonly agreed-upon measurements;

any special system constraints, such as the system having a real-time response to certain inputs; unusual features of the project, such as being the first time that a new technology, programming language, or software development environment is used; any interoperable software systems that may impact the cost of the software; and, of course, the cost of the software itself. In many organizations, the baseline information may be broken down into the cost of different subsystems.

The determination of "similarity" or "sameness" is made by examining the characteristics of projects in the baseline database and selecting the projects for which information is deemed to be relevant. The effective determination of "similarity" is largely a matter of experience.

Ideally, the cost estimation will be developed in the form of a cost for the system (and each subsystem for which cost estimates are to be made) and an expected range in which the costs are likely to be for the system whose cost is being estimated.

1.8.5 Reviews and Inspections

Different organizations have developed different approaches to ensure that software projects produce useful, correct software. Some techniques appear to work better in certain development environments than others and, thus, there is difficulty in simply adapting one successful approach to a totally different environment.

However, there is one thing that has been shown to be highly successful in every organization in which it is carried out properly: the design review. Reviews should be held at each major milestone of a project, including requirements, design, source code, and test plans. For a software life cycle that uses an iterative process, a review should be held at each iteration of the creation of major items within the software life cycle.

The basic idea is simple: each artifact produced for a specific milestone during development must be subjected to a formal review in which all relevant documents are provided; a carefully designed set of questions must be answered; and the results of the review must be written down and disseminated to all concerned parties.

The following will describe several kinds of reviews for requirements, design, source code, and test plans. In the sense in which they are used here, the two terms *review* and *inspection* are synonymous. In the general literature, the term *inspection* may be used in this sense or may instead refer to a procedure for reading the relevant documents.

There is one other point that should be made about reviews and inspections. This book emphasizes systematic methods that attempt to determine problems in requirements and design well before the software is reduced to code. This emphasis is consistent with experiences across a wide variety of software projects in nearly every application area.

The open source approach of the development of the Linux operating system avoids reviews of requirements and designs, and instead relies on the efforts of a large number of programmers who review source code posted on line. Thus each source code module is intensively reviewed, often by hundreds of programmers. Modifications to the source code are controlled by a much smaller team of experts. Of course, there is also configuration management and testing, just in different order. Consult the article by McConnell (McConnell, 1999) for an overview of the open source approach.

1.9 STANDARDS COMMONLY USED FOR SOFTWARE DEVELOPMENT PROCESSES

Many organizations have very specific software development processes in place. These processes are motivated by both organizational experience and the pressures brought on by outside customers from both government and industry. The most common formalized software development processes in the United States are

- The Capability Maturity Model (CMM) from the Software Engineering Institute (SEI) at Carnegie Mellon University. (Technically, this is not a process but an evaluation of how systematic an organization's software development process is and a guideline for assessing the possibility of process improvement.) Information on the CMM is available from the website http://www.sei.cmu.edu/cmmi/.
- The CMM model has been superseded by the Capability Maturity Model Integration (CMMI). The primary website for this model has been moved from its original home at the Software Engineering Institute (http://www.sei.cmu.edu/cmmi/) to the 100 percent-owned subsidiary of Carnegie Mellon University, http://cmmiinstitute.com/. This newer website has several examples of how appropriate measurement of software development processes can be used to improve quality and reduce costs.
- The Process Improvement Paradigm (PIP) from the Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center. (As with the CMM, technically, this is not a process but an evaluation of how systematic a software development process is and a guideline for assessing the possibility of process improvement.)
- The Department of Defense standard MIL-STD 2167A.
- The Department of Defense standard MIL-STD 1574A.
- The Department of Defense standard MIL-STD 882C.
- The Electronic Industries Association (EIA) SEB-6-A.

Some international standard software development processes include the following:

- The European ESPRIT project.
- The International Standards Organization (ISO) standard ISO 9001.
- United Kingdom MOD 0055.

Each of these software process models has a heavy emphasis on the collection and use of software metrics to guide the software development process. The models from the SEI and the SEL stress the development of baseline information to measure current software development practice. These two models can be used with classical waterfall, rapid prototyping, and spiral methodologies of software development. The more rigid standard DOD 2167A is geared primarily toward the waterfall approach, but it, too, is evolving to consider other software development practices.

These are probably the most common nonproprietary standardized software development processes being used in the United States. Note that all these process models emphasize the importance of software reuse because of its potential for cost savings and quality improvement.

Many private organizations use models based on one of these standards. For example, the rating of a software development organization's practice from 1 ("chaotic") to 5 ("optimizing") on the CMM scale is now a factor in the awarding of government contracts in many application domains. The rating is done independently. The reason for rating the development organization's practices is to ensure a high-quality product that is delivered on time and within budget. The details of the CMM scale are given in Table 1.3.

As another example of the importance of software development processes, Bell Communications Research (1989) had used a process assessment tool to evaluate the development procedure of its software subcontractors for many years. An assessment of the subcontractor's software development process was an essential ingredient in the decision to award a contract.

Both Allied Signal Technical Corporation and Computer Sciences Corporation, among others, have an elaborate written manual describing their software development processes. The reference (Computer Sciences Corporation, 1992) is typical. You should expect that

TABLE 1.3 Description of the Levels of the Capability Maturity Model (CMM) Developed by the Software Engineering Institute (SEI)

Level 5 (Optimizing Process Level)
The major characteristic of this level is continuous improvement. Specifically, the software development organization has quantitative feedback systems in place to identify process weaknesses and strengthen them proactively. Project teams analyze defects to determine their causes; software processes are evaluated and updated to prevent known types of defects from recurring.
Level 4 (Managed Process Level)
The major characteristic of this level is predictability. Specifically, detailed software processes and product quality metrics are used to establish the quantitative evaluation foundation. Meaningful variations in process performance can be distinguished from random noise, and trends in process and product qualities can be predicted.
Level 3 (Defined Process Level)
The major characteristic of this level is a standard and consistent process. Specifically, processes for management and engineering are documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing software.
Level 2 (Repeatable Process Level)
The major characteristic of this level is that previous experience provides intuition that guides software development. Specifically, basic management processes are established to track cost, schedule, and functionality. Planning and managing new products is based on experience with similar projects.
Level 1 (Initial Process Level)
The major characteristics of this level are that the software development process is largely ad hoc. Specifically, few processes are defined, and success depends more on individual heroic efforts than on following a process and using a synergistic team effort. (The term *chaotic* is often used to describe software development processes with these characteristics.)

Source: Software Engineering Institute.

any large software company that you work for will have its own software procedures and will probably have a standards and practices manual.

There are many other software standards that are still being followed to some extent. A count of standards activities of the IEEE indicates 35 distinct standards in the area of software engineering at the end of 1995. One of the most important IEEE standards is number P610.12(R), which is titled "Standard Glossary for Software Engineering." See also the SWEBOK website.

Any introductory book on the topic of software engineering would be incomplete if it did not mention the cleanroom software development process (Mills et al., 1987). This process was initially developed at IBM by Harlan Mills and his colleagues. The goal of the cleanroom process is to produce software without errors by never introducing errors in the first place. The term *cleanroom* was chosen to evoke images of the clean rooms used for manufacture of silicon wafers or computer chips. The workers in such rooms wear white gowns to reduce the possibility of introducing contaminants into their product. In the cleanroom approach, any errors in software are considered to have arisen from a flawed development process.

The major technique of the cleanroom process is to use mathematical reasoning about correctness of program functions and procedures. The idea is to create the source code with such care that there is no need for testing. Rigorous mathematical reasoning and very formal code reviews are both integral parts of the cleanroom process.

Chapter 6 will indicate that the purpose of software testing is to uncover errors in the software, not to show that the software has no errors. The practitioners of the cleanroom approach believe that their approach may be more conducive to the goal of the software development process: producing error-free software.

We will not discuss the cleanroom approach any further in this book. Instead, our limited emphasis on the role of mathematics in software engineering will be restricted largely to reliability theory and a brief mention of some other formal development methods that we will discuss in Chapters 2 and 3.

There is an approval process used by the Apple App Store before an app is allowed to be sold either for Macintosh computers or other Apple products such as iPads and iPhones. The App Store's Review Guidelines provide rules and examples for such things as user interface design, functionality, content, and the use of specific technologies. There are also guidelines for the use of push technology and for the secure handling of data. More information on these particular guidelines can be found at the site https://developer.apple.com/app-store/review/. (Note: It is good to be an Apple developer if you can get to tell your grandson that you can get discounts on several different Apple products.)

Not surprisingly, other app stores are beginning to have similar review processes.

1.10 ORGANIZATION OF THE BOOK

Each of the major phases of the software development life cycle will be covered in detail in a separate chapter after we discuss project management in Chapter 2. Beginning with Chapter 3, each of the chapters will consider a relatively large project that will be continued throughout the book.

The purpose of considering the same project throughout the book is to eliminate the effects of different decisions for the requirements, design, and coding of software. Some

of the source code and documentation will be given explicitly in the book itself. However, most of the rest will not be made available, allowing instructors to have more flexibility in tailoring the assignments especially to individual groups within their classes.

A continuing case study of an agile software development project will also be given throughout the book to provide a complete picture of how a successful agile project can work.

The order of Chapters 3 through 8 is essentially the same as in the classical waterfall software development model. This order is artificial (something had to be presented first!) and the chapters can largely be read in any order. Some instructors may choose to cover several development methodologies rather quickly in order to focus on the development methodology that fits best with the educational objectives of the course.

The book will work best when coordinated with the creation of team software projects that reuse existing code, but require something more elaborate than a simple cut-and-paste of source code.

As you will learn, there are many nonprogramming activities necessary in software engineering. Some of these activities require cost estimation, or data collection and analyses. Spreadsheet forms have been included to illustrate typical industry practices and to provide you with experience.

Each of Chapters 3 through 8 will include one or more sections that provide typical views of software managers on the relevant technical activities. The purpose of the managerial view is to provide you with additional perspective on software engineering, not to prepare you to be a manager. In most organizations, a considerable amount of varied project experience is essential before a person is given a managerial position at any level.

Chapter 9 discusses how to read the software engineering literature and provides a large set of important references.

The three appendices on software patents, command-line arguments, and flowcharts can be read if necessary.

SUMMARY

Software engineering is the application of good engineering practice to produce high-quality software in an efficient manner. Good software engineering practice can reduce the number of problems that occur in software development. The goal of software engineering is to develop software that is efficient, reliable, usable, modifiable, portable, testable, reusable, easy to maintain, and can interact properly with other systems. Most important, the software should be correct in the sense of meeting both its specifications and the true wishes of the user.

The goals of software engineering include efficiency, reliability, usability, modifiability, portability, testability, reusability, maintainability, interoperability, and correctness. In support of these goals, software engineering involves many activities:

- Analysis
- Requirements

- Design
- Coding
- Testing and integration
- Installation and delivery
- Documentation
- Maintenance
- Quality assurance
- Training

Project management is essential to coordinate all these activities.

There are several common models of the software development process. In the classical waterfall process model, the requirements, design, coding, testing and integration, delivery and installation, and maintenance steps follow in sequence, with feedback only to the previous step. In the rapid prototyping and spiral development models, the software is developed iteratively, with customer or user feedback given on intermediate systems before a final system is produced.

A disciplined software engineering approach is necessary, regardless of the application domain: development for the Internet, personal computer applications, and health- or safety-critical applications. The risk analysis that is typically part of any engineering activity might have prevented the Year 2000 problem from occurring.

The open source model involves a small group of experts who act as gatekeepers evaluating the submissions from a vast number of individual software engineers to help solve problems set by these expert gatekeepers. All software submitted is done so under the understanding that others are free to use or modify it as long as they do not charge for it; that is, the software must remain free, although charging for installation, setup, and consulting are allowed.

A major goal of the agile programming model is to provide fast solutions to problems where the team has significant knowledge of the particular application domain and there are a lot of high-quality software components that are available for reuse to help solve these problems.

The use of high-level programming languages can increase programmer productivity. However, reusing high-quality software components can have a greater effect.

KEYWORDS AND PHRASES

Software engineering, software reuse, classical waterfall software development model, rapid prototyping model, spiral model, agile software development, agile methods, open source software development, market-driven software development, software evolution, reengineering, CASE tools, metrics

FURTHER READING

There are many excellent books on software engineering, including Pfleeger and Atlee (Pfleeger and Atlee, 2010), Schach (Schach, 2011) Sommerville (Sommerville, 2012), and Pressman and Maxim (Pressman and Maxim, 2015). Some interesting older general software engineering books are by Ghezzi, Mandrioli, and Jayazerri (Ghezzi, Jayazerri, and Mandrioli, 2002), Jalote (Jalote, 1991), and Shooman (Shooman, 1983). The book *Software Development: An Open Source Approach* (Tucker and de Silva, 2011) is devoted to open source software development. The classic book by Boehm (Boehm, 1981) provides a good overview of software engineering economics. Fenton and Pfleeger (Fenton and Pfleeger, 1996) and Fenton and Bieman in a later edition (Fenton and Bieman, 2014) provide a detailed, rigorous description of software metrics. Beizer (Beizer, 1983, 1990) Howden (Howden, 1987), and Myers (Myers, 1979) provide excellent introductions to software testing that are still relevant today. An excellent introduction to the cleanroom process can be found in the article "Cleanroom Software Engineering" (Mills et al., 1987).

A large-scale effort by the software engineering community has created the "Software Engineering Body of Knowledge," commonly known by the acronym SWEBOK. This effort can be found in a collection of online PDF documents (SWEBOK, 2013).

The helpful 2014 Research Edition of the Software Almanac from Software Quality Management is available for download at qsm.com.

The classic 1978 paper by Ritchie and Thompson (Ritchie and Thompson, 1978) provides both an overview of the extremely successful UNIX operating system and an insight into the design decisions that were used. The Linux operating system is a modern variant of UNIX.

Information on computer languages can be found in many places. The Ada language is described in Ada (1983, 1995) and Ichbiah (1986). The rationale for the original description of C++ is given by Stroustrup (Stroustrup, 1994), and a detailed language manual is provided in the book by Ellis and Stroustrup (1990). Kernighan and Ritchie provide an original description of C (Kernighan and Ritchie, 1982), with a second edition (Kernighan and Ritchie, 1988) describing the ANSI C features. There are many other excellent books on various computer languages.

Nielsen (1994) and Shneiderman (1980) provide excellent overviews of human–computer interaction. A brief introduction to the role of color in user interfaces can be found in the article by Wright, Mosser-Wooley, and Wooley (1993). The older paper by Miller (1968) provides good empirical justification for limiting the number of choices available to system users. It is especially relevant to a designer of a computer system's menu structure.

Laplante (2014) provides an interesting perspective on an evolving position on requiring licensing for software engineers.

The article by Cusumano and Selby (1995) provides an excellent overview of some software development practices at Microsoft.

Information on the Capability Maturity Model (CMM) is available from the website http://www.sei.cmu.edu/cmmi/.

More information on the Capability Maturity Model Integration (CMMI) is available from the website http://cmmiinstitute.com/.

The paper by Basili and Rombach (1988) is a good introduction to the GQM paradigm. The book by Leach (1997) is a good introduction to software reuse.

EXERCISES

1. Name the major types of software development life cycles. How do they differ?
2. Name the activities that are common to all software development life cycles.
3. What are the goals of software engineering?
4. Explain the GQM paradigm.
5. Consider the largest software project you ever completed as part of a class assignment. How large was the project? How long did it take you to develop the system? Did it work perfectly, did it work most of the time, or did it fail the simplest test cases?
6. Repeat Exercise 5 for software that you wrote as part of a job working in industry or government.
7. Examine the classified advertisements of your college or university placement office for listings of jobs in the computer field. Classify each of the jobs as being for analysis, requirements, design, coding, testing and integration, installation and delivery, documentation, maintenance, quality assurance, or training. What, if anything, is said about open source software? What about agile programming?
8. If you have a part-time (or full-time) job in the computer field, ask to see your employer's software development manual. Determine which software development process standard is followed by your employer.
9. Several software development process standards were mentioned in this chapter. Find one or more of these in the library or on the Internet. Examine the details of the standard. Can you explain why they are necessary?
10. Examine a relatively large software system that you did not develop yourself and for which source code is available. (The GNU software tools from the Free Software Foundation are an excellent source for this question.) Can you tell anything about the several software development process standards used in this software from an examination of the source code? Why or why not?
11. We discussed some issues about the role of networking in the future development of word processors. There are other issues, such as the responsibility of system administrators responsible for software updates. Discuss the effect of the four options given in Section 1.1 from the perspective of system administrators. You may find it helpful to read the discussion on how an index might be implemented on a multicore

processor in Andrew Tanenbaum's recent book *Modern Operating Systems*, 4th edition (Tanenbaum, 2014).

12. Research Lockheed Martin's Skunk Works project, also known as "Advanced Development Programs," to obtain a description of its very successful quick-and-dirty development process. Compare this to the goals of the "Agile Manifesto."

CHAPTER 2

Project Management

In this chapter, we will discuss some of the issues involved with project management. As was pointed out in Chapter 1, most of the discussion of this topic will appear quite remote from the experience of a beginning software engineer. It is difficult to communicate the complexity of software project management to a person who has not been part of a multiperson software project in government or industry. Nevertheless, it is useful for every software engineering student to have at least an introduction to those software engineering activities that are typically associated with group projects. Therefore, we will present a high-level view of project management in this chapter. The intention is to introduce the subject, not to create instant experts. Keep in mind the point that most modern software development is done by teams.

Many of the most successful software companies, especially the ones that have achieved almost overnight successes, appear to take considerable risks in the sense that they produce software that performs a task or solves a problem that might not have been considered for a computer solution before the software was created. Software intended for word processing, spreadsheet analysis, or the Internet itself, with its collection of browsers and search engines, were clearly revolutionary breakthroughs. As such, they were considered risky investments until they achieved some success. All so-called killer apps have this feature. Risk is inherent in most software engineering endeavors.

Keep in mind, however, that an organization whose existence depends upon the continued success of its software products must take a systematic approach to management of its software projects. This approach must be more systematic than would be necessary for a student who is working on a simple class project that may require only one person or, at most, a small group of classmates. It is generally more systematic than what might be appropriate for a single programmer creating a small, stand-alone app. Software project management should attempt to reduce the risk in the way that the software project follows a schedule and is developed efficiently according to reasonable software engineering standards for quality and productivity.

The primary goal of this chapter is to introduce project management issues in sufficient detail so that you can understand the types of organizational cultures you might meet when