PART

1

Introduction to Software Engineering

The first nine chapters of this book play a dual role. They introduce the reader to the software process, and they also provide an overview of the book. The software process is the way we produce software. It starts with concept exploration and ends when the product is finally decommissioned. During this period, the product goes through a series of phases such as requirements, specification, design, implementation, integration, maintenance, and ultimately, retirement. The software process also includes the tools and techniques we use to develop and maintain software as well as the software professionals involved.

In Chapter 1, "The Scope of Software Engineering," it is pointed out that techniques for software production must be cost effective and promote constructive interaction between the members of the software production team. The importance of objects is stressed throughout the book, starting with this chapter.

"The Software Process" is the title of Chapter 2. Each phase of the process is discussed in detail. Many problems of software engineering are described, but no solutions are put forward in this chapter. Instead, the reader is informed where in the book each problem is tackled. In this way, the chapter serves as a guide to the rest of the book. The chapter concludes with material on software process improvement.

A variety of different software life-cycle models are discussed in detail in Chapter 3, "Software Life-Cycle Models." These include the waterfall model, the rapid prototyping model, the incremental model, extreme programming, the synchronize-and-stabilize model, and the spiral model. To enable the reader to decide on an appropriate life-cycle model for a specific project, the various life-cycle models are compared and contrasted.

Chapter 4 is entitled "Teams." Today's projects are too large to be completed by a single individual within the given time constraints. Instead, a team of software professionals collaborate on the project. The major topic of this chapter is how teams should be organized so that team members work together productively. Various different ways of organizing teams are discussed, including democratic teams, chief programmer teams, synchronize-and-stabilize teams, and extreme programming teams.

Chapter 5 discusses "The Tools of the Trade." A software engineer needs to be able to use a number of different tools, both theoretical and practical. In this chapter, the reader is introduced to a variety of software engineering tools. One such tool is stepwise refinement, a technique for decomposing a large problem into smaller, more tractable problems. Another tool is cost–benefit analysis, a technique for determining whether a software project is financially feasible. Then, computer-aided software engineering (CASE) tools are

described. A CASE tool is a software product that assists software engineers to develop and maintain software. Finally, to manage the software process, it is necessary to measure various quantities to determine whether the project is on track. These measures (metrics) are critical to the success of a project.

The last two topics of Chapter 5, CASE tools and metrics, are treated in detail in Chapters 10 through 16, which describe the specific phases of the software life cycle. There is a discussion of the CASE tools that support each phase, as well as a description of the metrics needed to manage that phase adequately.

An important theme of this book is that testing is not a separate phase to be carried out just before delivering the product to the client or even at the end of each phase of the software life cycle. Instead, testing is performed in parallel with all software production activities. In Chapter 6, "Testing," the concepts underlying testing are discussed. The consideration of testing techniques specific to individual phases of the software life cycle is deferred until Chapters 10 through 16.

Chapter 7 is entitled "From Modules to Objects." A detailed explanation is given of classes and objects, and why the object-oriented paradigm is proving to be more successful than the structured paradigm. The concepts of this chapter then are utilized in the rest of the book, particularly Chapter 12, "Object-Oriented Analysis Phase," and in Chapter 13, "Design Phase," in which object-oriented design is presented.

The ideas of Chapter 7 are extended in Chapter 8, "Reusability, Portability, and Interoperability." It is important to be able to write reusable software that can be ported to a variety of different hardware and run on distributed architectures such as client–server. The first part of the chapter is devoted to reuse; the topics include a variety of reuse case studies as well as reuse strategies such as object-oriented patterns and frameworks. Portability is the second major topic; portability strategies are presented in some depth. The chapter concludes with interoperability topics such as CORBA and COM. A recurring theme of this chapter is the role of objects in achieving reusability, portability, and interoperability.

The last chapter in Part 1 is Chapter 9, "Planning and Estimating." Before starting a software project, it is essential to plan the entire operation in detail. Once the project begins, management must closely monitor progress, noting deviations from the plan and taking corrective action where necessary. Also, it is vital that the client be provided accurate estimates of how long the project will take and how much it will cost. Different estimation techniques are presented, including function points and COCOMO II. A detailed description of a software project management plan is given. The material of this chapter is utilized in Chapters 11 and 12. When the classical paradigm is used, major planning and estimating activities take place at the end of the specification phase, as explained in Chapter 11. When software is developed using the object-oriented paradigm, this planning takes place at the end of the object-oriented analysis phase (Chapter 12).

chapter

1

The Scope of Software Engineering

A well-known story tells of an executive who received a computer-generated bill for $0.00. After having a good laugh with friends about "idiot computers," the executive tossed the bill away. A month later a similar bill arrived, this time marked 30 days. Then came the third bill. The fourth bill arrived a month later, accompanied by a message hinting at possible legal action if the bill for $0.00 was not paid at once.

The fifth bill, marked 120 days, did not hint at anything—the message was rude and forthright, threatening all manner of legal actions if the bill was not immediately paid. Fearful of his organization's credit rating in the hands of this maniacal machine, the executive called an acquaintance who was a software engineer and related the whole sorry story. Trying not to laugh, the software engineer told the executive to mail a check for $0.00. This had the desired effect, and a receipt for $0.00 was received a few days later. The executive carefully filed it away in case at some future date the computer might allege that $0.00 was still owing.

This well-known story has a less well-known sequel. A few days later the executive was summoned by his bank manager. The banker held up a check and asked, "Is this your check?"

The executive agreed that it was.

"Would you mind telling me why you wrote a check for $0.00?" asked the banker.

So the whole story was retold. When the executive had finished, the banker turned to him and she quietly asked, "Have you any idea what your check for $0.00 did to *our* computer system?"

A computer professional can laugh at this story, albeit somewhat nervously. After all, every one of us has designed or implemented a product that, in its original form, would have resulted in the equivalent of sending dunning letters for $0.00. Up to now, we have always caught this sort of fault during testing. But our laughter has a hollow ring to it, because at the back of our minds is the fear that someday we will not detect the fault before the product is delivered to the customer.

A decidedly less humorous software fault was detected on November 9, 1979. The Strategic Air Command had an alert scramble when the worldwide military command and control system (WWMCCS) computer network reported that the Soviet Union had launched missiles aimed toward the United States [Neumann, 1980]. What actually happened was that a simulated attack was interpreted as the real thing, just as in the movie *WarGames* some 5 years later. Although the U.S. Department of Defense understandably has not given details about the precise mechanism by which test data were taken for actual data, it seems reasonable to ascribe the problem to a software fault. Either the system as a whole was not designed to differentiate between simulations and reality, or the user interface did not include the necessary checks for ensuring that end users

Just in Case You Wanted to Know

In the case of the WWMCCS network, disaster was averted at the last minute. However, the consequences of other software faults sometimes have been tragic. For example, between 1985 and 1987, at least two patients died as a consequence of severe overdoses of radiation delivered by the Therac-25 medical linear accelerator [Leveson and Turner, 1993]. The cause was a fault in the control software.

During the 1991 Gulf War, a Scud missile penetrated the Patriot antimissile shield and struck a barracks near Dhahran, Saudi Arabia. In all, 28 Americans were killed and 98 wounded. The software for the Patriot missile contained a cumulative timing fault. The Patriot was designed to operate for only a few hours at a time, after which the clock was reset. As a result, the fault never had a significant effect and therefore was not detected. In the Gulf War, however, the Patriot missile battery at Dhahran ran continuously for over 100 hours. This caused the accumulated time discrepancy to become large enough to render the system inaccurate.

During the Gulf War, the United States shipped Patriot missiles to Israel for protection against the Scuds. Israeli forces detected the timing problem after only 8 hours and immediately reported it to the manufacturer in the United States. The manufacturer corrected the fault as quickly as it could but, tragically, the new software arrived the day after the direct hit by the Scud [Mellor, 1994].

of the system would be able to distinguish fact from fiction. In other words, a software fault, if indeed the problem was caused by software, could have brought civilization as we know it to an unpleasant and abrupt end. (See the Just in Case You Wanted to Know box above for information on disasters caused by other software faults.)

Whether we are dealing with billing or air defense, much of our software is delivered late, over budget, and with residual faults. Software engineering is an attempt to solve these problems. In other words, software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user's needs. Furthermore, the software must be easy to modify when the user's needs change. To achieve these goals, a software engineer has to acquire a broad range of skills, both technical and managerial. These skills have to be applied not just to programming but to every phase of software production, from requirements to maintenance.

The scope of software engineering is extremely broad. Some aspects of software engineering can be categorized as mathematics or computer science; other aspects fall into the areas of economics, management, or psychology. To display the wide-reaching realm of software engineering, five different aspects now will be examined.

1.1 Historical Aspects

It is a fact that electric power generators fail, but far less frequently than payroll products. Bridges sometimes collapse, but considerably less often than operating systems. In the belief that software design, implementation, and maintenance could be

put on the same footing as traditional engineering disciplines, a NATO study group in 1967 coined the term *software engineering*. The claim that building software is similar to other engineering tasks was endorsed by the 1968 NATO Software Engineering Conference held in Garmisch, Germany [Naur, Randell, and Buxton, 1976]. This endorsement is not too surprising; the very name of the conference reflected the belief that software production should be an engineeringlike activity. A conclusion of the conferees was that software engineering should use the philosophies and paradigms of established engineering disciplines to solve what they termed the *software crisis;* namely, that the quality of software generally was unacceptably low and that deadlines and cost limits were not being met.

Despite many software success stories, a considerable amount of software still is being delivered late, over budget, and with residual faults. That the software crisis still is with us, over 30 years later, tells us two things. First, the software production process, while resembling traditional engineering in many respects, has its own unique properties and problems. Second, the software crisis perhaps should be renamed the *software depression,* in view of its long duration and poor prognosis.

Certainly, bridges collapse less frequently than operating systems. Why then cannot bridge-building techniques be used to build operating systems? What the NATO conferees overlooked is that bridges are as different from operating systems as chalk is from cheese.

A major difference between bridges and operating systems lies in the attitudes of the civil engineering community and the software engineering community to the act of collapsing. When a bridge collapses, as the Tacoma Narrows bridge did in 1940, the bridge almost always has to be redesigned and rebuilt from scratch. The original design was faulty and posed a threat to human safety; certainly, the design requires drastic changes. In addition, the effects of the collapse in almost every instance will have caused so much damage to the bridge fabric that the only reasonable thing is to demolish what is left of the faulty bridge, then completely redesign and rebuild it. Furthermore, other bridges built to the same design have to be carefully inspected and, in the worst case, redesigned and rebuilt.

In contrast, an operating system crash is not considered unusual and rarely triggers an immediate investigation into its design. When a crash occurs, it may be possible simply to reboot the system in the hope that the set of circumstances that caused the crash will not recur. This may be the only remedy if, as often is the case, there is no evidence as to the cause of the crash. The damage caused by the crash usually will be minor: a database partially corrupted, a few files lost. Even when damage to the file system is considerable, backup data often can restore the file system to a state not too far removed from its precrash condition. Perhaps, if software engineers treated an operating system crash as seriously as civil engineers treat a bridge collapse, the overall level of professionalism within software engineering would rise.

Now consider a real-time system, that is, a system able to respond to inputs from the real world as fast as they occur. An example is a computer-controlled intensive care unit. Irrespective of how many medical emergencies occur virtually simultaneously, the system must continue to alert the medical staff to every new emergency without ceasing to monitor those patients whose condition is critical but stable. In general, the failure of a real-time system, whether it controls an intensive care unit, a nuclear

reactor, or the climatic conditions aboard a space station, has significant effects. Most real-time systems, therefore, include some element of fault tolerance to minimize the effects of a failure. That is, the system is designed to attempt an automatic recovery from any failure.

The very concept of fault tolerance highlights a major difference between bridges and operating systems. Bridges are engineered to withstand every reasonably anticipated condition: high winds, flash floods, and so on. An implicit assumption of all too many software builders is that we cannot hope to anticipate all possible conditions that the software must withstand, so we must design our software to try to minimize the damage that an unanticipated condition might cause. In other words, bridges are assumed to be perfectly engineered. In contrast, most operating systems are assumed to be imperfectly engineered; many are designed in such a way that rebooting is a simple operation that the user may perform whenever needed. This difference is a fundamental reason why so much software today cannot be considered to be *engineered.*

It might be suggested that this difference is only temporary. After all, we have been building bridges for thousands of years, and we therefore have considerable experience and expertise in the types of conditions a bridge must withstand. We have only 50 years of experience with operating systems. Surely with more experience, the argument goes, we will understand operating systems as well as we understand bridges and so eventually will be able to construct operating systems that will not fail.

The flaw in this argument is that hardware, and hence the associated operating system, is growing in complexity faster than we can master it. In the 1960s, we had multiprogramming operating systems; in the 1970s, we had to deal with virtual memory; and now, we are attempting to come to terms with multiprocessor and distributed (network) operating systems. Until we can handle the complexity caused by the interconnections of the various components of a software product such as an operating system, we cannot hope to understand it fully; and if we do not understand it, we cannot hope to engineer it.

Part of the reason for the complexity of software is that, as it executes, software goes through discrete states. Changing even one bit causes the software to change state. The total number of such states can be vast, and many of them have not been considered by the development team. If the software enters such an unanticipated state, the result often is software failure. In contrast, bridges are continuous (analog) systems. They are described using continuous mathematics, essentially calculus. However, discrete systems such as operating systems have to be described using discrete mathematics [Parnas, 1990]. Software engineers therefore have to be skilled in discrete mathematics, a primary tool in trying to cope with this complexity.

A second major difference between bridges and operating systems is maintenance. Maintaining a bridge generally is restricted to painting it, repairing minor cracks, resurfacing the road, and so on. A civil engineer, if asked to rotate a bridge through 90° or to move it hundreds of miles, would consider the request outrageous. However, we think nothing of asking a software engineer to convert a batch operating system into a time-sharing one or to port it from one machine to another with totally different architectural characteristics. It is not unusual for 50 percent of the source

code of an operating system to be rewritten over a 5-year period, especially if it is ported to new hardware. But no engineer would consent to replacing half a bridge; safety requirements would dictate that a new bridge be built. The area of maintenance, therefore, is a second fundamental aspect in which software engineering differs from traditional engineering. Further maintenance aspects of software engineering are described in Section 1.3. But first, economic-oriented aspects are presented.

1.2 Economic Aspects

An insight into the relationship between software engineering and computer science can be obtained by comparing and contrasting the relationship between chemical engineering and chemistry. After all, computer science and chemistry are both sciences, and both have a theoretical component and a practical component. In the case of chemistry, the practical component is laboratory work; in the case of computer science, the practical component is programming.

Consider the process of extracting gasoline from coal. During World War II, the Germans used this process to make fuel for their war machine because they largely were cut off from oil supplies. While the antiapartheid oil embargo was in effect, the government of the Republic of South Africa poured billions of dollars into SASOL (an Afrikaans acronym standing for "South African coal into oil"). About half of South Africa's liquid fuel needs were met in this way.

From the viewpoint of a chemist, there are many possible ways to convert coal into gasoline and all are equally important. After all, no one chemical reaction is more important than any other. But from the chemical engineer's viewpoint, at any one time there is exactly one important mechanism for synthesizing gasoline from coal—the reaction that is economically the most attractive. In other words, the chemical engineer evaluates all possible reactions, then rejects all but that one reaction for which the cost per liter is the lowest.

A similar relationship holds between computer science and software engineering. The computer scientist investigates a variety of ways to produce software, some good and some bad. But the software engineer is interested in only those techniques that make sound economic sense.

For instance, a software organization currently using coding technique CT_{old} discovers that new coding technique, CT_{new}, would result in code being produced in only nine-tenths of the time needed by CT_{old} and, hence, at nine-tenths of the cost. Common sense seems to dictate that CT_{new} is the appropriate technique to use. In fact, although common sense certainly dictates that the faster technique is the technique of choice, the economics of software engineering may imply the opposite.

One reason is the cost of introducing new technology into an organization. The fact that coding is 10 percent faster when technique CT_{new} is used may be less important than the costs incurred in introducing CT_{new} into the organization. It may be necessary to complete two or three projects before recouping the cost of training.

Also, while attending courses on CT_{new}, software personnel are unable to do productive work. Even when they return, a steep learning curve may be involved; it may take months of practice with CT_{new} before software professionals become as proficient with CT_{new} as they currently are with CT_{old}. Therefore, initial projects using CT_{new} may take far longer to complete than if the organization had continued to use CT_{old}. All these costs need to be taken into account when deciding whether to change to CT_{new}.

A second reason why the economics of software engineering may dictate that CT_{old} be retained is the maintenance consequence. Coding technique CT_{new} indeed may be 10 percent faster than CT_{old}, and the resulting code may be of comparable quality from the viewpoint of satisfying the client's current needs. But the use of technique CT_{new} may result in code that is difficult to maintain, making the cost of CT_{new} higher over the life of the product. Of course, if the software developer is not responsible for any maintenance, then, from the viewpoint of just that developer, CT_{new} is a most attractive proposition. After all, use of CT_{new} would cost 10 percent less. The client should insist that technique CT_{old} be used and pay the higher initial costs with the expectation that the total lifetime cost of the software will be lower. Unfortunately, often the sole aim of both the client and the software provider is to produce code as quickly as possible. The long-term effects of using a particular technique generally are ignored in the interests of short-term gain. Applying economic principles to software engineering requires the client to choose techniques that reduce long-term costs.

We now consider the importance of maintenance.

1.3 Maintenance Aspects

The series of steps that software undergoes, from concept exploration through final retirement, is termed its *life cycle*. During this time, the product goes through a series of phases: requirements, specification, design, implementation, integration, maintenance, and retirement. Life-cycle models are discussed in greater detail in Chapter 3; the topic is introduced at this point so that the concept of maintenance can be defined.

Until the end of the 1970s, most organizations were producing software using as their life-cycle model what now is termed the *waterfall model*. There are many variations of this model, but by and large, the product goes through seven broad phases. These phases probably do not correspond exactly to the phases of any one particular organization, but they are sufficiently close to most practices for the purposes of this book. Similarly, the precise name of each phase varies from organization to organization. The names used here for the various phases have been chosen to be as general as possible in the hope that the reader will feel comfortable with them. For easy reference, the phases are summarized in Figure 1.1, which also indicates the chapters in this book in which they are presented.

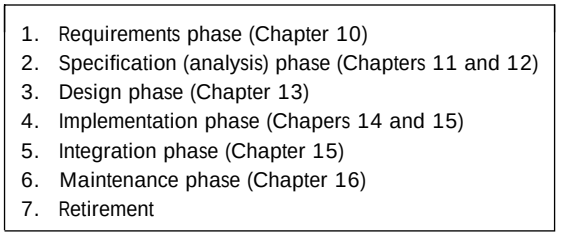

Figure 1.1 The phases of the software life cycle and the chapters in this boo in which they are presented.

1. *Requirements phase.* The concept is explored and refined, and the client's requirements are elicited.
2. *Specification (analysis) phase.* The client's requirements are analyzed and presented in the form of the *specification document,* "what the product is supposed to do." This phase sometimes is called the *analysis phase.* At the end of this phase, a plan is drawn up, the *software project management plan,* describing the proposed software development in full detail.
3. *Design phase.* The specifications undergo two consecutive design processes. First comes *architectural design,* in which the product as a whole is broken down into components, called *modules.* Then, each module is designed; this process is termed *detailed design.* The two resulting *design documents* describe "how the product does it."
4. *Implementation phase.* The various components are coded and tested.
5. *Integration phase.* The components of the product are combined and tested as a whole. When the developers are satisfied that the product functions correctly, it is tested by the client (*acceptance testing*). This phase ends when the product is accepted by the client and installed on the client's computer. (We will see in Chapter 15 that the integration phase should be performed in parallel with the implementation phase.)
6. *Maintenance phase.* The product is used to perform the tasks for which it was developed. During this time, it is maintained. Maintenance includes all changes to the product once the client has agreed that it satisfies the specification document (but see the Just in Case You Wanted to Know box on page 10). Maintenance includes *corrective maintenance* (or *software repair*), which consists of the removal of residual faults while leaving the specifications unchanged, as well as *enhancement* (or *software update*), which consists of changes to the specifications and the implementation of those changes. There are, in turn, two types of enhancement. The first is *perfective maintenance,* changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time. The second is *adaptive maintenance,* changes made in response to changes in the environment in which the product operates, such as

new government regulations. Studies have indicated that, on average, maintainers spend approximately 17.5 percent of their time on corrective maintenance, 60.5 percent on perfective maintenance, and 18 percent on adaptive maintenance [Lientz, Swanson, and Tompkins, 1978].

7. *Retirement.* The product is removed from service. This occurs when the functionality provided by the product no longer is of any use to the client organization.

Just in Case You Wanted to Know

In the 1970s, software production was viewed as consisting of two distinct activities performed sequentially: *development* followed by *maintenance.* Starting from scratch, the software product was developed then installed on the client's computer. Any change to the software after installation, whether to fix a residual fault or extend the functionality, constituted classical maintenance [IEEE 610.12, 1990]. Hence, the way that software was developed classically can be described as the *development-then-maintenance* model.

This is a temporal definition; that is, an activity is classified as development or maintenance depending on when it is performed. Suppose that a fault in the software is detected and corrected a day after the software has been installed. This clearly constitutes maintenance. But if the identical fault is detected and corrected the day before the software is installed, in terms of the classical definition, this constitutes development.

There are two reasons why this model is unrealistic today. First, nowadays it is certainly not unusual for construction of a product to take a year or more. During this time, the client's requirements may well change. For example, the client might insist that the product now be implemented on a faster microprocessor that has become available. Alternatively, the client organization may have expanded into Canada while development was under way, and the product now has to be modified so it also can handle sales in Canada. To see how this sort of change in requirements affects the software life cycle, suppose that the client's requirements change while the design is being developed. The software engineering team has to suspend development and modify the specification document to reflect the changed requirements. Furthermore, it then may be necessary to modify the design as well, if the changes to the specifications necessitate corresponding changes to those portions of the design already completed. Only when these changes have been made can development proceed. In other words, developers have to perform "maintenance" long before the product is installed.

A second problem with the classical development-then-maintenance model arose as a result of the way in which we now construct software. In classical software engineering, a characteristic of development was that the development team built the target product from scratch. In contrast, as a consequence of the high cost of software production today, developers try to reuse parts of existing software wherever possible in the software to be constructed (reuse is discussed in detail in Chapter 8). Therefore, the development-then-maintenance model is inappropriate whenever there is reuse.

A more realistic way of looking at maintenance is to view maintenance as the process that occurs when "software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation" [ISO/IEC 12207, 1995]. By this definition, maintenance occurs whenever a fault is fixed or the requirements change, irrespective of whether this takes place before or after installation of the product.

However, until such time as the majority of software engineers realize that the development-then-maintenance model is outdated, there is little point in trying to change the usage of the word *maintenance.* In this book, I accordingly still refer to maintenance as the activity carried out after development is complete and the product installed. Nevertheless, I hope that the true nature of maintenance soon will be more widely recognized.

Returning to the topic of maintenance, it sometimes is said that only bad software products undergo maintenance. In fact, the opposite is true; bad products are thrown away, whereas good products are repaired and enhanced, for 10, 15, or even 20 years. Furthermore, a software product is a model of the real world, and the real world is perpetually changing. As a consequence, software has to be maintained constantly for it to remain an accurate reflection of the real world.

For instance, if the sales tax rate changes from 6 percent to 7 percent, almost every software product that deals with buying or selling has to be changed. Suppose the product contains the C×× statement

```
const float salesTax = 6.0;
```

or the equivalent Java statement

```
public static final float salesTax = (float)6.0;
```

declaring that salesTax is a floating-point constant initialized to the value 6.0. In this case, maintenance is relatively simple. With the aid of a text editor the value 6.0 is replaced by 7.0 and the code is recompiled and relinked. However, if instead of using the name salesTax, the actual value 6.0 has been used in the product wherever the value of the sales tax is invoked, then such a product will be extremely difficult to maintain. For example, there may be occurrences of the value 6.0 in the source code that should be changed to 7.0 but are overlooked or instances of 6.0 that do not refer to sales tax but incorrectly are changed to 7.0. Finding these faults almost always is difficult and time consuming. In fact, with some software, it might be less expensive in the long run to throw away the product and recode it rather than try to determine which of the many constants need to be changed and how to make the modifications.

The real-time real world also is constantly changing. The missiles with which a jet fighter is armed may be replaced by a new model, requiring a change to the weapons control component of the associated avionics system. A six-cylinder engine is to be offered as an option in a popular four-cylinder automobile; this implies changing the on-board computer that controls the fuel injection system, timing, and so on.

Healthy organizations change; only dying organizations are static. This means that maintenance in the form of enhancement is a positive part of an organization's activities, reflecting that the organization is on the move.

But just how much time is devoted to maintenance? The pie chart in Figure 1.2 was obtained by averaging data from various sources, including [Elshoff, 1976; Daly, 1977; Zelkowitz, Shaw, and Gannon, 1979; and Boehm, 1981]. Figure 1.2 shows the approximate percentage of time (= money) spent on each phase of the software life cycle. About 15 years later, the proportion of time spent on the various development phases had hardly changed. This is shown in Figure 1.3, which compares the data in Figure 1.2 with more recent data on 132 Hewlett-Packard projects [Grady, 1994]. The data from Figure 1.2 have been grouped to make them comparable to the newer data.[1]

[1] figure 1.3 revects only the de-elopment phases. The proportion of de-elopment time de-oted to the require- ments and specification phases in figure 1.2 is (2 / 5), 33% or 21A% as shown in figure 1.3.

adverse effect on maintenance, an astute software manager will think twice before changing coding practices. The entire staff will have to be retrained, new software development tools purchased, and perhaps additional staff members hired who are experienced in the new technique. All this expense and disruption has to be endured for a possible 0.5 percent decrease in software costs because, as shown in Figure 1.2, module coding constitutes on average only 5 percent of total software costs.

Now suppose a new technique that reduces maintenance by 10 percent is developed. This probably should be introduced at once because, on average, it will reduce overall costs by 6.7 percent. The overhead involved in changing to this technique is a small price to pay for such large overall savings.

Because maintenance is so important, a major aspect of software engineering consists of those techniques, tools, and practices that lead to a reduction in maintenance costs.

1.4 Specification and Design Aspects

Software professionals are human and therefore sometimes make errors while developing a product. As a result, there will be a fault in the software. If the error is made during the requirements phase, then the resulting fault probably also will appear in the specifications, the design, and the code. Clearly, the earlier we correct a fault, the better.

The relative costs of fixing a fault at various phases in the software life cycle are shown in Figure 1.4 [Boehm, 1981]. The figure reflects data from IBM [Fagan, 1974], GTE [Daly, 1977], the Safeguard project [Stephenson, 1976], and some smaller TRW projects [Boehm, 1980]. The solid line in Figure 1.4 is the best fit for the data relating to the larger projects, and the dashed line is the best fit for the smaller projects. For each of the phases of the software life cycle, the corresponding relative cost to detect and correct a fault is depicted in Figure 1.5. Each step on the solid line in Figure 1.5 is constructed by taking the corresponding point on the solid straight line of Figure 1.4 and plotting the data on a linear scale.

Suppose it costs $40 to detect and correct a specific fault during the design phase. From the solid line in Figure 1.5 (projects between 1974 and 1980), that same fault would cost only about $30 to fix during the specification phase. But, during the maintenance phase, that fault would cost around $2000 to detect and correct. Newer data show that now it is even more important to detect faults early. The dashed line in Figure 1.5 shows the cost of detecting and correcting a fault during the development of system software for the IBM AS/400 [Kan et al., 1994]. On average, the same fault would have cost $3680 to fix during the maintenance phase of the AS/400 software.

The reason that the cost of correcting a fault increases so steeply is related to what has to be done to correct a fault. Early in the development life cycle, the product essentially exists only on paper, and correcting a fault may simply mean using an eraser and pencil. The other extreme is a product already delivered to a client. At the very least, correcting a fault means editing the code, recompiling and relinking it, and then carefully testing that the problem is solved. Next, it is critical to check that making the change has not created a new problem elsewhere in the product. All

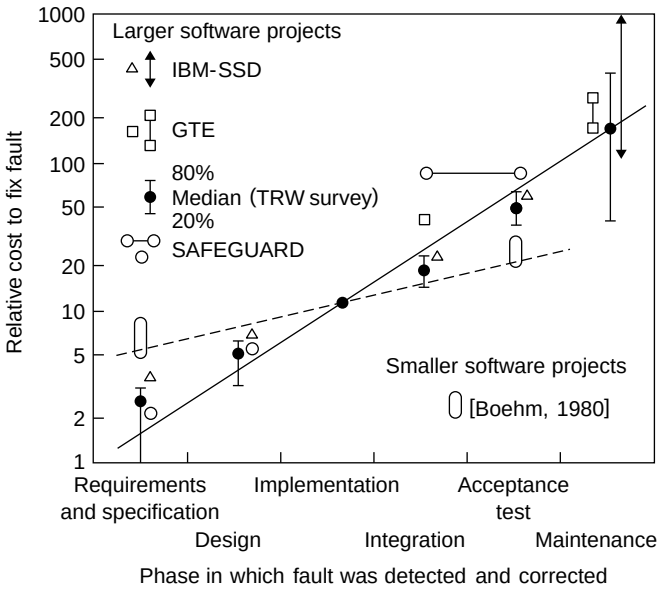

Figure 1.4 Relati-e cost of fi8ing a fault at each phase of the software life cycle. The solid line is the best fit for the data relating to the larger software pro‖ects% and the dashed line is the best fit for the smaller software pro‖ects. (©arry ©oehm% *Software Engineering Economics,* E1j91% p. 40. xdapted by permission of Brentice Pall% Inc.% Nnglewood Cliffs% J0.)

the relevant documentation, including manuals, then needs to be updated. Finally, the corrected product must be delivered and installed. The moral of the story is this: We must find faults early or else it will cost us money. We therefore should employ techniques for detecting faults during the requirements and specification (analysis) phases.

There is a further need for such techniques. Studies have shown [Boehm, 1979] that between 60 and 70 percent of all faults detected in large-scale projects are specification or design faults. Newer results bear out this preponderance of specification and design faults. An inspection is a careful examination of a document by a team (Section 6.2.3). During 203 inspections of Jet Propulsion Laboratory software for the NASA unmanned interplanetary space program, on average, about 1.9 faults were detected per page of a specification document, 0.9 faults per page of a design, but only 0.3 faults per page of code [Kelly, Sherif, and Hops, 1992].

Therefore, it is important that we improve our specification and design techniques, not only so that faults can be found as early as possible but also because specification and design faults constitute such a large proportion of all faults. Just as the example in the previous section showed that reducing maintenance costs by 10 percent will reduce overall costs by nearly 7 percent, reducing specification and design faults by 10 percent will reduce the overall number of faults by 6 to 7 percent.

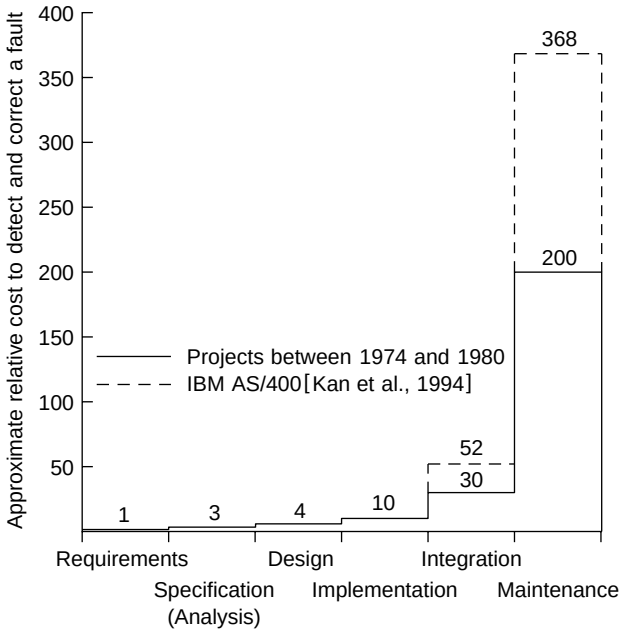

Figure 1.5 The solid line depicts the points on the solid line of fligure 1.4 plotted on a linear scale. The dashed line depicts newer data.

Newer data on a number of projects are reported in [Bhandari et al., 1994]. For example, a compiler was undergoing extensive changes. At the end of the design phase, the faults detected during the project were categorized. Only 13 percent of the faults were carry-overs from previous versions of the compiler. Of the remaining faults, 16 percent were introduced during the specification phase, and 71 percent were introduced during the design phase. That so many faults are introduced early in the software life cycle highlights another important aspect of software engineering; namely, techniques that yield better specifications and designs.

Most software is produced by a team of software engineers rather than by a single individual responsible for every phase of the development and maintenance life cycle. We now consider the implications of this.

1.5 Team Programming Aspects

The performance–price factor of a computer may be defined as follows:

$$\text{performance–price factor} * \frac{\text{time to perform 1 million additions}}{\text{cost of CPU and main memory}}$$

This quantity has decreased by an order of magnitude with each succeeding generation of computers. This decrease has been a consequence of discoveries in electronics, particularly the transistor, and very large-scale integration (VLSI).

The result of these discoveries has been that organizations easily can afford hardware that can run large products, that is, products too large to be written by one person within the allowed time constraints. For example, if a product has to be delivered within 18 months but would take a single programmer 15 years to complete, then the product must be developed by a team. However, team programming leads to interface problems among code components and communication problems among team members.

For example, Joe and Freda code modules p and q, respectively, where module p calls module q. When Joe codes p, he writes a call to q with five arguments in the argument list. Freda codes q with five arguments but in a different order from those of Joe. Unless function prototypes are used, this will not be detected by an ANSI C compiler. A few software tools, such as the Java interpreter and loader, *lint* for C (Section 8.7.4), or an Ada linker, detect such a type violation and only if the interchanged arguments are of different types; if they are of the same type, then the problem may not be detected for a long period of time. It may be debated that this is a design problem, and if the modules had been more carefully designed, this problem would not have happened. That may be true, but in practice a design often is changed after coding commences but notification of a change may not be distributed to all members of the development team. Thus, when a design that affects two or more programmers has been changed, poor communication can lead to the interface problems Joe and Freda experienced. This sort of problem is less likely to occur when only one individual is responsible for every aspect of the product, as was the case before powerful computers that can run huge products became affordable.

But interfacing problems are merely the tip of the iceberg when it comes to problems that can arise when software is developed by teams. Unless the team is properly organized, an inordinate amount of time can be wasted in conferences between team members. Suppose that a product takes a single programmer 1 year to complete. If the same task is assigned to a team of three programmers, the time for completing the task frequently is closer to 1 year than the expected 4 months, and the quality of the resulting code may well be lower than if the entire task had been assigned to one individual. Because a considerable proportion of today's software is developed and maintained by teams, the scope of software engineering must include techniques for ensuring that teams are properly organized and managed.

As has been shown in the preceding sections, the scope of software engineering is extremely broad. It includes every phase of the software life cycle, from requirements to retirement. It also includes human aspects, such as team organization; economic aspects; and legal aspects, such as copyright law. All these aspects implicitly are incorporated in the definition of software engineering given at the beginning of this chapter; namely, that software engineering is a discipline whose aim is the production of fault-free software delivered on time, within budget, and satisfying the user's needs.

1.6 THE OBJECT-ORIENTED PARADIGM

Before 1975, most software organizations used no specific techniques; each individual worked his or her own way. Major breakthroughs were made between approximately 1975 and 1985, with the development of the so-called structured paradigm. The techniques constituting the structured paradigm include structured systems analysis (Section 11.3), data flow analysis (Section 7.1), structured programming, and structured testing (Section 14.8.2). These techniques seemed extremely promising when first used. However, as time passed, they proved to be somewhat less successful in two respects. First, the techniques sometimes were unable to cope with the increasing size of software products. That is, the structured techniques were adequate when dealing with products of (say) 5,000 or even 50,000 lines of code. Today, however, products containing 500,000 lines of code are not considered large; even products of 5 million or more lines of code are not that unusual. However, the structured techniques frequently could not scale up sufficiently to handle today's larger products.

The maintenance phase is the second area in which the structured paradigm did not live up to earlier expectations. A major driving force behind the development of the structured paradigm 30 years ago was that, on average, two-thirds of the software budget was being devoted to maintenance (see Figure 1.2). Unfortunately, the structured paradigm has not solved this problem; as pointed out in Section 1.3, many organizations still spend up to 80 percent of their time and effort on maintenance [Yourdon, 1996].

The reason for the limited success of the structured paradigm is that the structured techniques are either action oriented or data oriented but not both. The basic components of a software product are the actions[2] of the product and the data on which those actions operate. For example, determine average height is an action that operates on a collection of heights (data) and returns the average of those heights (data). Some structured techniques, such as data flow analysis (Section 13.3), are action oriented. That is, such techniques concentrate on the actions of the product; the data are of secondary importance. Conversely, techniques such as Jackson system development (Section 13.5) are data oriented. The emphasis here is on the data; the actions that operate on the data are less significant.

In contrast, the object-oriented paradigm considers both data and actions to be equally important. A simplistic way of looking at an object is as a unified software component that incorporates both the data and the actions that operate on the data. This definition is incomplete and will be fleshed out later in the book, once *inheritance* has been defined (Section 7.7). Nevertheless, the definition captures much of the essence of an object.

A bank account is one example of an object (see Figure 1.6). The data component of the object is the account balance. The actions that can be performed on that account balance include deposit money, withdraw money, and determine balance. From the viewpoint of the structured paradigm, a product that deals with banking

[2] The word *action* is used in this boo rather than *process* to a-oid confusion with the term *software process.*

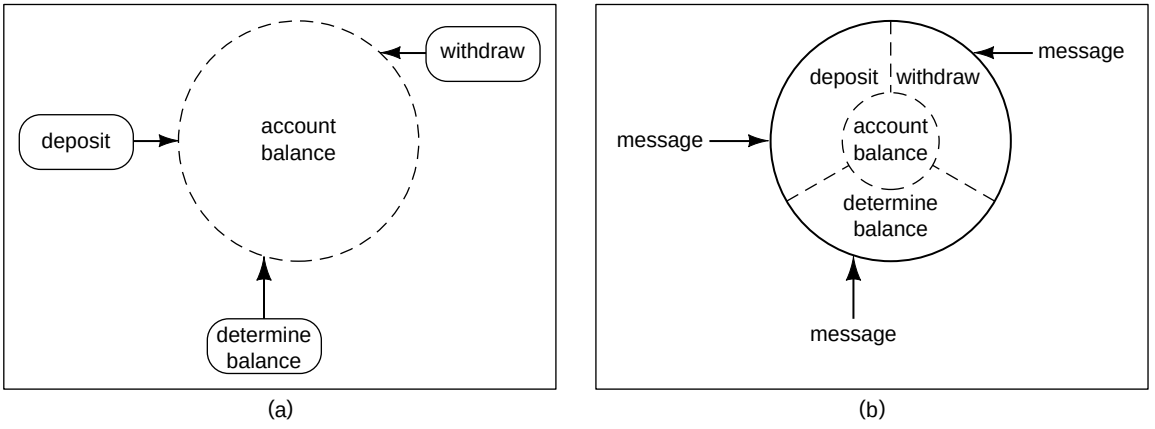

Figure 1.6 Comparison of implementations of bank account using (a) structured paradigm and (b) object-oriented paradigm. The solid black line surrounding the object denotes that details as to how account balance is implemented are not known outside the object.

would have to incorporate a data element, the account balance, and three actions, deposit, withdraw, and determine balance. From the object-oriented viewpoint, a bank account is an object. This object combines a data element together with the three actions performed on that data element in a single unit.

Up to now, there seems to be little difference between the two approaches. However, a key point is the way in which an object is implemented. Specifically, details as to how the data element of an object is stored are not known from outside the object. This is an instance of "information hiding" and is discussed in more detail in Section 7.6. In the case of the bank account object shown in Figure 1.6(b), the rest of the software product is aware that there is such a thing as a balance within a bank account object, but it has no idea as to the format of account balance. That is, there is no knowledge outside the object as to whether the account balance is implemented as an integer or a floating-point number or whether it is a field (component) of some larger structure. This information barrier surrounding the object is denoted by the solid black line in Figure 1.6(b), which depicts an implementation using the object-oriented paradigm. In contrast, a dashed line surrounds account balance in Figure 1.6(a), because all the details of account balance are known to the modules in the implementation using the structured paradigm, and the value of account balance therefore can be changed by any of them.

Returning to Figure 1.6(b), the object-oriented implementation, if a customer deposits $10 in an account, then a *message* is sent to the deposit action (*method*) of the relevant object telling it to increment the account balance data element (*attribute*) by $10. The deposit method is within the bank account object and knows how the account balance is implemented; this is denoted by the dashed line inside the object. But no entity external to the object needs to have this knowledge. That the three methods in

JUST IN CASE YOU WANTED TO KNOW

Suppose that you live in New Orleans, and you want to have a floral arrangement delivered to your aunt in Iowa City on her birthday [Budd, 1991]. One way would be to try to obtain a list of all the florists in Iowa City, then determine which one is located closest to your aunt's home. An easier way is to call 1-800-FLOWERS and leave the entire responsibility for delivering the floral arrangement to that organization. You need not know the identity of the Iowa City florist who will deliver the flowers.

In exactly the same way, when a message is sent to an object, not only is it entirely irrelevant how the request is carried out, but the unit that sends the message is not even allowed to know the internal structure of the object. The object itself is entirely responsible for every detail of carrying out the message.

Another positive feature of the object-oriented paradigm is that it promotes reuse: Because objects are independent entities, they can be utilized in future products. This reuse of objects reduces the time and cost of both development and maintenance, as explained in Chapter 8.

When the object-oriented paradigm is utilized, the software life cycle (Figure 1.1) has to be modified somewhat. Figure 1.7 shows the software life cycles of both structured and object-oriented paradigms. To appreciate the difference, first consider the design phase of the structured paradigm. As stated in Section 1.3, this phase is divided into two subphases: architectural design followed by detailed design. In the architectural design subphase, the product is decomposed into components, called *modules*. Then, during the detailed design subphase, the data structures and algorithms of each module are designed in turn. Finally, during the implementation phase, these modules are implemented.

If the object-oriented paradigm is used instead, one of the steps during the object-oriented analysis phase is to determine the objects. Because an object is a kind of module, architectural design therefore is performed during the object-oriented analysis

Structured æaradig0	O#Øct(Oriented æaradig0
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2◇. >bHect÷oriented analysis phase
3. Design phase	3◇. >bHect÷oriented design phase
4. Implementation phase	4◇. >bHect÷oriented programming phase
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase
7. Retirement	7. Retirement

Figure 1.7 Comparison of life cycles of the structured paradigm and the obHect÷oriented paradigm.

Structured æaradig0	O#Øct(Oriented æaradig0
2. Specification (analysis) phase · Determine what the product is to do	2◇. >b‖ect-oriented analysis phase · Determine what the product is to do · N8tract the ob‖ects
3. Design phase · xrchitectural design (e8tract the modules) · Detailed design	3◇. >b‖ect-oriented design phase · Detailed design
4. Implementation phase · Implement in appropriate programming language	4◇. >b‖ect-oriented programming phase · Implement in appropriate ob‖ect-oriented programming language

Figure 1.8 Differences between the structured paradigm and the ob‖ect-oriented paradigm.

phase. Thus, object-oriented analysis goes further than the corresponding specification (analysis) phase of the structured paradigm. This is shown in Figure 1.8.

This difference between the two paradigms has major consequences. When the structured paradigm is used, there almost always is a sharp transition between the analysis (specification) phase and the design phase. After all, the aim of the specification phase is to determine *what* the product is to do, whereas the purpose of the design phase is to decide *how* to do it. In contrast, when object-oriented analysis is used, objects enter the life cycle from the very beginning. The objects are extracted in the analysis phase, designed in the design phase, and coded in the implementation phase. Thus, the object-oriented paradigm is an integrated approach; the transition from phase to phase is far smoother than with the structured paradigm, thereby reducing the number of faults during development.

As already mentioned, it is inadequate to define an object merely as a software component that encapsulates both data and actions and implements the principle of information hiding. A more complete definition is given in Chapter 7, where objects are examined in depth. But first, the terminology used in this book must be considered in greater detail.

1.7 TERMINOLOGY

A word used on almost every page of this book is *software.* Software consists of not just code in machine-readable form but also all the documentation that is an intrinsic component of every project. Software includes the specification document, the design document, legal and accounting documents of all kinds, the software project management plan, and other management documents as well as all types of manuals.

Since the 1970s, the difference between a *program* and a *system* has become blurred. In the "good old days," the distinction was clear. A program was an autonomous piece of code, generally in the form of a deck of punched cards, that could be executed. A system was a related collection of programs. Thus, a system might consist of programs P, Q, R, and S. Magnetic tape T_1 was mounted, then program P was run. It caused a deck of data cards to be read in and produced as output tapes T_2 and T_3. Tape T_2 then was rewound, and program Q was run, producing tape T_4 as output. Program R now merged tapes T_3 and T_4 into tape $T_5 \pm T_5$ served as input for program S, which printed a series of reports.

Compare that situation with a product, running on a machine with a front-end communications processor and a back-end database manager, that performs real-time control of a steel mill. The single piece of software controlling the steel mill does far more than the old-fashioned system, but in terms of the classic definitions of program and system, this software undoubtedly is a program. To add to the confusion, the term *system* now also is used to denote the hardware–software combination. For example, the flight control system in an aircraft consists of both the in-flight computers and the software running on them. Depending on who is using the term, the flight control system also may include the controls, such as the joystick, that send commands to the computer and the parts of the aircraft, such as the wing flaps, controlled by the computer.

To minimize confusion, this book uses the term *product* to denote a nontrivial piece of software. There are two reasons for this convention. The first is simply to obviate the program versus system confusion by using a third term. The second reason is more important. This book deals with the *process* of software production, and the end result of a process is termed a *product. Software production* consists of two activities: *software development* followed by *maintenance.* Finally, the term *system* is used in its modern sense, that is, the combined hardware and software, or as part of universally accepted phrases, such as *operating system* and *management information system.*

Two words widely used within the context of software engineering are *methodology* and *paradigm.* Both are used in the same sense, a collection of techniques for carrying out the complete life cycle. This usage offends language purists; after all, *methodology* means the science of methods and a *paradigm* is a model or a pattern. Notwithstanding the best efforts of the author and others to encourage software engineers to use the words correctly, the practice is so widespread that, in the interests of clarity, both words are used in this book in the sense of a *collection of techniques.* Erudite readers offended by this corruption of the English language are warmly invited to take up the cudgels of linguistic accuracy on the author's behalf; he is tired of tilting at windmills.

One term that is avoided as far as possible is *bug* (the history of this word is in the Just in Case You Wanted to Know box on page 23). The term *bug* nowadays is simply a euphemism for *error.* Although there generally is no real harm in using euphemisms, the word *bug* has overtones that are not conducive to good software production. Specifically, instead of saying, "I made an error," a programmer will say, "A bug crept into the code" (not *my* code but *the* code), thereby transferring responsibility for the error from the programmer to the bug. No one blames a programmer for coming

Just in Case You Wanted to Know

The first use of the word *bug* to denote a fault is attributed to the late Rear Admiral Grace Murray Hopper, one of the designers of COBOL. On September 9, 1945, a moth flew into the Mark II computer that Hopper and her colleagues used at Harvard and lodged between the contact plates of a relay. Thus, there was actually a bug in the system. Hopper taped the bug to the log book and wrote, "First actual case of bug being found." The log book, with moth still attached, is in the Naval Museum at the Naval Surface Weapons Center, in Dahlgren, Virginia.

Although this may have been the first use of *bug* in a computer context, the word was used in engineering slang in the 19th century [Shapiro, 1994]. For example, Thomas Alva Edison wrote on November 18, 1878, "This thing gives out and then that—'Bugs'—as such little faults and difficulties are called …" [Josephson, 1992]. One of the definitions of *bug* in the 1934 edition of *Webster's New English Dictionary* is "A defect in apparatus or its operation." It is clear from Hopper's remark that she, too, was familiar with the use of the word in that context; otherwise, she would have explained what she meant.

down with a case of influenza, because the flu is caused by the flu bug. Referring to an error as a *bug* is a way of casting off responsibility. In contrast, the programmer who says, "I made an error," is a computer professional who takes responsibility for his or her actions.

There is considerable confusion regarding object-oriented terminology. For example, in addition to the term *attribute* for a data component of an object, the term *state variable* sometimes is used in the object-oriented literature. In Java, the term is *instance variable;* in C××, the term *field* is used. With regard to the actions of an object, the term *method* usually is used; in C××, however, the term is *member function.* In C××, a *member* of an object refers to either an attribute ("field") or a method. In Java, the term *field* is used to denote either an attribute ("instance variable") or a method. To avoid confusion, wherever possible, the generic terms *attribute* and *method* are used in this book.

Fortunately, some terminology is widely accepted. For example, when a method within an object is invoked, this almost universally is termed *sending a message* to the object.

In this section we defined the various terms used in this book. One of those terms, *process,* is the subject of the next chapter.

Chapter Review

Software engineering is defined (Section 1.1) as a discipline whose aim is the production of fault-free software that satisfies the user's needs and is delivered on time and within budget. To achieve this goal, appropriate techniques have to be used in

all phases of software production, including specification (analysis) and design (Section 1.4) and maintenance (Section 1.3). Software engineering addresses all phases of the software life cycle and incorporates aspects of many different areas of human knowledge, including economics (Section 1.2) and the social sciences (Section 1.5). In Section 1.6, objects are introduced, and a brief comparison between the structured and object-oriented paradigms is made. In the final section (Section 1.7), the terminology used in this book is explained.

FOR FURTHER READING

A classic source of information on the scope of software engineering is [Boehm, 1976]. [DeMarco and Lister, 1989] is a report on the extent to which software engineering techniques actually are used. For an analysis of the extent to which software engineering can be considered to be a true engineering discipline, see [Wasserman, 1996] and [Ebert, Matsubara, Pezzé, and Bertelsen, 1997]. The future of software engineering is discussed in [Lewis 1996a, 1996b; Leveson, 1997; Brereton et al., 1999; Kroeker et al., 1999; and Finkelstein, 2000]. Critical factors in software development are discussed in the May/June 1999 issue of *IEEE Software,* especially [Reel, 1999].

The current practice of software engineering is described in [Yourdon, 1996]. For a view on the importance of maintenance in software engineering and how to plan for it, see [Parnas, 1994]. The unreliability of software and the resulting risks (especially in safety-critical systems) are discussed in [Littlewood and Strigini, 1992; Mellor, 1994; and Neumann, 1995]. Modern views of the software crisis appear in [Gibbs, 1994] and [Glass, 1998]. [Zvegintzov, 1998] explains how little accurate data on software engineering practice actually is available.

The fact that mathematics underpins software engineering is stressed in [Parnas, 1990]. The importance of economics in software engineering is discussed in [Boehm, 1981] and [Baetjer, 1996].

Two classic books on the social sciences and software engineering are [Weinberg, 1971] and [Shneiderman, 1980]. Neither book requires prior knowledge of psychology or the behavioral sciences in general. A newer book on the topic is [DeMarco and Lister, 1987].

Brooks's timeless work, *The Mythical Man-Month* [Brooks, 1975], is a highly recommended introduction to the realities of software engineering. The book includes sections on all the topics mentioned in this chapter.

Excellent introductions to the object-oriented paradigm are [Budd, 1991] and [Meyer, 1997]. A balanced perspective of the paradigm is given in [Radin, 1996]. [Khan, Al-A'ali, and Girgis, 1995] explains the differences between the classical and object-oriented paradigms. Three successful projects carried using the object-oriented paradigm are described in [Capper, Colgate, Hunter, and James, 1994]. A survey of the attitudes of 150 experienced software developers toward the object-oriented paradigm is reported in [Johnson, 2000]. Lessons learned from developing large-scale object-oriented products are presented in [Maring, 1996] and [Fichman and Kemerer, 1997].

[Scholtz et al., 1993] is a report on a workshop held in April 1993 on the state of the art and the practice of object-oriented programming. A variety of short articles on recent trends in the object-oriented paradigm can be found in [El-Rewini et al., 1995]. Important articles on the object-oriented paradigm are found in the October 1992 issue of *IEEE Computer,* the January 1993 issue of *IEEE Software,* and the January 1993 and November 1993 issues of the *Journal of Systems and Software.* Potential pitfalls of the object-oriented paradigm are described in [Webster, 1995].

PROBLEMS

3 You are in charge of developing a raw materials control system for a major manufacturer of digital telephones. Your development budget is $430,000. Approximately how much money should you devote to each phase of the development life cycle? How much do you expect future maintenance will cost?

3' You are a software-engineering consultant. The executive vice-president of a publisher of paperback books wants you to develop a product that will carry out all the accounting functions of the company and provide online information to the head office staff regarding orders and inventory in the various company warehouses. Terminals are required for 15 accounting clerks, 32 order clerks, and 42 warehouse clerks. In addition, 18 managers need access to the data. The president is willing to pay $30,000 for the hardware and the software together and wants the complete product in 4 weeks. What do you tell him? Bear in mind that, as a consultant, you want his business, no matter how unreasonable his request.

3 You are a vice-admiral of the Navy of the Republic of Claremont. It has been decided to call in a software development organization to develop the control software for a new generation of ship-to-ship missiles. You are in charge of supervising the project. To protect the government of Claremont, what clauses do you include in the contract with the software developers?

34 You are a software engineer and your job is to supervise the development of the software in Problem 1.3. List ways your company can fail to satisfy the contract with the Navy. What are the probable causes of such failures?

35 Fifteen months after delivery, a fault is detected in a mechanical engineering product that determines the optimal viscosity of oil in internal combustion engines. The cost of fixing the fault is $18,730. The cause of the fault is an ambiguous sentence in the specification document. Approximately how much would it have cost to have corrected the fault during the specification phase?

38. Suppose that the fault in Problem 1.5 had been detected during the implementation phase. Approximately how much would it have cost to have fixed it then?

3% You are the president of an organization that builds large-scale software. You show Figure 1.5 to your employees, urging them to find faults early in the software life cycle. Someone responds that it is unreasonable to expect anyone to remove faults before having entered the product. For example, how can anyone remove a fault during the design phase if the fault in question is a coding fault? What do you reply?

3 Look up the word *system* in a dictionary. How many different definitions are there? Write down those definitions that are applicable within the context of software engineering.

3 It is your first day at your first job. Your manager hands you a listing and says, "See if you can find the bug." What do you reply?

3 6 You are in charge of developing the product in Problem 1.1. Will you use the object-oriented paradigm or the structured paradigm? Give reasons for your answer.

3 (Term Project) Suppose that the Broadlands Area Children's Hospital (BACH) product of Appendix A has been implemented exactly as described. Now the Joining Children with their Families (JCF) program is to be extended to all patients, not just those living within a 500-mile radius of the city of Broadlands. In what ways will the existing product have to be changed? Would it be better to discard everything and start again from scratch?

3 " (Readings in Software Engineering) Your instructor will distribute copies of [Reel, 1999]. How would you manage a product to be able to detect any signs of project failure as early as possible?

REFERENCES

[Baetjer, 1996] H. BAETJER, *Software as Capital: An Economic Perspective on Software Engineering,* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Bhandari et al., 1994] I. BHANDARI, M. J. HALLIDAY, J. CHAAR, R. CHILLAREGE, K. JONES, J. S. ATKINSON, C. LEPORI-COSTELLO, P. Y. JASPER, E. D. TARVER, C. C. LEWIS, AND M. YONEZAWA, "In-Process Improvement through Defect Data Interpretation," *IBM Systems Journal* **33** (No. 1, 1994), pp. 182–214.

[Boehm, 1976] B. W. BOEHM, "Software Engineering," *IEEE Transactions on Computers* **C-25** (December 1976), pp. 1226–41.

[Boehm, 1979] B. W. BOEHM, "Software Engineering, R & D Trends and Defense Needs," in *Research Directions in Software Technology,* P. Wegner (Editor), The MIT Press, Cambridge, MA, 1979.

[Boehm, 1980] B. W. BOEHM, "Developing Small-Scale Application Software Products: Some Experimental Results," *Proceedings of the Eighth IFIP World Computer Congress,* October 1980, pp. 321–26.

[Boehm, 1981] B. W. BOEHM, *Software Engineering Economics,* Prentice Hall, Englewood Cliffs, NJ, 1981.

[Brereton et al., 1999] P. BRERETON, D. BUDGEN, K. BENNETT, M. MUNRO, P. LAYZELL, L. MACAULAY, D. GRIFFITHS, AND C. STANNETT, "The Future of Software," *Communications of the ACM* **42** (December 1999), pp. 78–84.

[Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, Reading, MA, 1975. Twentieth Anniversary Edition, Addison-Wesley, Reading, MA, 1995.

[Budd, 1991] T. A. BUDD, *An Introduction to Object-Oriented Programming,* Addison-Wesley, Reading, MA, 1991.

[Capper, Colgate, Hunter, and James, 1994] N. P. CAPPER, R. J. COLGATE, J. C. HUNTER, AND M. F. JAMES, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal* **33** (No. 1, 1994), pp. 131–57.

[Coleman, Ash, Lowther, and Oman, 1994] D. COLEMAN, D. ASH, B. LOWTHER, AND P. OMAN, "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer* **27** (August 1994), pp. 44–49.

[Daly, 1977] E. B. DALY, "Management of Software Development," *IEEE Transactions on Software Engineering* **SE-3** (May 1977), pp. 229–42.

[DeMarco and Lister, 1987] T. DEMARCO AND T. LISTER, *Peopleware: Productive Projects and Teams,* Dorset House, New York, 1987.

[DeMarco and Lister, 1989] T. DEMARCO AND T. LISTER, "Software Development: The State of the Art vs. State of the Practice," *Proceedings of the 11th International Conference on Software Engineering,* Pittsburgh, May 1989, pp. 271–75.

[Ebert, Matsubara, Pezzé, and Bertelsen, 1997] C. EBERT, T. MATSUBARA, M. PEZZÉ, AND O. W. BERTELSEN, "The Road to Maturity: Navigating between Craft and Science," *IEEE Software* **14** (November/December 1997), pp. 77–88.

[El-Rewini et al., 1995] H. EL-REWINI, S. HAMILTON, Y.-P. SHAN, R. EARLE, S. MCGAUGHEY, A. HELAL, R. BADRACHALAM, A. CHIEN, A. GRIMSHAW, B. LEE, A. WADE, D. MORSE, A. ELMAGRAMID, E. PITOURA, R. BINDER, AND P. WEGNER, "Object Technology," *IEEE Computer* **28** (October 1995), pp. 58–72.

[Elshoff, 1976] J. L. ELSHOFF, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering* **SE-2** (June 1976), 113–20.

[Fagan, 1974] M. E. FAGAN, "Design and Code Inspections and Process Control in the Development of Programs," Technical Report IBM-SSD TR 21.572, IBM Corporation, December 1974.

[Fichman and Kemerer, 1997] R. G. FICHMAN AND C. F. KEMERER, "Object Technology and Reuse: Lessons from Early Adopters," *IEEE Computer* **30** (July 1997), pp. 47–57.

[Finkelstein, 2000] A. FINKELSTEIN (EDITOR), *The Future of Software Engineering,* IEEE Computer Society Press, Los Alamitos, CA, 2000.

[Gibbs, 1994] W. W. GIBBS, "Software's Chronic Crisis," *Scientific American* **271** (September 1994), pp. 86–95.

[Glass, 1998] R. L. GLASS, "Is There Really a Software Crisis?" *IEEE Software* **15** (January/February 1998), pp. 104–5.

[Grady, 1994] R. B. GRADY, "Successfully Applying Software Metrics," *IEEE Computer* **27** (September 1994), pp. 18–25.

[IEEE 610.12, 1990] "A Glossary of Software Engineering Terminology," IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1990.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[Johnson, 2000] R. A. JOHNSON, "The Ups and Downs of Object-Oriented System Development," *Communications of the ACM* **43** (October 2000), pp. 69–73.

[Josephson, 1992] M. JOSEPHSON, *Edison: A Biography,* John Wiley and Sons, New York, 1992.

[Kan et al., 1994] S. H. KAN, S. D. DULL, D. N. AMUNDSON, R. J. LINDNER, AND R. J. HEDGER, "AS/400 Software Quality Management," *IBM Systems Journal* **33** (No. 1, 1994), pp. 62–88.

[Kelly, Sherif, and Hops, 1992] J. C. KELLY, J. S. SHERIF, AND J. HOPS, "An Analysis of Defect Densities Found during Software Inspections," *Journal of Systems and Software* **17** (January 1992), pp. 111–17.

[Khan, Al-A'ali, and Girgis, 1995] E. H. KHAN, M. AL-A'ALI, AND M. R. GIRGIS, "Object-Oriented Programming for Structured Procedural Programming," *IEEE Computer* **28** (October 1995), pp. 48–57.

[Kroeker et al., 1999] K. K. KROEKER, L. WALL, D. A. TAYLOR, C. HORN, P. BASSETT, J. K. OUSTERHOUT, M. L. GRISS, R. M. SOLEY, J. WALDO, AND C. SIMONYI, "Software [R]evolution: A Roundtable," *IEEE Computer* **32** (May 1999), pp. 48–57.

[Leveson, 1997] N. G. LEVESON, "Software Engineering: Stretching the Limits of Complexity," *Communications of the ACM* **40** (February 1997), pp. 129–31.

[Leveson and Turner, 1993] N. G. LEVESON AND C. S. TURNER, "An Investigation of the Therac-25 Accidents," *IEEE Computer* **26** (July 1993), pp. 18–41.

[Lewis, 1996a] T. LEWIS, "The Next $10{,}000_2$ Years: Part I," *IEEE Computer* **29** (April 1996), pp. 64–70.

[Lewis, 1996b] T. LEWIS, "The Next $10{,}000_2$ Years: Part II," *IEEE Computer* **29** (May 1996), pp. 78–86.

[Lientz, Swanson, and Tompkins, 1978] B. P. LIENTZ, E. B. SWANSON, AND G. E. TOMPKINS, "Characteristics of Application Software Maintenance," *Communications of the ACM* **21** (June 1978), pp. 466–71.

[Littlewood and Strigini, 1992] B. LITTLEWOOD AND L. STRIGINI, "The Risks of Software," *Scientific American* **267** (November 1992), pp. 62–75.

[Maring, 1996] B. MARING, "Object-Oriented Development of Large Applications," *IEEE Software* **13** (May 1996), pp. 33–40.

[Mellor, 1994] P. MELLOR, "CAD: Computer-Aided Disaster," Technical Report, Centre for Software Reliability, City University, London, U.K., July 1994.

[Meyer, 1992a] B. MEYER, "Applying 'Design by Contract'," *IEEE Computer* **25** (October 1992), pp. 40–51.

[Meyer, 1997] B. MEYER, *Object-Oriented Software Construction,* 2nd ed., Prentice Hall, Upper Saddle River, NJ, 1997.

[Naur, Randell, and Buxton, 1976] P. NAUR, B. RANDELL, AND J. N. BUXTON (Editors), *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences,* Petrocelli-Charter, New York, 1976.

[Neumann, 1980] P. G. NEUMANN, Letter from the Editor, *ACM SIGSOFT Software Engineering Notes* **5** (July 1980), p. 2.

[Neumann, 1995] P. G. NEUMANN, *Computer-Related Risks,* Addison-Wesley, Reading, MA, 1995.

[Parnas, 1990] D. L. PARNAS, "Education for Computing Professionals," *IEEE Computer* **23** (January 1990), pp. 17–22.

[Parnas, 1994] D. L. PARNAS, "Software Aging," *Proceedings of the 16th International Conference on Software Engineering,* Sorrento, Italy, May 1994, pp. 279–87.

[Radin, 1996] G. RADIN, "Object Technology in Perspective," *IBM Systems Journal* **35** (No. 2, 1996), pp. 124–126.

[Reel, 1999] J. S. REEL, "Critical Success Factors in Software Projects," *IEEE Software* **16** (May/June 1999), pp. 18–23.

[Scholtz et al., 1993] J. SCHOLTZ, S. CHIDAMBER, R. GLASS, A. GOERNER, M. B. ROSSON., M. STARK, AND I. VESSEY, "Object-Oriented Programming: The Promise and the Reality," *Journal of Systems and Software* **23** (November 1993), pp. 199–204.

[Shapiro, 1994] F. R. SHAPIRO, "The First Bug," *Byte* **19** (April 1994), p. 308.

[Shneiderman, 1980] B. SHNEIDERMAN, *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishers, Cambridge, MA, 1980.

[Stephenson, 1976] W. E. STEPHENSON, "An Analysis of the Resources Used in Safeguard System Software Development," Bell Laboratories, Draft Paper, August 1976.

[Wasserman, 1996] A. I. WASSERMAN, "Toward a Discipline of Software Engineering," *IEEE Software* **13** (November/December 1996), pp. 23–31.

[Webster, 1995] B. F. WEBSTER, *Pitfalls of Object-Oriented Development,* M&T Books, New York, 1995.

[Weinberg, 1971] G. M. WEINBERG, *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.

[Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER, *Designing Object-Oriented Software,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Yourdon, 1996] E. YOURDON, *Rise and Resurrection of the American Programmer,* Yourdon Press, Upper Saddle River, NJ, 1996.

[Zelkowitz, Shaw, and Gannon, 1979] M. V. ZELKOWITZ, A. C. SHAW, AND J. D. GANNON, *Principles of Software Engineering and Design,* Prentice Hall, Englewood Cliffs, NJ, 1979.

[Zvegintzov, 1998] N. ZVEGINTZOV, "Frequently Begged Questions and How to Answer Them," *IEEE Software* **15** (January/February 1998), pp. 93–96.