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.