2. So many changes may have been made to the original design that interdependencies inadvertently have been built into the product, and even a small change to one minor module might have a drastic effect on the functionality of the product as a whole.
3. The documentation may not have been adequately maintained, thus increasing the risk of a regression fault to the extent that it would be safer to recode than to maintain.
4. The hardware (and operating system) on which the product runs is to be replaced; it may be more economical to rewrite from scratch than to modify.

In each of these instances the current version is replaced by a new version, and the software process continues.

True retirement, on the other hand, is a somewhat rare event that occurs when a product has outgrown its usefulness. The client organization no longer requires the functionality provided by the product, and it finally is removed from the computer.

After this review of the complete software process, together with some of the difficulties attendant on each phase, it is time to consider difficulties associated with software production as a whole.

2.9 Problems with Software Production: Essence and Accidents

Over the past 50 years, hardware has become cheaper and faster. Furthermore, hardware has shrunk in size. In the 1950s, companies paid hundreds of thousands of preinflation dollars for a machine as large as a room that was no more powerful than today's desktop personal computer selling for under $1000. It would seem that this trend is inexorable and that computers will continue to become smaller, faster, and cheaper.

Unfortunately, this is not the case. A number of physical constraints must eventually impose limits on the possible future size and speed of hardware. The first of these constraints is the speed of light. Electrons, or more precisely, electromagnetic waves, simply cannot travel faster than 186,300 miles per second. One way to speed up a computer therefore is to miniaturize its components. In that way, the electrons have shorter distances to travel. However, there also are lower limits on the size of components. An electron travels along a path that can be as narrow as three atoms in width. But if the path along which the electron is to travel is any narrower than that, then the electron can stray onto an adjacent path. For the same reason, parallel paths must not be located too close to one another. Thus, the speed of light and the nonzero width of an atom impose physical limits on hardware size and speed. We are nowhere near these limits yet—computers easily can become at least two orders of magnitude faster and smaller without reaching these physical limits. But intrinsic laws of nature eventually will prevent electronic computers from becoming arbitrarily fast or arbitrarily small.

Now what about software? Software essentially is conceptual, and therefore nonphysical, although of course it always is stored on some physical medium such as paper or magnetic disk. Superficially, it might appear that with software anything is possible. But Fred Brooks, in a landmark article entitled "No Silver Bullet" [Brooks, 1986], exploded this belief. He argued that, analogous to hardware speed and size limits that cannot be exceeded physically, there are inherent problems with current techniques of software production that never can be solved. To quote Brooks, "building software will always be hard. There is inherently no silver bullet." (An explanation of the term *silver bullet* is in the Just in Case You Wanted to Know box below.)

Recall that the title of Brooks's article is "*No* Silver Bullet" [author's italics]. Brooks's theme is that the very nature of software makes it highly unlikely that a silver bullet will be discovered that magically will solve all the problems of software production, let alone help achieve software breakthroughs comparable to those that have occurred with unfailing regularity in the hardware field. He divides the difficulties of software into two Aristotelian categories: *essence,* the difficulties inherent in the intrinsic nature of software, and *accidents,* the difficulties encountered today that are not inherent in software production. That is, essence constitutes those aspects of software production that probably cannot be changed, whereas accidents are amenable to research breakthroughs, or silver bullets.

What aspects of software production are inherently difficult? Brooks lists four, which he terms *complexity, conformity, changeability,* and *invisibility.* Brooks's use of the word *complexity* is somewhat unfortunate in that the term has many different meanings in computer science in general and in software engineering in particular. In the context of his article, Brooks uses the word *complex* in the sense of "complicated" or "intricate." In fact, the names of all four aspects are used in their nontechnical sense. Each of the four aspects is now examined.

2.9.1 Complexity

Software is more complex than any other construct made by human beings. Even hardware is almost trivial compared to software. To see this, consider a 16-bit word w in the main memory of a computer. Because each of the 16 bits can take on exactly two values, 0 and 1, word w as a whole can be in any of 2^{16} different states. If we have

Just in Case You Wanted to Know

The "silver bullet" in the title of Brooks's article refers to the recommended way of slaying werewolves, otherwise perfectly normal human beings who suddenly turn into wolves. Brooks's line of inquiry is to determine whether a similar silver bullet can be used to solve the problems of software. After all, software usually appears to be innocent and straightforward. But, like a werewolf, software can be transformed into something horrifying, in the shape of late deadlines, exceeded budgets, and residual specification and design faults not detected during testing.

two words, w_1 and w_2, each 16 bits in length, then the number of possible states of words w_1 and w_2 together is 2^{16} times 2^{16}, or 2^{32}. In general, if a system consists of a number of independent pieces, then the number of possible states of that system is the product of the numbers of possible states of each component.

Suppose the computer is to be used to run a software product p and the 16-bit word w is to be used to store the value of an integer x. If the value of x is read in by a statement such as read (x), then, because the integer x can take on 2^{16} different values, at first sight it might seem that the number of states in which the product could be is the same as the number of states in which the word could be. If the product p consists only of the single statement read (x), then the number of states of p indeed would be 2^{16}. But in a realistic, nontrivial software product, the value of a variable that is input is later used elsewhere in the product. There is an interdependence between the read (x) statement and any statement that uses the value of x. The situation is more complex if the flow of control within the product depends on the value of x. For example, x may be the control variable in a **switch** statement, or there may be a **for** loop or **while** loop whose termination depends on the value of x. Thus the number of states in any nontrivial product is greater, because of this interaction, than the product of the number of states of each variable. As a consequence of this combinatorial explosion in the number of states, complexity does not grow linearly with the size of the product but much faster.

Complexity is an inherent property of software. Irrespective of how a nontrivial piece of software is designed, the pieces of the product interact. For example, the states of a module depend on the states of its arguments, and the states of global variables (variables that can be accessed by more than one module) also affect the state of the product as a whole. Certainly, complexity can be reduced; for example, by using the object-oriented paradigm. Nevertheless, it never can be eliminated totally. In other words, complexity is an essential property of software not an accidental one.

Brooks points out that complex phenomena can be described and explained in disciplines such as mathematics and physics. Mathematicians and physicists have learned how to abstract the essential features of a complex system, to build a simple model that reflects only those essential features, to validate the simple model, and to make predictions from it. In contrast, if software is simplified, the whole exercise is useless; the simplifying techniques of mathematics and physics work only because the complexities of those systems are accidents, not essence, as is the case with software products.

The consequence of this essential complexity of software is that a product becomes difficult to understand. In fact, often no one really understands a large product in its entirety. This leads to imperfect communication between team members that, in turn, results in the time and cost overruns that characterize the development of large-scale software products. In addition, errors in specifications are made simply because of a lack of understanding of all aspects of the product.

This essential complexity affects not only the software process itself but also the management of the process. Unless a manager can obtain accurate information regarding the process he or she is managing, it is difficult to determine personnel needs for the succeeding stages of the project and to budget accurately. Reports to senior management regarding both progress to date and future deadlines are likely to be inaccurate. Drawing up a testing schedule is difficult when neither the manager nor anyone reporting to the manager knows what loose ends still have to be tied. And if a project staff member leaves, trying to train a replacement can be a nightmare.

A further consequence of the complexity of software is that it complicates the maintenance process. As shown in Figure 1.2, about two-thirds of the total software effort is devoted to maintenance. Unless the maintainer really understands the product, corrective maintenance or enhancement could damage the product in such a way that further maintenance is required to repair the damage caused by the original maintenance. The possibility of this sort of damage being caused by carelessness always is present, even when the original author makes the change, but it is exacerbated when the maintenance programmer effectively is working in the dark. Poor documentation; or worse, no documentation; or still worse, incorrect documentation often is a major cause of incorrectly performed maintenance. But, no matter how good the documentation may be, the inherent complexity of software transcends all attempts to cope with it; and this complexity has an unfavorable impact on maintenance. Again, the object-oriented paradigm can help reduce complexity (and hence improve maintenance), but it cannot eliminate it completely.

2.9.2 Conformity

A manually controlled gold refinery is to be computerized. Instead of the plant being operated via a series of buttons and levers, a computer will send the necessary control signals to the components of the plant. Although the plant is working perfectly, management feels that a computerized control system will increase the gold yield. The task of the software development team is to construct a product that will interface with the existing plant. That is, the software must conform to the plant, not the plant to the software. This is the first type of conformity identified by Brooks, where software acquires an unnecessary degree of complexity because it has to interface with an existing system.

What if a brand-new computerized gold refinery were to be constructed? It would appear that the mechanical engineers, metallurgical engineers, and software engineers together could come up with a plant design in which the machinery and the software fit together in a natural and straightforward manner. In practice, however, there generally is a perception that it is easier to make the software interface conform to the other components than to change the way the other components have been configured in the past. As a result, even in a new gold refinery, the other engineers will insist on designing the machinery as before, and the software will be forced to conform to the hardware interfaces. This is the second type of conformity identified by Brooks, where software acquires an unnecessary degree of complexity because of the misconception that software is the most conformable component.

The problems caused by this forced conformity cannot be removed by redesigning the software, because the complexity is not due to the structure of the software itself. Instead, it is due to the structure of software caused by the interfaces, to humans or to hardware, imposed on the software designer (but see the Just in Case You Wanted to Know box on page 48 for details of how this may change in the future).

JUST IN CASE YOU WANTED TO KNOW

According to Section 1.4, pages 15–17 of the *Technical Manual* for the *Star Trek* starship U.S.S. Enterprise, NCC–1701–D, much of the software development was started before the hardware development [Sternbach and Okuda, 1991]. I hope that, in this respect at least, *Star Trek* turns out to be science fact rather than science fiction.

2.9.3 CHANGEABILITY

As pointed out in Section 1.1, it is considered unreasonable to ask a civil engineer to move a bridge 300 miles or rotate it 90°, but it is perfectly acceptable to tell a software engineer to rewrite half an operating system over a 5-year period. Civil engineers know that redesigning half a bridge is expensive and dangerous; it is both cheaper and safer to rebuild it from scratch. Software engineers are equally well aware that, in the long run, extensive maintenance is unwise and rewriting the product from scratch sometimes will prove less expensive. Nevertheless, clients frequently demand major changes to software.

Brooks points out that there always will be pressures to change software. After all, it *is* easier to change software than, say, the hardware on which it runs; that is the reason behind the terms *soft*ware and *hard*ware. In addition, the functionality of a system is embodied in its software, and changes in functionality are achieved through changing the software. It has been suggested that the problems caused by frequent and drastic maintenance are merely problems caused by ignorance, and if the public at large were better educated with regard to the nature of software, then demands for major changes would not occur. But Brooks points out that changeability is a property of the essence of software, an inherent problem that cannot be surmounted. That is, the very nature of software is such that, no matter how the public is educated, there always will be pressures for changes in software, and often these changes will be drastic.

There are four reasons why useful software has to undergo change:

1. As pointed out in Section 1.3, software is a model of reality, and as the reality changes, so the software must adapt or die.
2. If software is found to be useful, then there are pressures, chiefly from satisfied users, to extend the functionality of the product beyond what is feasible in terms of the original design.
3. One of the greatest strengths of software is that it is so much easier to change than hardware.
4. Successful software survives well beyond the lifetime of the hardware for which it was written. In part, this is because, after 4 or 5 years, hardware often does not

function as well as it did. But more significant is the fact that technological change is so rapid that more appropriate hardware components, such as larger disks, faster CPUs, or more powerful monitors, become available while the software still is viable. In general, the software has to be modified to some extent to run on the new hardware.

For all these reasons, part of the essence of software is that it has to be changed, and this inexorable and continual change has a deleterious effect on the quality of software.

2.9.4 Invisibility

A major problem with the essence of software is that it is "invisible and unvisualizable" [Brooks, 1986]. Anyone who has been handed a 200-page listing and told to modify the software in some way knows exactly what Brooks is saying. Unfortunately, there is no acceptable way to represent either a complete product or some sort of overview of the product. In contrast, architects, for example, can provide 3-dimensional models that give an idea of the overall design, as well as 2-dimensional blueprints and other detailed diagrams that, to the trained eye, will reflect every detail of the structure to be built. Chemists can build models of molecules, engineers can construct scale models, and plastic surgeons can use the computer to show potential clients exactly how their faces will look after surgery. Diagrams can be drawn to reflect the structure of silicon chips and other electronic components; the components of a computer can be represented by means of various sorts of schematics, at various levels of abstraction.

Certainly, there are ways in which software engineers can represent specific views of their product. For example, a software engineer can draw one directed graph depicting flow of control, another showing flow of data, a third with patterns of dependency, and a fourth depicting time sequences. Also, UML diagrams (Chapters 12 and 13) have proven to be a powerful tool for depicting views of large-scale software. The problem is that few of these graphs are planar, let alone hierarchical. The many crossovers in these graphs are a distinct obstacle to understanding. Parnas suggests cutting the arcs of the graphs until one or more becomes hierarchical [Parnas, 1979]. The problem is that the resulting graph, though comprehensible, makes only a subset of the software visualizable and the arcs that have been cut may be critical from the viewpoint of comprehending the interrelationships among the components of the software.

The result of this inability to represent software visually not only makes software difficult to comprehend, it also severely hinders communication among software professionals—there seems to be no alternative to handing a colleague a 200-page listing together with a list of modifications to be made.

It must be pointed out that visualizations of all kinds, such as flowcharts, data flow diagrams (Section 11.3.1), module interconnection diagrams, or UML diagrams (Chapters 12 and 13) are extremely useful and powerful ways of visualizing certain aspects of the product. Visual representations are an excellent means of communicating with the client as well as with other software engineers. The problem is that such diagrams cannot embody *every* aspect of the product, nor is there a way to determine what is missing from any one visual representation of the product.

2.9.5 No Silver Bullet?

Brooks's article [Brooks, 1986] by no means is totally gloom filled. He describes what he considers to be the three major breakthroughs in software technology: high-level languages, time sharing, and software development environments (such as the UNIX Programmer's Workbench), but stresses that they solved only accidental, and not essential, difficulties. He evaluates various technical developments currently advanced as potential silver bullets, including proofs of correctness (Section 6.5), object-oriented design (Section 13.6), Ada, and artificial intelligence and expert systems. Although some of these approaches may solve remaining accidental difficulties, Brooks feels they are irrelevant to the essential difficulties.

To achieve comparable future breakthroughs, Brooks suggests that we change the way software is produced. For example, whenever possible, software products should be bought off the shelf (that is, COTS software) rather than custom built. For Brooks, the hard part of building software lies in the requirements, specification, and design phases—not in the implementation phase. The use of rapid prototyping (Section 10.3) for him is a major potential source of an order-of-magnitude improvement. Another suggestion that, in Brooks's opinion, may lead to a major improvement in productivity is greater use of the principle of incremental development, where instead of trying to build the product as a whole, it is constructed stepwise. This concept is described in Section 5.1.

For Brooks, the greatest hope of a major breakthrough in improving software production lies in training and encouraging great designers. As stated previously, in Brooks's opinion, one of the hardest aspects of software production is the design phase, and to get great designs, we need great designers. Brooks cites UNIX, APL, Pascal, Modula-2, Smalltalk, and FORTRAN as exciting products of the past. He points out that they all have been the products of one, or a very few, great minds. On the other hand, more prosaic but useful products like COBOL, PL/I, ALGOL, MVS/360, and MS-DOS have been products of committees. Nurturing great designers for Brooks is the most important objective if we wish to improve software production.

Parts of Brooks's paper make depressing reading. After all, from the title onward, he states that the inherent nature (or essence) of current software production makes finding a silver bullet a dubious possibility. Nevertheless, he concludes on a note of hope, suggesting that, if we change our software production strategies by buying ready-made software wherever possible, using rapid prototyping and incremental building techniques, and attempting to nurture great designers, we may increase software productivity. However, an order-of-magnitude breakthrough, the "silver bullet," is most unlikely.

Brooks's pessimism must be put into perspective. Over the past 20 years, the software industry has shown a steady increase in productivity of roughly 6 percent per year. This productivity increase is comparable to what has been observed in many manufacturing industries. What Brooks is looking for, though, is a "silver bullet," a way of rapidly obtaining an order-of-magnitude increase in productivity. It is difficult to disagree with his view that we cannot hope to double productivity overnight. At the same time, the compound growth rate of 6 percent means that productivity is doubling every 12 years. This improvement may not be as rapid and spectacular as we would like, but the software engineering process is improving steadily from year to year.

The remainder of this chapter is devoted to national and international initiatives aimed at process improvement.

2.10 Improving the Software Process

Our global economy is critically dependent on computers and hence on software. For this reason, the governments of many countries are concerned about the software process. For example, in 1987 a task force of the U.S. Department of Defense (DoD) reported: "After two decades of largely unfulfilled promises about productivity and quality gains from applying new software methodologies and technologies, industry and government organizations are realizing that their fundamental problem is the inability to manage the software process" [DoD, 1987].

In response to this and related concerns, DoD founded the Software Engineering Institute (SEI) and set it up at Carnegie Mellon University in Pittsburgh on the basis of a competitive procurement process. One major success of the SEI has been the capability maturity model (CMM) initiative. Related software process improvement efforts include the ISO 9000-series standards of the International Standards Organization, and ISO/IEC 15504, an international software improvement initiative involving more than 40 countries. We begin by describing CMM.

2.11 Capability Maturity Models

The *capability maturity models* of the SEI are a related group of strategies for improving the software process, irrespective of the actual life-cycle model used. (The term *maturity* is a measure of the goodness of the process itself.) The SEI has developed CMMs for software (SW–CMM), for management of human resources (P–CMM; the *P* stands for "people"), for systems engineering (SE–CMM), for integrated product development (IPD– CMM), and for software acquisition (SA–CMM). There are some inconsistencies between the models and a inevitable level of redundancy. Accordingly, in 1997, it was decided to develop a single integrated framework for maturity models, capability maturity model integration (CMMI). Four of these five existing capability maturity models have been integrated into CMMI, and SA–CMM is due to be incorporated later. Additional disciplines may be added in the future [SEI, 2000].