CHAPTER 8
Myths of Correctness

29-CENT DISCREPANCY
GENERATES ERRONEOUS
$161 BILLION CREDIT

The headline summarized a filler item—two column-inches buried within a recent edition of *The Washington Post.* It was the usual story: a department store charged someone too much; the customer complained and was given a credit, but the amount was 29 cents short; the customer complained again and this time received a welcome $161 billion credit. I grinned, clipped the story for my "computer errors" file, and read on. Only later did I notice that the story didn't mention a computer.

We're all quick to blame computers. We do it when we have billing problems, credit problems, delivery problems, reservation problems, and information problems. And if we don't blame the computer, someone else does—usually in a defensive voice over a phone line.

Does the computer deserve all this blame? Literally speaking, of course not. If a machine can ever be said to have caused a problem, the machine is at most a victim of circumstances—people designed the machine, people built it, people chose to use it, and people controlled it. When a car crashes and causes serious damage, we may blame the driver, we may blame some other person at the scene, and we may blame the manufacturer, but we don't seriously blame the car.

Such linguistic distinctions aside, I'm convinced that "the computer" often gets blamed when in fact it was either not present at all or present only as an innocent bystander. When we don't get around to doing something or don't want to bother trying, it's easy to say "the computer goofed" or "the computer's down." In a recent newspaper column about these modern excuses, Ellen Goodman cited a friend's comment: "The computer is down" is another way to spell "coffee break." As a modern scapegoat, what could be more convenient and convincing? And satisfying—it's a bit like finding out that your parents are human. Equipped with considerable means, having frequent opportunities, and exempt from the requirement of motive, the computer is an ideal villain. Moreover, when we resolve some mystery by saying, plausibly, "The computer did it," the computer can't defend itself. It is, to use Ellen Goodman's words, the silent butler.

Wrongly accused as it may often be, the computer is responsible for plenty of true horror stories. Indeed, pointing out its occasional and perhaps considerable innocence is a little like absolving Lizzie Borden from an ax murder she happened not to commit. In discussing computer foulups, however, it's important to make those linguistic distinctions that I mentioned before. When we "blame" a computer for some problem, we're really saying either that a hardware component stopped working properly or that someone goofed. That someone could be a hardware designer, a hardware builder, a programmer, or a user.

Hardware components do fail, but modern hardware is remarkably reliable; in a well-established computer system, hardware failures are a relatively infrequent cause of problems. And hardware designs do contain flaws, particularly when they're new designs. But it's fair to say that hardware design flaws are also a relatively infrequent cause of problems, especially in well-established systems. When a computer crashes, chances are that someone tripped it, either a programmer or a user.

I've already discussed how easy it is for novice users to cause problems, especially when confronted with inadequate user-interfaces. Veterans have similar problems, but they often downplay the importance of user-interfaces and blame their computer problems on software bugs; the bugs, however, are not always at fault. Recently I tracked down a variety of stories about computer problems that supposedly were caused by software bugs, and a surprising number of them turned out to be caused by incorrect program operation rather than incorrect programming. You may recall talk of a computer problem during the final moments of the first moon landing—the computer became overloaded and the software responded by restarting itself several times. It turned out that an astronaut had obeyed an incorrect checklist and left a radar switch in the wrong position, thereby generating 13 percent more load on the computer than had been anticipated. This was not the only such problem. Moreover, the astronauts proved themselves capable of making mistakes without being told to do so. Indeed, one study of the computer problems that were encountered during the Apollo space-flight program concluded that about 75 percent of the problems were caused by operator errors. Translated roughly into computer-speak, this means that the Apollo software wasn't user-friendly.

The right stuff, it appears, is no substitute for a good user-interface. Unfortunately, while a good user-interface can protect us from our own errors, we still have to suffer from the other main cause of computer problems: programming errors. And what a cause it is—most large computer programs, even well-established ones, are plagued with bugs.

The Software Crisis

It sounds so melodramatic—the kind of phrase that might be coined by the news media and snickered at by software experts. To the contrary, the phrase arose in the software community ten to fifteen years ago, and it appears regularly in academic textbooks. Indeed, the increasingly apparent difficulty of writing large, reliable computer programs led not only to the term "software crisis," but to a whole field of study, called **software engineering.** Within the computer industry today, the software crisis is widely recognized and widely battled, except by a few who accept it as a fixture along with cancer, venereal disease, and budget deficits.

When people refer to the software crisis, they have in mind several trends. While hardware costs have plummeted, software costs have not. Ten years ago, hardware made up 80 percent of the total cost of a typical large computer system. Today, software makes up 80 percent of the total cost, and this dominant cost can not be predicted well. On top of these financial problems, software usually takes longer to finish than was promised; and when it is finished, it's bigger, slower, and less capable than was promised. These deficiencies might be bearable if the resulting software were reliable, but it isn't. It tends to fail often, and efforts to fix it are just as costly and error-prone as its original development. Moreover, software is hard to change—efforts to improve software capabilities are even more costly and error-prone. While these problems tend to be most obvious in large software systems built for the Department of Defense, they are well known in commercial systems as well. There are, of course, exceptions, but the general trends are clear and widely recognized.

Of all the problems inherent in the software crisis, the most difficult, troublesome, and dangerous is our inability to write error-free software. Two examples may help to make this point.

An Election Night Story

Jimmy Carter conceded the 1980 election before the polls closed in California. His pollster Pat Cadell had warned him that he was going to lose, and the TV networks—using computer predictions based on early returns—declared Reagan the winner early in the evening. Shortly thereafter Carter made it official. West Coast citizens felt cheated. And Democratic candidates on the West Coast, especially those who lost the election, were furious. They believed that the early concession had kept Democratic voters at home and Democrats out of office. Someone even introduced a bill in the U.S. Congress that would make it illegal to concede before all the polls were closed.

It's a tough issue. It's hard to make a strong case that Carter's concession changed the outcome of any particular West Coast election—the races just weren't that close; besides, didn't Republicans stay home too? But the events were still disturbing. The technology of time zones, telecommunications, and computers might have distorted the will of the people. Moreover, this could have happened even if Carter hadn't conceded. After all, people still would have watched TV. They still would have seen those checkmarks next to Reagan's name on the network tote-boards, and they still would have heard well-known correspondents reporting on and interpreting a computer's landslide projections.

These events provide yet another example of our tendency to trust technology. People question how technology is used in elections, they question its effects, but they rarely question its validity. The election process is affected by instant information—information that's gathered, analyzed, and distributed faster than the sun sets. People question the effects of this instant information, but they don't question the information itself.

They should. Consider what happened during the 1981 provincial election in Quebec, Canada. The two main parties in Quebec are the Parti Quebeçois (PQ) and the Liberal Party. There is also the Union Nationale, a small splinter party, and the Marxist-Leninist party, which has only a few hundred members. On election night two TV stations provided coverage. Viewers of one station saw unsurprising results—the Liberals and the PQ led almost all the races (Canadians call them ridings), and the station declared at 8:45 P.M. that PQ had won a majority government.

Things were different on the other station, as well as on a cooperating radio network. The Union Nationale, having been given essentially no chance by anyone, was leading 19 ridings, and a Marxist-Leninist candidate was leading one. The PQ was leading 20 ridings, and the Liberals only 9. The results were astounding, and the commentators lost no time in explaining them. One person dismissed sarcastically "the so-called experts and commentators who had written off the Union Nationale"; the experts were wrong—"the people have spoken."

This apparent upset by the Union Nationale was not only astounding, it was wrong. Software bugs were attributing votes to the wrong candidates, and the TV station went right along. Eventually they admitted the mistake—twenty minutes after the other station had declared a PQ majority—and they subsequently filed a million dollar lawsuit against the company that had assisted them in producing the faulty software. The postmortems were merciless—a columnist in *La Presse* cried "shame . . . dishonor . . . humiliation." A *Montreal Gazette* columnist likened the election night show to *Monty Python's Flying Circus*, citing "general agreement that the election night show was 'the greatest fiasco' in Quebec TV history."

Irony On Board the Space Shuttle

One of the best examples of the software crisis is carried on board every U.S. Space Shuttle Orbiter. NASA has stringent and effective reliability policies. As a result, all of the critical systems in the Orbiter are redundant in one way or another. In the case of data processing, the redundancy is achieved by five identical computers. Four of them are arranged as a voting group—during critical flight periods, all four run exactly the same program and compare their results. Any computer that disagrees with the others is immediately switched out, a procedure that protects the Orbiter against computer hardware malfunctions. The fifth computer operates more or less independently, executing a different program, written by a different contractor. This program provides a backup flight-control system for use if all four voting computers fail.

All four voting computers could fail simultaneously if all four coincidently suffered hardware failures or if there was a fatal bug in the program that they all run. But the possibility of simultaneous hardware failures is not the reason for having the fifth computer run a different program—if it were, more overall reliability could be achieved by having all five computers run the same program while voting. Rather, the existence of independent backup software is motivated by the possibility of a fatal bug in the primary flight software. It's an expensive precaution—by the time the space shuttle program is over, millions of dollars will have been spent on that backup program. Those millions will be spent because NASA recognized that it couldn't develop error-free flight software. That backup computer is a running symbol of the software crisis.

As it happens, the backup computer was involved in a famous bug. While the backup computer operates more or less independently, it is in fact connected to the other four. It "listens in" on them so that it has up-to-date information in case it has to take over. When the backup flight software begins running on the backup computer, it has to synchronize with the primary flight software before it can begin to listen in. This is a bit like "tuning in" to a dance rhythm before trying to follow a dancing partner, which can be tricky. It's particularly tricky for the Orbiter software because the operating systems for the primary and backup flight software exercise control according to different philosophies. The primary software is asynchronous or **priority-driven,** which means that it pays attention to tasks on demand and in accordance with announced importance—it acts like teachers in a progressive day care center who interrupt what they're doing in order to pay attention to the loudest screaming child. The backup flight software is synchronous or **time-slotted,** which means that it pays equal-length attention to tasks in a fixed sequence. It acts like the stern moderator of a debate among eight political candidates, disallowing interruptions and giving equal time to each candidate in turn regardless of how much they have to say or how loud they say it.

Under these circumstances, arranging for the backup software to "tune in" resulted in a bug. As far as bugs go, it was relatively minor, in the sense that it was relatively easy to fix. But you can't fix a bug that you don't know about. This particular bug was extraordinarily subtle, and it remained unknown for a long time. When it finally transpired, it became perhaps the most widely known bug in history, but until then it lurked unknown, waiting for an unlikely but possible series of coincidences. Finally they occurred, not during testing, but at T minus 20 minutes during the countdown of the space shuttle's first scheduled launch, when the backup flight software was turned on. It turned on, but it didn't tune in. The launch was dropped for the day.

The bug didn't prevent the primary flight software from working properly. And the bug wouldn't have prevented the backup flight software from working properly, had the backup software been able to tune in. The bug arose from combining the primary software with backup software that exists only in case the primary software doesn't work. How ironic. And how illustrative of the difficulties inherent in software engineering.

The election night story and the space shuttle story are just two examples of the software crisis, but they're not exceptions. Both of these programs are almost certain to have additional bugs, and most other large computer programs are also erroneous. This fact surprises many people, especially since it's common to hear that a computer program has passed from "development" into "maintenance."

The Meaning of Maintenance

Hardly a week goes by without our being treated to a prediction of how the onslaught of computers will affect employment. It's easy to get the impression that soon we'll all be programming computers or serving fast food. Such predictions are naive, unrealistic, and myopic, but secretly I relish them because they put me in the category of those who can choose what to do. You might think that programming would win hands-down, but there are some distinct advantages to serving fast food. When you slap together a hamburger, for example, at least you know when you're done. Not so with software development—permit me the following distortion of a well-known saying:

A fast food job lasts from bun to bun,
But a programming job is never done.

It's obvious that a programming job can be a long one, especially if the object is to develop a large software system, but many people don't realize just how long it can take. One reason for this is the term "software maintenance"—a common linguistic obfuscation that is itself an amusing symptom of the software crisis.

When I mentioned at the beginning of this chapter that we don't seriously blame a car when it crashes, I left out one legitimate target for blame: insufficient or faulty maintenance. Some car parts need occasional attention to keep them working properly, and others need occasional replacement. If these needs are neglected or improperly met, a crash may ensue.

Can insufficient or faulty maintenance be blamed for a software crash? No, at least not in the same sense. Cars and other physical products need maintenance because they wear out with use. But software has no physical properties, only logical properties. When you run the same program a thousand times, some of the computer hardware components may wear and require maintenance, but the software can't wear out. If I made Aunt Martl's Sachertorte a thousand times, I might wear out my kitchen, myself, and my friends, and I might require a hundred fresh copies of the recipe, but the recipe itself couldn't wear out.

A program can work properly a thousand times and fail suddenly the next time. It might thereby give the appearance of having worn out, but what really happened is that an unusual set of circumstances was encountered for the first time. A program doesn't fail because you wear it out; it fails because it didn't work properly to begin with and you finally have occasion to notice that fact. Software is one of the few products of engineering that can truly be said to last forever. This is a marketing dream; unfortunately, few software products are good enough to take advantage of it.

Although the term "software maintenance" is inappropriate, it's common to hear it. Indeed, when most customers buy a large program, they usually enter into a maintenance contract. In return for a monthly or yearly payment, the customer receives occasional revisions of the program. The revisions contain fixes for some of the bugs that were in the original product and perhaps some improvements. The maintenance contract may also provide access to consultants who can help customers work around bugs that they encounter. To a large extent "maintenance" is a software euphemism for "continued development." This language doesn't fool anyone, but it's a convenient fiction. It allows software developers to declare, in analogy with equipment contracts, that a software product has been "developed" and is ready to enter service and be "maintained." If software products were never accepted until they were error-free, few companies could ever finish their development contracts.

If you buy a new car and spend the next year having the dealer fix things that didn't work right to begin with, you don't say that your car is being maintained; you say you bought a lemon. By this criterion, most software products are lemons.

Why Is Software So Hard?

If a car manufacturer produces mostly lemons, we judge the manufacturer to be incompetent, a judgment supported by the existence of many well-built cars. But there is not much well-built software anywhere, and from the bitter taste of a software product, it's unfair to conclude that the builder is incompetent. In general, we know more about building cars well than we know about building software well. What makes software so hard? The short answer is "complexity," but the question deserves a slightly longer answer.

The Inadequacy of Testing

A common reaction to the story about the space shuttle bug is surprise that the bug wasn't caught during testing. Doesn't NASA conduct exhaustive testing before they try to shoot off a rocket with some men in it? Are the software-testing personnel incompetent, the testing procedures inadequate?

The answers to the last questions are "no" and "yes": No, the software-testing personnel are *not* incompetent; and yes, the software-test procedures *are* inadequate. The inadequacy of software testing is not the result of incompetence. No matter how competent you are, it's impossible to expose all of the bugs in a program by means of exhaustive testing.

This fact does not apply just to large-scale software. You hold a more down-to-earth example in the palm of your hand every time you use an electronic calculator. These calculators are popular and convenient, and they usually display correct results. But not always. Indeed, most popular calculators display wrong answers some of the time. An American mathematician and computer scientist, William Kahan, has documented, studied, and helped to correct such problems for years. Here's one of his examples, a compound interest problem for financial calculators. The problem is called "A Penny for Your Thoughts."

A bank retains a legal consultant whose thoughts are so valuable that she is paid for them at the rate of a penny per second, day and night. Lest the sound of pennies dropping distract her, they are deposited into her account to accrete with interest at the rate of 10% per annum compounded every second. How much will have accumulated after a year (365 days)?

You can solve such problems with financial calculators just by pressing a few keys. In this case Kahan did so using ten different popular calculator models. He got four different answers:

$331,667.00
$293,539.00
$334,858.18
$331,559.30

In calculator software as in space shuttle software, errors can result from bugs that lead directly to wrong answers or from the gradual buildup of inaccuracies in repetitive calculations. But, whatever the cause of the errors, it isn't possible to find all of them by testing.

To see why, let's suppose that you've accepted a job with Calcutronics—a new company that makes electronic calculators. Your first assignment is to evaluate the prototype of a new model, a standard calculator that handles eight-digit numbers, and see if it works correctly. Like other electronic calculators, the Calcutronics model is really a small computer. When you press a key to multiply, divide, or compute compound interest, your action invokes a program stored in read only memory (ROM). Even the addition key initiates a program; the CPU does have an ADD instruction, but it only adds one-digit numbers. The addition of two eight-digit numbers is accomplished by a program.

How should you go about evaluating the calculator? One approach is simply to try every operation and see if the calculator gives the correct results. This approach is called **black box testing**, probably because black boxes are opaque and their contents have to be evaluated by means of externally observable characteristics. Because black box testing focuses on the calculator's actual behavior and ignores its internal design, it seems to be the least biased and most reliable way to proceed. Unfortunately, you would never finish.

Consider just the addition function. To test it exhaustively, you would have to add every possible combination of eight-digit numbers. Depending on how you like to see it written, there are 10^8, or 100,000,000, or one hundred million different eight-digit numbers. As for the number of different combinations of two eight-digit numbers, there are 10^{16}, or 10,000,000,000,000,000, or ten million billion of them. So to test addition, you would have to add ten million billion pairs of numbers. You might be tempted to cut this in half since, for example, $8+2=2+8$. But $8+2$ and $2+8$ are not the same operations from the calculator's point of view. The difference is just the order in which you enter the two numbers, but that might result in their being handled differently, so you have to try both cases.

How long the complete test would take depends on how fast you work, but it's doubtful that you could test more than one addition per second. If you did one per second, forty hours per week, year round, it would take about 1,300 million years to finish, give or take a few million years. You might be able to save time by building a machine that worked around the clock, punching in numbers and photographing the answers for you to check during the day. Such a machine would also take about one second per operation, but its constant attention to the task would reduce the total time a lot. With the machine, you could finish in about 320 million years.

You could bypass the mechanical operation of the keys, connect a computer directly to the electronics inside the calculator, and use the computer to try all the different combinations. But you still wouldn't be done in a reasonable amount of time. Even if the computer could test one million additions per second (faster than most calculators would allow), the total time would be about 320 years. Job security, perhaps, but not exactly what Calcutronics was looking for.

It might seem reasonable just to test the program with selected input values scattered throughout the overall range and conclude, if the program works properly for these values, that it also works properly for every value in between. This approach could be valid if a large number of input values were tried, provided that the addition process is continuous—i.e., provided that a small change in the inputs corresponds to a small change in the output. But we can not assume continuity. For example, a program can easily treat one particular input value in a totally different manner than it treats nearby values—all it takes is a single **if-then** statement. And if you view the program as a black box, you can't rule out the possibility of such **if-then** exceptions being inside. Just because the result of

```
32000 + 767
```

is correct doesn't mean that the result of

```
32001 + 767
```

is likely to be correct.

The calculator's addition program had only two eight-digit inputs, yet the total number of input possibilities made black box testing impossible. It wasn't even close. Black box testing would be exhausting but not exhaustive—there just isn't enough time. Furthermore, most programs have even more input possibilities. The conclusion is inescapable: black box testing is hopelessly inadequate as the basis for any thorough evaluation of computer program correctness.

This conclusion and the arguments that lead to it are summarized succinctly by a remark made by the Dutch computer scientist Edsger W. Dijkstra. It is probably the most often-quoted statement in the computer science literature:

> Program testing can be used to show the presence of bugs, but never to show their absence!

No matter how diligently you test computer programs, you cannot test them completely. The point is really that simple. And that profound.

That program testing is inadequate doesn't mean that we should eschew it—not to test a program before depending on it would be foolish. But the inadequacy of testing does mean that the role of testing is corroborative rather that definitive—testing can corroborate our belief in the correctness of a program, but we must have other evidence. Where can that other evidence come from?

If black-box testing is inadequate, the only alternative is to open the box and examine what goes on inside. It follows that evidence for a program's correctness must somehow involve not just running the program, but studying the program itself. Stated differently, our belief in the correctness of a computer program must arise substantially from intellectual arguments based on the program's written text. So far, so good. Unfortunately, if we cannot understand the program's text, we certainly cannot argue effectively about its correctness, and therein lies a major problem.

The Curse of Flexibility

Fluency in English doesn't guarantee that you can understand every document that's written in English. It all depends on the clarity of the writing. Not so long ago, I struggled with a particularly unpleasant tax form. Neither my educational background nor my ego encouraged me to give up, but after three hours I did. Unfortunately I still had to fill out the form, so I called the special phone number that the IRS maintains for citizens who need help. Eventually I spoke to a real, live IRS agent and explained my problem. Almost immediately, he laughed. "Oh that form! Listen, nobody understands that form. Here's what you do. . . ."

The situation with many computer programs is similar. A few years ago some colleagues and I embarked on a software-engineering project in which we proposed to demonstrate various modern methods in a practical way by rebuilding the obsolescent flight program of a naval aircraft. In the preface to our first report on the project, we mentioned that the existing software, among its other faults, "is not fully understood by the maintenance personnel." This statement got us into trouble— not with the maintenance personnel, who agreed with us emphatically, but with their project sponsors, who happened also to be our sponsors. I never regretted the statement, however. It describes accurately the program in question, and many other programs as well.

There are many reasons for programs being hard to understand, but at the root is the mixed blessing of flexibility. A computer's behavior can be changed radically by changes to its software. Whether a programmer wants to fix a bug, change an existing function, or add a new function, it's easy to bend the software accordingly. In principle, this flexibility is a blessing; major changes can be accomplished quickly and at low cost. But the blessing is hard to receive. Because it's easy to make changes quickly and without considering all the ramifications, software complexity can grow quickly, leading to software that's hard to read, hard to understand, likely to contain more errors, and likely to require further modifications. These results are hard to avoid, and their effects can be crippling. In this light, the computer's flexibility looks less like a blessing and more like a curse.

The computer's flexibility is unique. No other kind of machine can be changed so much without physical modifications. Moreover, drastic modifications are as easy to make as minor ones, which is unfortunate, since drastic modifications are more likely to cause problems. With other kinds of machines, drastic modifications are correspondingly harder to make than minor ones. This fact provides natural constraints to modification that are absent in the case of computer software. Such natural constraints bring discipline to machine design and construction as well as to machine modifications. In the case of airplane construction, for example, feasible designs are governed by the mechanical limitations of design materials and by the laws of aerodynamics. There is a resulting, nature-imposed discipline to the design process that helps to control complexity. In the case of software construction there are no such limitations or natural laws. This makes it too easy to build enormously complex software. Indeed, the structure of typical software systems makes the humorous complexities of Rube Goldberg's fanciful machines look elegant in comparison.

The flexibility of software also encourages the redefinition of tasks quickly, often, and late in the development process. Imagine that General Motors is 90 percent finished with a new car development; deliveries are scheduled to begin in a few months. Suddenly it's decided to have the driver control the car from the back seat and to include a separate air conditioning system for the trunk. It's a laughable scenario, but analogous ones are common in software development. Software is the resting place of afterthoughts.

Software's flexibility is deceptive and seductive. It encourages programmers to plunge in, and they tend to do so; premature construction is a common software problem. Few would be foolish enough to begin building a large airplane before the designers finished detailed plans. In the case of large computer programs, however, few are wise enough to wait. Because the software medium appears to be so forgiving, it encourages us to begin working with it too soon; we begin our attempts before we understand our tasks. It's extremely hard to build a large computer program that works correctly under all required conditions, but it's easy to build one that works 90 percent of the time. It's also hard to build reliable airplanes, but it's not particularly easy to build an airplane that flies 90 percent of the time.

With software, it's easy to start out and hard to finish. Total success is difficult because the flexibility of software facilitates partial success at the expense of unmanaged complexity. And once a program's complexity has become unmanageable, each change becomes as likely to hurt as it is to help. Each new feature may interfere with several old features. Each attempt to fix a bug may create several more. The feeling of "one step forward, two steps back" is a common one. The programmer facing poorly understood, overly complex software is like Brer Rabbit facing the Tar Baby.

Like airplane complexity, software complexity can be controlled by an appropriate design discipline. But to reap this benefit, people have to impose that discipline; nature won't do it. As the name implies, computer software exploits a "soft" medium, with intrinsic flexibility that is both its strength and its weakness. Offering so much freedom and so few constraints, computer software has all the advantages of free verse over sonnets; and all the disadvantages.

Invisible Interfaces

A common and sensible approach to writing large programs is to divide the problem into parts and write a separate program for each part. Unfortunately, to do so is simple in principle but surprisingly difficult in practice. Much of the difficulty arises because the separate programs must interact to solve the overall problem—some programs have to exchange information with other programs, and some programs have to control other programs. These interactions take place across various interfaces that exist among the programs.

Consider once again the magic-trick example. If you enter '0' (zero) in response to the prompt

```
PLEASE ENTER A NUMBER BETWEEN 1 AND 10:,
```

the magic trick fails, and the program stops with the explanation

```
FATAL ERROR . . . REGISTER OVERFLOW AT AF45
712 547 234 232
777 234 342 455
209 487 439 332
>
```

This error can be viewed as the programmer's fault for not having checked the number that was entered before using it as a divisor. But a subtler explanation is possible if more than one programmer is involved. Suppose that the magic-trick problem were divided into two parts, and that there were two programmers—one responsible for a program that obtains a number from the user, the other responsible for a program that performs the calculations involved in the magic trick. Not knowing that the input value would end up as a divisor, the first programmer may have assumed that the range 1 to 10 was a suggestion rather than a requirement. Meanwhile, the second programmer knew that the input value would end up as a divisor, but assumed that the first programmer wouldn't permit anything outside of the required range 1 to 10. The fatal error arose because the two programmers made conflicting assumptions about the number that was to be passed from one program to the other. This error is a small example of an interface-related bug.

A larger and classic example is provided by the space shuttle bug. Two separate programs—the primary flight software and the backup flight software—were written in order to solve the overall problem of flying the Orbiter reliably. Considered separately, they both worked (at least from the viewpoint of the bug in question), but a bug in their interface prevented the backup flight software from starting up properly.

Here's another example of an interface-related bug, one that I encountered on my Lisa. There are two Lisa programs that I've been using regularly. One is the word-processing program that I'm using to write this book; the other is a telecommunications program that allows me to use the Lisa as a remote terminal connected by telephone to computers elsewhere. Although these are separate programs, they can be used together—you can switch back and forth from one to the other and you can also transfer information from one to the other. For example, you might switch from the telecommunications program to the word-processing program, pick up two or three paragraphs from a word-processing document stored on the Lisa, switch back to the telecommunications program, and transmit those paragraphs to the remote computer. The two programs worked together well until I needed to transmit not just a few paragraphs, but an entire copy of a long document. I successfully transmitted the text in the prescribed manner, but when I switched back to the word-processing program, it crashed.

The crash was inconvenient and somewhat damaging; I lost the most recent changes I'd made to the document I was working on. The crash, however, was graceful, and in this respect it exemplified a good user-interface. For example, the crash was accompanied by a series of error messages that were user-oriented instead of programmer-oriented. The messages spoke of "technical difficulties" (a term that covers a multitude of sins but is generally accurate), they spoke of attempts to recover my document, and they warned me that I might have lost the changes that indeed were lost. And when the crash was finally over, only the word-processing program was affected—I didn't have to restart the Lisa's operating system, and the telecommunications program was still connected properly to the remote computer.

The cause of the crash was a bug in the interface between the word-processing program and the telecommunications program. The two programs made conflicting assumptions about the use of the Lisa's main memory—when I used the word-processing and telecommunication programs separately, the problem didn't arise because the programs didn't have to share the available memory with each other. But when I used them together, they did have to share. It appears that this fact was overlooked somewhere in the telecommunications program. When I transmitted a copy of the long document, the telecommunications program had to use more memory, and it ended up using a portion of the memory that was already being used by the word-processing program. The resulting interference caused the word-processing program to crash.

The more complicated an interface, the more likely it is that something will fall through a crack. Software interfaces are so error-prone because it's so easy to build complicated interfaces. The curse of flexibility strikes again—it's easy to make practically anything depend on practically anything else. Moreover, the dependencies can be subtle, and they're almost impossible to detect by studying the programs involved. For example, one program might work properly only if another program can be relied on to finish its job in a specific amount of time. Indeed, the shuttle bug originated from just such an assumption. About a year before the first attempted launch, a change was made to the primary flight software that caused certain operations to take longer. This violated a subtle assumption that was implicit in the backup flight software, but not stated explicitly anywhere. The resulting bug eventually showed up on launch day. This example shows the need for a broader definition of "interface": namely, a software **interface** between two programs comprises all of the assumptions that the programs make about each other.

Like software, physical machines such as cars and airplanes are built by dividing the design problems into parts and building a separate unit for each part. The spatial separation of the resulting parts has several advantages: It limits their interactions, it makes their interactions relatively easy to trace, and it makes new interactions difficult to introduce. If I want to modify a car so that the loudness of its horn depends on the car's speed, it can be done, at least in principle. And if I want the car's air conditioner to adjust automatically according to the amount of weight present in the back seat, that too can be done—again, in principle. But in practice such changes are hard to make, so they require careful design and detailed planning. The interfaces in hardware systems, from airplanes to computer circuits, tend to be simpler than those in software systems because physical constraints discourage complicated interfaces. The costs are immediate and obvious. No comparable constraints operate in software systems. Indeed, the medium encourages complicated interfaces. The total costs of a software interface, like the costs of a hardware interface, grow quickly with complexity. But the costs of a software interface are neither immediate nor obvious.

Interface related problems are common in any computer program, but their prevalence grows rapidly with the size of the program. In computer programs that are designed and built by a large team of programmers, all sorts of subtle assumptions are made, and there's plenty of opportunity for conflict. This is just one example of the problems that beset programmers when they attempt to scale up their previous successes.

Scaling Up Is Hard to Do

When I was a boy I liked to build model airplanes and fly them at a park near my house. And it wasn't so long ago that a friend and I put together a twelve-inch-long rocket and fired it off gleefully. As small as these achievements were, I'm proud of them. But I've never attempted to build jumbo jets or space shuttles.

Such attempts would be absurd. Yet analogous attempts are common in the case of computer software. Programmers are constantly acting as if the skill and effort required to build small computer programs can be scaled up easily to build large programs. They're wrong.

I'm as guilty as anyone. When I first went to work at the Naval Research Laboratory, I was asked whether I had any experience with simulation programs. Indeed, I had. As part of a political science project that I did while a college senior, I had written a FORTRAN program that simulated the international arms race. I was quite impressed with this project, for which I recall receiving an "A," and I felt confident that I could handle the assignment that was proposed for me: managing the development of a computer program that would simulate in detail the electronic systems aboard large naval ships. That my previous experience consisted of a small program embodying the straightforward computation of a trivial equation gave me only slight pause. After all, the principles were the same—only the size of the program was different.

I accepted the assignment along with someone else's estimate of what it would take: around $250,000 and two years. Both the money and the time seemed excessive to me, but I thought it prudent not to object. Soon thereafter I engaged a willing contractor, and within six months or so I began receiving design documents and programs. Some months after that, it became obvious that the project was in serious trouble. I appealed for more money, which I received and passed on to the contractor, who assigned more people to the task. The trouble got worse.

What I needed then was more knowledge and not more money. Among other things, I wish I had known about Brook's Law:

> Adding manpower to a late software project makes it later.

This is the high-tech equivalent of "too many cooks spoil the broth." It was first stated—at least in this eloquent form—by the American computer scientist Frederick P. Brooks, Jr., a man who should know. Fred Brooks has been described as "the father of the IBM System/360," a series of computers that was IBM's main line from the middle 1960s to the early 1970s. Brooks served as project manager during the development of the system, and he directed the development of OS/360, its software operating system. OS/360 was, by previous standards, a huge undertaking. The project had now-classic problems with time, effort, money, and reliability, and it has become a famous symbol of the software crisis. As Brooks said about just one of his decisions,

> It is a very humbling experience to make a multi-million-dollar mistake, but it is also very memorable.

Like the problems of my simulation project, the problems of OS/360 and countless other projects arose in large part because the people involved underestimated the difficulty of scaling up their previous efforts. Brooks put it this way:

> The second system is the most dangerous system a man ever designs.

He called this the **second-system effect**, and the term has caught on.

Brooks was right when he said that "all programmers are optimists," but where does this optimism come from? There is, I think, a basic human urge to accomplish more. In engineering we see this as an urge to scale up previous results. We see it in airplanes, bridges, and buildings. We also see it in software. But the urge to scale up the size or performance of physical objects is tempered by natural and obvious impediments that require technological improvements in making and handling building materials. No matter how much you may want to build a Boeing 747 instead of a Piper Cub, a Saturn Booster instead of a V2 rocket, or the Brooklyn Bridge instead of a stream crossing, the difficulties and the costs of mistakes are sufficiently obvious to serve as effective restraints.

With computer software, it's different. The obvious limitations to performance are the computer's speed and its storage capacity; if both are doubled, the goal of doing twice as much looks easy to achieve with software. But the appearance is deceptive, because the process is highly nonlinear. Even if twice as much is possible when you double a computer's capacity, it becomes much more than twice as hard to succeed.

Scaled-up software is not only harder to produce, it's harder to maintain. Clearly written software is important generally because of the inadequacy of testing. But the importance of clarity grows as software is scaled up because large programs that are used over a period of many years tend to be maintained by programmers other than those who wrote the program to begin with. When you maintain a program you wrote yourself, your understanding of its operation is assisted by your memory. When you maintain a program that someone else wrote, your understanding must rely solely on the program's text.

Increased computer capacity attracts the programmer like a charm, but its effects are tempered by the curse of flexibility. Faster computers and larger memories are natural catalysts for the production of complexity. To manage that complexity, we need appropriate tools—something that's analogous to the materials handling technology used in building physical objects. Such tools are themselves made out of software. Examples include compilers for programming languages, and various other programs that help us to write, analyze, and test software—generically such programs are called **support software.**

Support software helps in managing complexity, but programmers also need help in avoiding excessive complexity in the first place—they need something analogous to the discipline imposed by the natural constraints on building physical objects. Here we must, in the jargon of modern society, resort to "self-help." We must impose our own constraints, our own discipline. We can do so by means of appropriate programming languages, programming methods, and documentation methods. Complexity can't be eliminated, but it can be managed; the software engineer's job is to manage unavoidable complexity and avoid unmanageable complexity.

Advances in electronics technology have made small personal computers inexpensive and widely available. But although their components are modern, their capacities and their software tools often reflect twenty-year-old technology. Formative exposure to these machines has some potential for harm. If people are uninformed and also gain their computer experience by using twenty-year-old languages to write small programs on these small machines, they will learn nothing about those aspects of computers that led to the software crisis, and they may well be predisposed to follow in the intellectual footsteps of their predecessors. Of course, those who go on to bigger machines and bigger programs will soon learn about the problems I've been discussing, as will those who are educated in other ways. The rest, however, stand to be misled profoundly about modern computer technology.

The Myth of Incorrectness

Of all the potential effects of the personal computer craze, the one I find most troublesome concerns attitudes about software correctness. When people start writing their own bug-ridden programs, their experience may temper their harsh reactions to the computer foul-ups around them. People know that to err is human, and they may believe that programming errors are just another example of human fallibility. This is a myth. But it's a myth that will be reinforced if people are exposed to such nonsense as I found recently in the introductory manual of a popular personal computer:

"Bugs," in computerese, refer mostly to imperfections in software. These bugs are minor flaws that the people who wrote the program couldn't foresee.

Statements like this breed familiarity with bugs, but not contempt. Bugs are indeed flaws, but not all bugs are "minor flaws"—some of them can kill people. Moreover, it's not true that people *couldn't* foresee the bugs; what's true is that people *didn't* foresee the bugs.

Programmers do make mistakes, but today's software problems are not statistical evidence for human fallibility, they are the unavoidable results of the programming languages and methods that we use. The complexity of the computer programs we write has grown faster than our ability to write them correctly. This mismatch is one of the most important and difficult technical challenges of our time.