# Why Software Is So Bad

**By Charles C. Mann** July/August 2002

**For years we've tolerated buggy, bloated, badly organized computer programs. But soon, we'll innovate, litigate and regulate them into reliability.**

It's one of the oldest jokes on the Internet, endlessly forwarded from e-mailbox to e-mailbox. A software mogul—usually Bill Gates, but sometimes another—makes a speech. "If the automobile industry had developed like the software industry," the mogul proclaims, "we would all be driving $25 cars that get 1,000 miles to the gallon." To which an automobile executive retorts, "Yeah, and if cars were like software, they would crash twice a day for no reason, and when you called for service, they'd tell you to reinstall the engine."

The joke encapsulates one of the great puzzles of contemporary technology. In an amazingly short time, software has become critical to almost every aspect of modern life. From bank vaults to city stoplights, from telephone networks to DVD players, from automobile air bags to air traffic control systems, the world around us is regulated by code. Yet much software simply doesn't work reliably: ask anyone who has watched a computer screen flush blue, wiping out hours of effort. All too often, software engineers say, code is bloated, ugly, inefficient and poorly designed; even when programs do function correctly, users find them too hard to understand. Groaning beneath the weight of bricklike manuals, bookstore shelves across the nation testify to the perduring dysfunctionality of software.

"Software's simply terrible today," says Watts S. Humphrey, a fellow of Carnegie Mellon University's Software Engineering Institute who has written several well-known books on software quality. "And it's getting worse all the time." Good software, in Humphrey's view, "is usable, reliable, defect free, cost effective and maintainable. And software now is none of those things. You can't take something out of the box and know it's going to work." Over the years, in the view of Edsger W. Dijkstra, an emeritus computer scientist at the University of Texas at Austin, the average computer user "has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed."

Jim McCarthy is more generous. The founder, with his wife Michele, of a software quality training company in Woodinville, WA, McCarthy believes that "most software products have the necessary features to be worth buying and using and adopting." But, he allows, "only the extreme usefulness of software lets us tolerate its huge deficiencies." McCarthy sometimes begins talks at his school with a PowerPoint presentation. The first slide reads, "Most Software Sucks."

It is difficult to overemphasize the uniqueness of software's problems. When automotive engineers discuss the cars on the market, they don't say that vehicles today are no better than they were ten or fifteen years ago. The same is true for aeronautical engineers: nobody claims that Boeing or Airbus makes lousy planes. Nor do electrical engineers complain that chips and circuitry aren't improving. As the engineering historian Henry Petroski suggested in his 1992 book *The Evolution of Useful Things*, continual refinement is the usual rule in technology. Engineers constantly notice shortcomings in their designs and fix them little by little, a process Petroski wryly described as "form follows failure." As a result, products incrementally improve.

Software, alas, seems different. One would expect a 45-million-line program like Windows XP, Microsoft's newest operating system, to have a few bugs. And software engineering is a newer discipline than mechanical or electrical engineering; the first real programs were created only 50 years ago. But what's surprising—astonishing, in fact—is that many software engineers believe that software quality is not improving. If anything, they say, it's getting worse. It's as if the cars Detroit produced in 2002 were less reliable than those built in 1982.

As software becomes increasingly important, the potential impact of bad code will increase to match, in the view of Peter G. Neumann, a computer scientist at SRI International, a private R&D center in Menlo Park, CA. In the last 15 years alone, software defects have wrecked a European satellite launch, delayed the opening of the hugely expensive Denver airport for a year, destroyed a NASA Mars mission, killed four marines in a helicopter crash, induced a U.S. Navy ship to destroy a civilian airliner, and shut down ambulance systems in London, leading to as many as 30 deaths. And because of our growing dependence on the Net, Neumann says, "We're much worse off than we were five years ago. The risks are worse and the defenses are not as good. We're going backwards—and that's a scary thing."

Some software companies are responding to these criticisms by revamping their procedures; Microsoft, stung by charges that its products are buggy, is publicly leading the way. Yet problems with software quality have endured so long, and seem so intractably embedded in software culture, that some coders are beginning to think the unthinkable. To their own amazement, these people have found themselves wondering if the real problem with software is that not enough lawyers are involved.

**A Lack of Logic**

Microsoft released Windows XP on Oct. 25, 2001. That same day, in what may be a record, the company posted *18 megabytes* of patches on its Web site: bug fixes, compatibility updates, and enhancements. Two patches fixed important security holes. Or rather, one of them did; the other patch didn't work. Microsoft advised (and still advises) users to back up critical files before installing the patches. Buyers of the home version of Windows XP, however, discovered that the system provided no way to restore these backup files if things went awry. As Microsoft's online Knowledge Base blandly explained, the special backup floppy disks created by Windows XP Home "do not work with Windows XP Home."

Such slip-ups, critics say, are merely surface lapses—signs that the software's developers were too rushed or too careless to fix obvious defects. The real problems lie in software's basic design, according to R. A. Downes of Radsoft, a software consulting firm. Or rather, its *lack* of design. Microsoft's popular Visual Studio programming software is an example, to Downes's way of thinking. Simply placing the cursor over the Visual Studio window, Downes has found, invisibly barrages the central processing unit with thousands of unnecessary messages, even though the program is not doing anything. "It's cataclysmic….It's total chaos," he complains.

The issue, in the view of Dan Wallach, a computer scientist at Rice University, is not the pointless churning of the processor—after all, he notes, "processing power is cheap." Nor is Microsoft software especially flawed; critics often employ the company's products as examples more because they are familiar than because they are unusually bad. Instead, in Wallach's view, the blooming, buzzing confusion in Visual Studio and so many other programs betrays how the techniques for writing software have failed to keep up with the explosive increase in its complexity.

Programmers write code in languages such as Java, C and C++, which can be read by human beings. Specialized programs known as "compilers" transform this code into the strings of ones and zeroes used by computers. Importantly, compilers refuse to compile code with obvious problems—they spit out error messages instead. Until the 1970s, compilers sat on large mainframes that were often booked days or weeks in advance. Not wanting errors to cause delay, coders—who in the early days tended to be trained as mathematicians or physicists—stayed late in their offices exhaustively checking their work. Writing software was much like writing scientific papers. Rigor, documentation and peer-review vetting were the custom.

But as computers became widespread, attitudes changed. Instead of meticulously planning code, programmers stayed up in caffeinated all-night hacking sessions, constantly bouncing results off the compiler. Again and again, the compiler would spit back error messages; the programmers would fix the mistakes one by one until the software compiled properly. "The attitude today is that you can write any sloppy piece of code and the compiler will run

diagnostics," says SRI's Neumann. "If it doesn't spit out an error message, it must be done correctly, right?"

As programs grew in size and complexity, however, the limits of this "code and fix" approach became evident. On average, professional coders make 100 to 150 errors in every thousand lines of code they write, according to a multiyear study of 13,000 programs by Humphrey of Carnegie Mellon. Using Humphrey's figures, the business operating system Windows NT 4, with its 16 million lines of code, would thus have been written with about two million mistakes. Most would have been too small to have any effect, but some—many thousands—would have caused serious problems.

Naturally, Microsoft exhaustively tested NT 4 before release, but "in almost any phase of tests you'll find less than half the defects," Humphrey says. If Microsoft had gone through four rounds of testing, an expensive and time-consuming procedure, the company would have found at most 15 out of 16 bugs. "That's going to leave you with something like five defects per thousand lines of code," Humphrey says. "Which is very low"—but the software would still have as many as 80,000 errors.

Software engineers know that their code is often riddled with lacunae, and they have long been searching for new technologies to prevent them. To manage increasingly distended projects like Windows, for example, they have developed a variety of techniques, of which perhaps the best known is component-based design. Just as houses are built with standardized two-by-fours and electrical fittings, component-based programs are built out of modular, interchangeable elements: an example is the nearly identical menu bar atop every Windows or Macintosh program. Such standardized components, according to Wallach, are not only good engineering practice, they are "the only way you can make something the size of Microsoft Office work at all." Microsoft, he says, was an early, aggressive promoter of this approach—"it's the single best engineering decision they ever made."

Unfortunately, critics say, the components are often glued together with no real central plan—as if contractors tried to erect large structures with no blueprints. Incredibly, Humphrey says, the design for large software projects is sometimes "nothing but a couple bubbles on the back of an envelope." Worse, for marketing reasons companies wire as many features as possible into new software, counteracting the benefits of modular construction. The most widespread example is Windows itself, which Bill Gates testified in an April session of the Microsoft antitrust trial simply would not function if customers removed individual components such as browsers, file managers or e-mail programs. "That's an incredible claim," says Neumann. "It means there's no structure or architecture or rhyme or reason in the way they've built those systems, other than to make them as bundled as possible, so that if you remove any part it will all fail."

The inadequate design in the final products, critics argue, reflects inadequate planning in the process of creating them. According to a study by the Standish Group, a consulting firm in West Yarmouth, MA, U.S. commercial software projects are so poorly planned and managed that in 2000 almost a quarter were canceled outright, creating no final product. The canceled projects cost firms $67 billion; overruns on other projects racked up another $21 billion. But because "code and fix" leads to such extensive, costly rounds of testing, even successful projects can be wildly inefficient. Incredibly, software projects often devote *80 percent* of their budgets to repairing flaws they themselves produced—a figure that does not include the even more costly process of furnishing product support and developing patches for problems found after release.

"System testing goes on for almost half the process," Humphrey says. And even when "they finally get it to work, there's still no design." In consequence, the software can't be updated or improved with any assurance that the updates or improvements won't introduce major faults. "That's the way software is designed and built everywhere—it's that way in spaceships, for God's sake."

**Is Software a Special Case?**

The potential risks of bad software were grimly illustrated between 1985 and 1987, when a computer-controlled radiation therapy machine manufactured by the government-backed Atomic Energy of Canada massively overdosed patients in the United States and Canada, killing at least three. In an exhaustive examination, Nancy Leveson, now an MIT computer scientist, assigned much of the blame to the manufacturer's inadequate software-engineering practices. Because the program used to set radiation intensity was not designed or tested carefully, simple typing errors triggered lethal blasts.

Despite this tragic experience, similar machines running software made by Multidata Systems International, of St. Louis, massively overdosed patients in Panama in 2000 and 2001, leading to eight more deaths. A team from the International Atomic Energy Agency attributed the deaths to "the entering of data" in a way programmers had not anticipated. As Leveson notes, simple data-entry errors should not have lethal consequences. So this failure, too, may be due to inadequate software.

Programming experts tend to agree that such disasters are distressingly common. Consider the Mars Climate Orbiter and the Polar Lander, both destroyed in 1999 by familiar, readily prevented coding errors. But some argue that software simply cannot be judged, measured and improved in the same way as other engineering products. "It's just a fact that there are things that other engineers can do that we can't do," says Shari Pfleeger, a senior researcher at the Rand think tank in Washington, DC, and author of the 1998 volume *Software Engineering: Theory and Practice*. If a bridge survives a 500-kilogram weight and a 50,000-kilogram weight, Pfleeger notes, engineers can assume that it will bear all the values between. With software, she says, "I can't make that assumption—I can't interpolate."

Moreover, software makers labor under extraordinary demands. Ford and General Motors have been manufacturing the same product—a four-wheeled box with an internal-combustion engine—for decades. In consequence, says Charles H. Connell, former principal engineer of Lotus Development (now part of IBM), they have been able to improve their products incrementally. But software companies are constantly asked to create products—Web browsers in the early 1990s, new cell phone interfaces today—unlike anything seen before. "It's like a car manufacturer saying, 'This year we're going to make a rocket ship instead of a car,'" Connell says. "Of course they'll have problems."

"The classic dilemma in software is that people continually want more and more and more stuff," says Nathan Myhrvold, former chief technology officer of Microsoft. Unfortunately, he notes, the constant demand for novelty means that software is always "in the bleeding-edge phase," when products are inherently less reliable. In 1983, he says, Microsoft Word had only 27,000 lines of code. "Trouble is, it didn't do very much"—which customers today wouldn't accept. If Microsoft had not kept pumping up Word with new features, the product would no longer exist.

"Users are tremendously non-self-aware," Myhrvold adds. At Microsoft, he says, corporate customers often demanded that the company simultaneously add new features and stop adding new features. "Literally, I've heard it in a single breath, a single sentence. 'We're not sure why we should upgrade to this new release—it has all this stuff we don't want—and when are you going to put in these three things?' And you say, 'Whaaat?'" Myhrvold's sardonic summary: "Software sucks because users demand it to."

**Higher Standards**

In January, Bill Gates issued a call to Microsoft employees to make "reliable and secure" computing their "highest priority." In what the company billed as one of its most important initiatives in years, Gates demanded that Microsoft "dramatically reduce" the number of defects in its products. A month later, the company took the unprecedented step of suspending all new code writing for almost two months. Instead, it gathered together programmers, a thousand at a time, for mass training sessions on reliability and security. Using huge screens in a giant auditorium, company executives displayed embarrassing snippets of flawed code produced by those in the audience.

.Gates's initiative was apparently inspired by the blast of criticism that engulfed Microsoft in July 2001 when a buffer overflow—a long-familiar type of error—in its Internet Information Services Web-server software let the Code Red worm victimize thousands of its corporate clients. (In a buffer overflow, a program receives more data than expected—as if one filled in the space for a zip code with a 50-digit number. In a computer, the extra information will spill into adjacent parts of memory, corrupting or overwriting the data there, unless it is carefully blocked.) Two months later, the Nimda worm exploited other flaws in the software to attack thousands more machines.

Battered by such experiences, software developers are becoming more attentive to quality. Even as Gates was rallying his troops, think tanks like the Kestrel Institute, of Palo Alto, CA, were developing "correct-by-construction" programming tool kits that almost force coders to write reliable programs (*see "First Aid for Faulty Code"* ). At Microsoft itself, according to Amitabh Srivastava, head of the firm's Programmer Productivity Research Center, coders are working with new, "higher-level" languages like C# that don't permit certain errors. And in May, Microsoft cofounded the $30 million Sustainable Computing Consortium—based at Carnegie Mellon—with NASA and 16 other firms to promote standardized ways to measure and improve software dependability. Quality control efforts can pay off handsomely: in helping Lockheed Martin revamp the software in its C130J aircraft, Praxis Critical Systems, of Bath, England, used such methods to cut development costs by 80 percent while producing software that passed stringent Federal Aviation Administration exams with "very few errors."

Critics welcome calls for excellence like those from Kestrel and Microsoft but argue that they will come to naught unless commercial software developers abandon many of their ingrained practices. "The mindset of the industry is to treat quality as secondary," says Cem Kaner, a computer scientist and lawyer at the Florida Institute of Technology. Before releasing products, companies routinely hold "bug deferral meetings" to decide which defects to fix immediately, which to fix later by forcing customers to download patches or buy upgrades, and which to forget about entirely. "Other industries get sued when they ignore known defects," Kaner says. "In software, it's standard practice. That's why you don't buy version 1.0 of a program." Exasperatingly, software vendors deliver buggy, badly designed products with incomprehensible help files—and then charge high fees for the inevitable customer service calls. In this way, amazingly, firms profit from poor engineering practices.

When engineers inside a software company choose to ignore a serious flaw, there are usually plenty of reviewers, pundits, hackers and other outsiders who will point it out. This is a good thing; as Petroski wrote in *The Evolution of Useful Things*, "a technologically savvy and understanding public is the best check on errant design." Unfortunately, companies increasingly try to discourage such public discussion. The fine print in many software licenses forbids publishing benchmark tests. When *PC Magazine* tried in 1999 to run a head-to-head comparison of Oracle and Microsoft databases, Oracle used the license terms to block it—even though the magazine had gone out of its way to assure a fair test by asking both firms to help it set up their software. To purchase Network Associates' popular McAfee VirusScan software, customers must promise not to publish reviews without prior consent from Network Associates—a condition so onerous that the State of New York sued the firm in February for creating an "illegal…restrictive covenant" that "chills free speech." (At press time, no trial date had been set.)

Even a few members of the software-is-different school believe that some programming practices must be reformed. "We don't learn from our mistakes," says Rand's Pfleeger. In 1996, for example, the French Ariane 5 rocket catastrophically

## First Aid for Faulty Code

Software quality has been so bad for so long, some engineers argue, that the only solutions are litigation and legislation. More optimistic observers believe that industry is slowly beginning to adopt new practices and technological tools for making better software. Among them:

**Perspective-based review.** Engineers who develop code don't look at software in the same way as the system administrators who maintain it, the marketers who sell it, or the end users who put it to work. Yet, says Steve McConnell of Construx Software, development teams rarely account for these diverse perspectives. Involving colleagues like business managers, administrators, customer support agents and user interface experts in software design meetings "is obvious when you think of it, but hardly used at all," McConnell says.

**Shared vision.** Incredibly, the purpose of new software is often not clearly spelled out before programmers begin writing it. Indeed, it often changes in midstream as marketers come up with wish lists, with predictably bad results. According to software quality trainers Jim and Michele McCarthy, one of the keys to improving software is for all parties to reach an agreement in advance on what they're doing—"a single, explicit, universally accepted focus."

**Correct by construction.** Some languages, such as Ada, are designed so that programmers simply cannot commit certain mistakes. Under the Kestrel Institute's "correct by construction" approach, programmers carefully design and assemble software modules using special programming tools that prevent errors such as buffer overflows. Similarly, recent improvements in Java compilers have helped automate the process of weeding out common problems in Java code.

**Tracking revisions.** According to Amitabh Srivastava of Microsoft, improvements will also come from new tools that meticulously tally changes in software code, allowing testers to focus on heavily rewritten sections that may contain more errors. These and other similar changes, he says, will reverse the now prevalent approach of slapping components together by inspiration in offices full of pizza boxes and Mountain Dew.

failed, exploding just 40 seconds after liftoff on its maiden voyage. Its $500 million satellite payload was a total loss. According to the subsequent committee of inquiry, the accident was due to "systematic software design error"—more precisely, a buffer overflow. In most engineering fields, Pfleeger says, such disasters trigger industrywide reforms, as the collapse of the World Trade Center seems likely to do for fireproofing in construction. But in software, "there is no well-defined mechanism for investigating failures and no mechanism for ensuring that people read about them." If the French coders had been drilled, like other engineers, in the history of their own discipline, the Ariane fiasco might have been avoided.

One way or another, some computer scientists predict, software culture will change. To the surprise of many observers, the industry is relatively free of product liability lawsuits. The "I Love You" virus, for instance, spread largely because Microsoft—against the vehement warnings of security experts—designed Outlook to run programs in e-mail attachments easily. According to Computer Economics, a consulting group in Carlsbad, CA, the total cost of this decision was $8.75 billion. "It's amazing that there wasn't a blizzard of lawsuits," Wallach says.

Software firms have been able to avoid product liability litigation partly because software licenses force customers into arbitration, often on unfavorable terms, and partly because such lawsuits would be highly technical, which means that plaintiffs would need to hire costly experts to build their cases. Nonetheless, critics predict, the lawsuits will eventually come. And when the costs of litigation go up enough, companies will be motivated to bulletproof their code. The downside of quality enforcement through class action lawsuits, of course, is that groundless litigation can extort undeserved settlements. But as Wallach says, "it just might be a bad idea whose time has come."

In fact, a growing number of software engineers believe that computers have become so essential to daily life that society will eventually be unwilling to keep giving software firms a free legal pass. "It's either going to be a big product liability suit, or the government will come in and regulate the industry," says Jeffrey Voas, chief scientist of Cigital Labs, a software-testing firm in Dulles, VA. "Something's going to give. It won't be pretty, but once companies have a gun to their head, they'll figure out a way to improve their code."

Charles C. Mann has written for *Technology Review* about the free software movement (January/February 1999) and the use of genetic engineering in agriculture (July/August 1999).