The Software Wars

Why You Can't Understand Your Computer

Paul De Palma

On a bright winter morning in Philadelphia, in 1986, my downtown office is bathed in sunlight. I am the lead programmer for a software system that my firm intends to sell to the largest companies in the country, but like so many systems, mine will never make it to market. This will not surprise me. If the chief architect of the office tower on whose twenty-sixth floor I am sitting designed his structure with the seat-of-the-pants cleverness that I am using to design my system, prudence would advise that I pack my business-issue briefcase, put on my business-issue overcoat, say good-bye to all that sunlight, and head for the front door before the building crumbles like a Turkish high-rise in an earthquake.

But I am not prudent; nor am I paid to be. Just the opposite. My body, on automatic pilot, deflects nearly all external stimuli. I can carry on a rudimentary conversation, but my mind is somewhere else altogether. In a book-length profile of Ted Taylor, a nuclear-weapons designer, that John McPhee wrote for *The New Yorker*, Dr. Taylor's wife tells McPhee a wonderful story about her husband. Mrs. Taylor's sister visits for the weekend. Taylor dines with her, passes her in the hall, converses. He asks his wife on Monday morning—her sister having left the day before—when she expects her sister to arrive. Mrs. Taylor calls this state "metaphysical absence." You don't have to build sophisticated weaponry to experience it. When my daughter was younger, she used to mimic an old John Prine song. "Oh my stars," she sang, "Daddy's gone to Mars." As you will see, we workaday programmers have more in common with weapons designers than mere metaphysical absence.

My mind reels back from Mars when a colleague tells me that the *Challenger* has exploded. The *Challenger*, dream child of NASA, complex in the extreme, designed and built by some of the country's most highly trained engineers, is light-years away from my large, and largely uninspired, piece of data-processing software. If engineering were music, the *Challenger* would be a Bach fugue and my system "Home on the Range." Yet despite the differences in technical sophistication, the software I am building will fail for many of the same reasons that caused the *Challenger* to explode seconds after launch nearly twenty years ago.

Software's unreliability is the stuff of legend. *Software Engineering Notes*, a journal published by the ACM, the largest professional association of computer scientists, is known mostly for the tongue-in-cheek catalogue of technical catastrophes that appears at the beginning of each issue. In the March 2001 issue—I picked this off my shelf at random—you can read about planes grounded in L.A. because a Mexican air-traffic controller keyed in too many characters of flight description data, about a New York database built to find uninsured drivers, which snared many of the insured as well, about Florida eighth graders who penetrated their school's computer system, about Norwegian trains that refused to run on January 1, 2001, because of a faulty Year 2000 repair. The list goes on for seven pages and is typical of a column that has been running for many years.

People often claim that one of every three large-scale software systems gets canceled midproject. Of those that do make it out the door, three-quarters are never implemented: some do not work as intended; others are just shelved.

People often claim that one of every three large-scale software systems gets canceled midproject. Of those that do make it out the door, three-quarters are never implemented: some do not work as intended; others are just shelved. Matters grow even more serious with large systems whose functions spread over several computers—the very systems that advances in networking technology have made possible in the past decade. A few years ago, an IBM consulting group determined that of twenty-four companies surveyed, 55 percent built systems that were over budget; 68 percent built systems that were behind schedule; and 88 percent of the completed systems had to be redesigned. Try to imagine the same kind of gloomy numbers for civil engineering: three-quarters of all bridges carrying loads below specification; almost nine of ten sewage treatment plants, once completed, in need of redesign; one-third of highway projects canceled because technical problems have grown beyond the capacity of engineers to solve them. Silly? Yes. Programming has miles to go before it earns the title "software engineering."

In civil engineering, on the other hand, failures are rare enough to make the news. Perhaps the best-known example is the collapse of the Tacoma-Narrows Bridge. Its spectacular failure in 1940, because of wind-induced resonance, was captured on film and has been a staple of physics courses ever since. The collapse of the suspended walkway in the Kansas City Hyatt Regency in 1981 is a more recent example. It failed because structural engineers thought that verifying the design of connections joining the walkway segments was the job of their manufacturer. The manufacturer had a different recollection. The American Society of Civil Engineers quickly adopted a protocol for checking shop designs. These collapses are remarkable for two related reasons. First, bridge and building failures are so rare in the United States that when they do occur we continue to talk about them half a century later. Second, in both cases, engineers correctly determined the errors and took steps not to repeat them. Programmers cannot make a similar claim. Even if the cause of system failure is discovered, programmers can do little more than try not to repeat the error in future systems. Trying not to repeat an error does not compare with building well-known tolerances into a design or establishing communications protocols among well-defined players. One is exhortation. The other is engineering.

None of this is new. Responding to reports of unusable systems, cost overruns, and outright cancellations, the NATO Science Committee convened a meeting of scientists, industry leaders, and programmers in 1968. The term *software engineering* was invented at this conference in the hope that, one day, systematic, quantifiable approaches to software construction would develop. Over the intervening years, researchers have created a rich set of tools and techniques, from design practices to improved programming languages to techniques for proving program correctness. Sadly, anyone who uses computers knows that they continue to fail regularly, inexplicably, and, sometimes, wonderfully—*Software Engineering Notes* continues to publish pages of gloomy tales each quarter. Worse, the ACM has recently decided not to support efforts to license software engineers because, in its words, "there is no form of licensing that can be instituted today assuring public safety." In effect, software-engineering discoveries of the past thirty years may be interesting, but no evidence suggests that understanding them will improve the software-development process.

As the committee that made this decision surely knows, software-engineering techniques are honored mostly in the breach. In other words, business practice, as much as a lack of technical know-how, produces the depressing statistics I have cited. One business practice in particular ought to be understood. The characteristics of software often cited as leading to failure—its complexity, its utter plasticity, its free-floating nature, unhampered by tethers to the physical world—make it oddly, even paradoxically, similar to the practice of military procurement. Here is where the *Challenger* and my system, dead these twenty long years, reenter the story.

In the mid-eighties I worked for a large management-consulting firm. Though this company had long employed a small group of programmers, mostly to support in-house systems, its software-development effort and support staff grew substantially, perhaps by a factor of ten, over a period of just a few years. A consulting firm, like a law firm, has a cap on its profits. Since it earns money by selling time, the number of hours its consultants can bill limits its revenue. And there is a ceiling to that. They have to eat and sleep, after all. The promise of software is the promise of making something from nothing. After development, only the number of systems that can be sold limits return on investment. In figuring productivity, the denominator remains constant. Forget about unhappy unions, as with cars and steel; messy sweatshops, as with clothing and shoes; environmental regulations, as with oil and petrochemicals. Software is a manufacturer's dream. The one problem, a very sticky problem indeed, is that it does not wear out. The industry responds by adding features, moving commercial software one step closer to military systems. More on this later. For now, just understand that my company, like so many others under the influence of the extraordinary attention that newly introduced personal computers were receiving at the time, followed the lure of software.

My system had one foot on the shrinking terra firma of large computers and the other in the roiling, rising sea of microcomputers. In fact, mine was the kind of system that three or four years earlier would have been written in COBOL, the language of business systems. It perhaps would have used a now obsolete database design, and it would have gone to market within a year. When told to build a similar system for a microcomputer, I did what I knew how to do. I designed a gray flannel system for a changing microcomputer market.

Things went along in a predictable if uninspired way until there was a shift in management. These changes occur so frequently in business that I had learned to ignore them. The routine goes like this. Everyone gets a new organization chart. They gather in conference rooms for mandatory pep talks. Then life goes on pretty much as before. Every so often, though, management decisions percolate down to the geeks, as when your manager arrives with a security officer and gives you five minutes to empty your desk, unpin your *Dilbert* comics, and go home. Or when someone like Mark takes over.

When that happened, I assumed falsely that we would go back to the task of producing dreary software. But this was the eighties. Junk bonds and leveraged buyouts were in the news. The arbitrageur was king. Business had become sexy. Mark, rumor had it, slept three hours a night. He shuttled between offices in New York, Philadelphia, and Montreal. Though he owned a house in Westchester County, now best known as the home of the Clintons, he kept an apartment in Philadelphia, where he managed to spend a couple of days each week. When Mark, the quintessential new manager ("My door is always open"), arrived, we began to live like our betters in law and finance. Great bags of bagels and cream cheese arrived each morning. We lunched in trendy restaurants. I, an erstwhile sixties radical, began to ride around in taxis, use my expense account, fly to distant cities for two-hour meetings. Life was sweet.

During this time, my daughter was an infant. Her 4:00 A.M. feeding was my job. Since I often had trouble getting back to sleep, I sometimes caught an early train to the office. One of these mornings my office phone rang. It was Mark. He sounded relaxed, as if finding me at work before dawn was no more surprising than bumping into a neighbor choosing apples at Safeway. This was a sign. Others followed. Once, Mark organized a dinner for our team in a classy hotel. When the time came for his speech, Mark's voice rose like Caesar's exhorting his troops before the Gallic campaign. He urged us to bid farewell to our wives and children. We would, he assured us, return in six months with our shields or upon them. I noticed then that a few of my colleagues were in evening dress. I felt like Tiresias among the crows. When programmers wear tuxedos, the world is out of joint.

Suddenly, as if by magic, we went from a handful of programmers producing a conventional system to triple that number, and the system was anything but conventional. One thing that changed was the programming language itself. Mark decided that the system would be splashier if it used a database-management system that had recently become commercially available for mainframes and was promised, shortly, for microcomputers. These decisions—hiring more people to meet a now unmeetable deadline; using a set of new and untested tools—represented two of the several business practices that have been at the heart of the software crisis. Frederick Brooks, in his classic book, *The Mythical Man-Month,* argues from his experience building IBM's System 360 operating system that any increased productivity achieved by hiring more people gets nibbled at by the increased complexity of communication among them. A system that one person can develop in thirty days cannot be developed in a single day by thirty people. This simple truth goes down hard in business culture, which takes, as an article of faith, the idea that systems can be decreed into existence.

The other practice, relying on new, untested, and wildly complex tools, is where software reigns supreme. Here, the tool was a relational database-management system. Since the late sixties, researchers have realized that keeping all data in a central repository, a database, with its own set of access techniques and backup mechanisms, was better than storing data with the program that used it. Before the development of database-management systems, it was common for every department in a company to have its own data, and for much of this data to overlap from department to department. So in a university, the registrar's office, which takes care of student records, and the controller's office, which takes care of student accounts, might both have copies of a student's name and address. The problem occurs when the student moves and the change has to be reported to two offices. The argument works less well for small amounts of data accessed by a single user, exactly the kind of application that the primitive microcomputers of the time were able to handle. Still, you could argue that a relational database-management system might be useful for small offices. This is exactly what Microsoft Access does. But Microsoft Access did not exist in 1986, nor did any other relational database-management system for microcomputers. Such systems had only recently become available for mainframes.

Something unique to software, especially new software: no experts exist in the sense that we might speak of an expert machinist, a master electrician, or an experienced civil engineer. There are only those who are relatively less ignorant.

One company, however, an infant builder of database-management systems, had such software for minicomputers and was promising a PC version. After weeks of meetings, after an endless parade of consultants, after trips to Washington, D.C., to attend seminars, Mark decided to go with the new product. One of these meetings illustrates something unique to software, especially new software: no experts exist in the sense that we might speak of an expert machinist, a master electrician, or an experienced civil engineer. There are only those who are relatively less ignorant. On an early spring evening, we met in a conference room with a long, polished wood table surrounded by fancy swivel chairs covered in gorgeous, deep purple fabric. The room's walls turned crimson from the setting sun. As the evening wore on, we could look across the street to another tower, its offices filled with miniature Bartlebys, bent over desks, staring into monitors, leafing through file cabinets. At the table with representatives from our company were several consultants from the database firm and an independent consultant Mark had hired to make sure we were getting the straight scoop.

Here we were: a management-consulting team with the best, though still less than perfect, grasp of what the proposed system was supposed to do, but almost no grasp of the tools being chosen; consultants who knew the tools quite well, but nothing about the software application itself, who were fully aware that their software was still being developed even as we spoke; and an independent consultant who did not understand either the software or its application. It was a perfect example of interdependent parasitism.

My company's sin went beyond working with complex, poorly understood tools. Neither the tools nor our system existed. The database manufacturer had a delivery date and no product. Their consultants were selling us a nonexistent system. To make their deadline, I am confident they hired more programmers and experimented with unproven software from still other companies with delivery dates but no products. And what of *those* companies? You get the idea.

No one in our group had any experience with this software once we adopted it. Large systems are fabulously complex. It takes years to know their idiosyncrasies. Since the introduction of the microcomputer, however, nobody has had years to develop this expertise. Because software does not wear out, vendors must consistently add new features in order to recoup development costs. That the word processor you use today bears almost no resemblance to the one you used ten years ago has less to do with technological advances than with economic realities. Our company had recently acquired a smaller I company in the South. This company owned a mini computer for which a version of the database software had already been released. Mark decided that until the PC database was ready for release, we could develop our system on this machine, using 1,200-baud modems, a modem about one-fiftieth as fast as the one your cable provider tells you is too slow for the Web, and a whole lot less reliable.

Let me put this all together. We had a new team of programmers who did not understand the application, using ersatz software that they also did not understand, which was running on a kind of machine no one had ever used before, using a remote connection that was slow and unstable.

Weeks before, I had begun arguing that we could never meet the deadline and that none of us had the foggiest idea of how to go about building a system with the tools we had. This was bad form. I had been working in large corporations long enough to know that when the boss asks if something can be done, the only possible response is "I'm your boy." Business is not a Quaker meeting. Mark didn't get to be my boss by achieving consensus. I knew that arguing was a mistake, but somehow the more I argued, the more I became gripped by a self-righteous fervor that, while unattractive in anyone (who likes a do-gooder?), is suicide in a corporate setting. Can-do-ism is the core belief. My job was to figure out how to extend the deadline, simplify the requirements, or both—not second-guess Mark. One afternoon I was asked if I might like to step down as chief architect and take over the documentation group. This was not a promotion.

Sitting in my new cubicle with a Raskolnikovian cloud over my head, I began to look more closely at the database-management system's documentation. Working with yet another consultant, I filled a paper database with hypothetical data. What I discovered caused me to win the argument but lose the war. I learned that given the size of the software itself and the amount of data the average client would store, along with the overhead that comes with a sophisticated database, a running system would fill a microcomputer hard disk, then limited to 30 megabytes, several times over. If, by some stroke of luck, some effort of will, some happy set of coincidences that I had yet to experience personally, we were able to build the system, the client would run up against hardware constraints as soon as he tried to use it. After weeks of argument, my prestige was slipping fast. I had already been reduced to writing manuals for a system I had designed. I was the sinking ship that every clearheaded corporate sailor had already abandoned. My triumphant revelation that we could not build a workable system, even if we had the skill to do so, was greeted with (what else?) complete silence.

Late in 1986 James Fallows wrote an article analyzing the *Challenger* explosion for the *New York Review of Books.* Instead of concentrating on the well-known O-ring problem, he situated the failure of the *Challenger* in the context of military procurement, specifically in the military's inordinate fondness for complex systems. This fondness leads to stunning cost overruns, unanticipated complexity, and regular failures. It leads to Osprey aircraft that fall from the sky, to anti-missile missiles for which decoys are easy to construct, to FA-22 fighters that are fabulously over budget. The litany goes on. What these failures have in common with the *Challenger* is, Fallows argues, "military procurement disease," namely, "over-ambitious schedules, problems born of too-complex design, shortages of spare parts, a 'can-do' attitude that stifles embarrassing truths ('No problem, Mr. President, we can lick those Viet Cong'), and total collapse when one component unexpectedly fails." Explanations for this phenomenon include competition among the services; a monopoly hold by defense contractors who are building, say, aircraft or submarines; lavish defense budgets that isolate military purchases from normal market mechanisms; the nature of capital-intensive, laptop warfare where hypothetical justifications need not—usually cannot—be verified in practice; and a little-boy fascination with things that fly and explode. Much of this describes the software industry too.

Fallows breaks down military procurement into five stages:

The Vegematic Promise, wherein we are offered hybrid aircraft, part helicopter, part airplane, or software that has more features than could be learned in a lifetime of diligent study. Think Microsoft Office here.

The Rosy Prospect, wherein we are assured that all is going well. I call this the 90 percent syndrome. I don't think I have ever supervised a project, either as a software manager overseeing professionals or as a professor overseeing students, that was not 90 percent complete whenever I asked.

The Big Technical Leap, wherein we learn that our system will take us to regions not yet visited, and we will build it using tools not yet developed. So the shuttle's solid-fuel boosters were more powerful than any previously developed boosters, and bringing it all back home, my system was to use a database we had never used before, running on a computer for which a version of that software did not yet exist.

The Unpleasant Surprise, wherein we learn something unforeseen and, if we are unlucky, calamitous. Thus, the shuttle's heat-resistant dies, all 31,000 of them, had to be installed at the unexpected rate of 2.8 days per tile, and my system gobbled so much disk space that there was scarcely any room for data.

The House of Cards, wherein an unpleasant surprise, or two, or three, causes the entire system to collapse. The Germans flanked the Maginot Line, and in my case, once we learned that our reliance on a promised database package outstripped operating-system limits, the choices were: one, wait for advances in operating systems; two, admit a mistake, beg for forgiveness, and resolve to be more prudent in the future; or, three, push on until management pulls the plug.

In our case, the first choice was out of the question. We were up against a deadline. No one knew when, or if, the 30 MB disk limit would be broken. The second choice was just as bad. The peaceable kingdom will be upon us, the lamb will lie down with the lion, long before you'll find a hard-driving manager admitting an error. These guys get paid for their testosterone, and for men sufficiently endowed, in the famous words of former NASA flight director Gene Kranz, "failure is not an option." We were left with the third alternative, which is what happened. Our project was canceled. Inside the fun house of corporate decision making, Mark was promoted—sent off to manage a growing branch in the South. The programmers left or were reassigned. The consultant who gave me the figures for my calculations was fired for reasons that I never understood. I took advantage of my new job as documentation chief and wrote an application to graduate school in computer science. I spent the next few years, while a student, as a well-paid consultant to our firm.

Just what is it about software, even the most conventional, the most mind-numbing software, that makes it similar to the classiest technology on the planet? In his book *Trapped in the Net,* the Berkeley physicist turned sociologist, Gene Rochlin, has this to say about computer technology:

> Only in a few specialized markets are new developments in hardware and software responsive primarily to user demand based on mastery and the full use of available technical capacity and capability. In most markets, the rate of change of both hardware and software is dynamically uncoupled from either human or organizational learning logistics and processes, to the point where users not only fail to master their most recent new capabilities, but are likely to not even bother to try, knowing that by the time they are through the steep part of their learning curve, most of what they have learned will be obsolete.

To give a homey example, I spent the last quarter hour fiddling with the margins on the draft copy of this article. Microsoft Word has all manner of arcane symbols—Exacto knives, magnifying glasses, thumbtacks, globes—plus an annoying little paper clip homunculus that pops up, seemingly at random, to offer help that I always decline. I don't know what any of this stuff does. Since one of the best-selling commercial introductions to the Microsoft Office suite now runs to nearly a thousand pages, roughly the size of Shakespeare's collected works, I won't find out either. To the untrained eye, that is to say, to mine, the bulk of what constitutes Microsoft Word appears to be useful primarily to brochure designers and graphic artists. This unused cornucopia is not peculiar to Microsoft, nor even to microcomputer software. Programmers were cranking out obscure and poorly documented features long before computers became a consumer product.

Though the medium on which it is stored might decay, the software itself, because it exists in the same ethereal way as a novel, scored music, or a mathematical theorem, lasts as long as the ability to decode it.

But why? Remember the nature of software, how it does not wear out. Adding features to a new release is similar, but not identical, to changes in fashion or automobile styling. In those industries, a change in look gives natural, and planned, obsolescence a nudge. Even the best-built car or the sturdiest pair of jeans will eventually show signs of wear. Changes in fashion just speed this process along. Not so with software. Though the medium on which it is stored might decay, the software itself, because it exists in the same ethereal way as a novel, scored music, or a mathematical theorem, lasts as long as the ability to decode it. That is why Microsoft Word and the operating systems that support it, such as Microsoft Windows, get more complex with each new release.

But this is only part of the story. While software engineers at Oracle or Microsoft are staying up late concocting features that no one will ever use, hardware engineers at Intel are inventing ever faster, ever cheaper processors to run them. If Microsoft did not take advantage of this additional capacity, someone else would. Hardware and software are locked in an intricate and pathological dance. Hardware takes a step. Software follows. Hardware takes another step, and so on. The result is the Vegematic Promise. Do you want to write a letter to your bank? Microsoft Word will work fine. Do you need to save your work in any one of fifteen different digital formats? Microsoft Word will do the job. Do you want to design a Web page, lay out a brochure, import clip art, or include the digitally rendered picture of your dog? The designers at Microsoft have anticipated your needs. They were able to do this because the designers at Intel anticipated theirs. What no one anticipated was the unmanageable complexity of the final product from the user's perspective and the stunning, internal complexity of the product that Microsoft brings to market. In another time, this kind of complexity would have been reserved for enterprises of true consequence, say the Manhattan Project or the *Apollo* missions. Now the complexity that launched a thousand ships, placed men on the moon, controlled nuclear fission and fusion, the complexity that demanded of its designers years of training and still failed routinely, sits on my desk. Only this time, developers with minimal, often informal, training, using tools that change before they master them, labor for my daughter, who uses the fruits of their genius to chat with her friends about hair, makeup, and boys.

As I say, accelerating complexity is not just a software feature. Gordon Moore, one of Intel's founders, famously observed, in 1965, that the number of transistors etched on an integrated circuit board doubled every year or so. In the hyperbolic world of computing, this observation, altered slightly for the age of microprocessors, has come to be called Moore's Law: the computing power of microprocessors tends to double every couple of years. Though engineers expect to reach physical limits sometime in the first quarter of this century, Moore has been on target for the past couple dozen years. As a related, if less glamorous example, consider the remote control that accompanies electronic gadgetry these days. To be at the helm of your VCR, TV, DVD player, stereo (never mind lights, fans, air-conditioning, and fireplace), is to be a kind of Captain Kirk of home and hearth. The tendency, the Vegematic Promise, is to integrate separate remote controls into a single device. A living room equipped with one of these marvels is part domicile, part mission control. I recently read about one fellow who, dazzled by the complexity of integrated remotes, fastened his many devices to a chunk of four-by-four with black electrical tape. I have ceded operation of my relatively low-tech equipment to my teenage daughter, the only person in my house with the time or inclination to decipher its runic symbols.

But software is different in one significant way. Hardware, by and large, works. When hardware fails, as early versions of the Pentium chip did, it is national news. It took a computer scientist in Georgia doing some fairly obscure mathematical calculations to uncover the flaw. If only software errors were so well hidden. Engineers, even electrical engineers, use well-understood, often off-the-shelf, materials with well-defined limits. To offer a simple example, a few years ago I taught a course in digital logic. This course, standard fare for all computer science and computer engineering majors, teaches students how to solve logic problems with chips. A common lab problem is to build a seven-segment display, a digital display of numbers, like what you might find on an alarm clock. Students construct it using a circuit board and chips that we order by the hundreds. These chips are described in a catalogue that lists the number and type of logical operations encoded, along with the corresponding pins for each. If you teach software design, as I do, this trespass into the world of the engineer is instructive. Software almost always gets built from scratch. Though basic sorting and string manipulation routines exist, these must be woven together in novel ways to produce new software. Each programmer becomes a craftsman with a store of tricks up his sleeve. The more experienced the programmer, the more tricks.

To be fair, large software-development operations maintain libraries of standard routines that developers may dip into when the need arises. And for the past ten years or so, new object-oriented design and development techniques have conceived of ways to modularize and standardize components. Unfortunately, companies have not figured out how to make money by selling components, probably for the same reason that the music industry is under siege from Napster's descendants. If your product is only a digital encoding, it can be copied endlessly at almost no cost. Worse, the object-oriented programming paradigm seems often to be more complex than a conventional approach. Though boosters claim that programmers using object-oriented techniques are more productive and that their products are easier to maintain, this has yet to be demonstrated.

Software is peculiar in another way. Though hardware can be complex in the extreme, software obeys no physical limits. It can be as feature-rich as its designers wish. If the computer's memory is too small, relatively obscure features can be stored on disk and called into action only when needed. If the computer's processor is too slow, just wait a couple of years. Designers want your software to be very feature-rich indeed, because they want to sell the next release, because the limits of what can be done with a computer are not yet known, and, most of all, because those who design computer systems, like the rich in the world of F. Scott Fitzgerald, are different from you and me. Designers love the machine with a passion not constrained by normal market mechanisms or even, in some instances, by managerial control.

On the demand side, most purchases are made by institutions, businesses, universities, and the government, where there is an obsessive fear of being left behind, while the benefits, just as in the military, are difficult to measure. The claims and their outcomes are too fuzzy to be reconciled. Since individual managers are rarely held accountable for decisions to buy yet more computing equipment, it should not surprise you that wildly complex technology is being underused. Thus: computer labs that no one knows what to do with, so-called smart classrooms that are obsolete before anyone figures out how to use them, and offices with equipment so complicated that every secretary doubles as a systems administrator. Even if schools and businesses buy first and ask questions later, *you* don't have to put up with this. You could tell Microsoft to keep its next Windows upgrade, your machine is working very nicely right now, thank you. But your impertinence will cost you. Before long, your computer will be an island where the natives speak a language cut off from the great linguistic communities. In a word, you will be isolated. You won't be able to buy new software, edit a report you wrote at work on your home computer, or send pictures of the kids to Grandma over the Internet. Further, a decision to upgrade later will be harder, perhaps impossible, without losing everything your trusted but obsolete computer has stored. This is what Rochlin means when he writes that hardware and software are "dynamically uncoupled from either human or organizational learning." To which I would add "human organizational need."

What if the massively complex new software were as reliable as hardware usually is? We still wouldn't know how to use it, but at least our screens wouldn't lock up and our projects wouldn't be canceled midstream. This reliability isn't going to happen, though, for at least three reasons. First, programmers love complexity, love handcrafted systems, with an ardor that most of us save for our spouses. You have heard about the heroic hours worked by employees of the remaining Internet startups. This is true, but true only partly so that young men can be millionaires by thirty. There is something utterly beguiling about programming a computer. You lose track of time, of space even. You begin eating pizzas and forgetting to bathe. A phone call is an unwelcome intrusion. Second, nobody can really oversee a programmer's work, short of reading code line by line. It is simply too complex for anyone but its creator to understand, and even for him it will be lost in the mist after a couple of weeks. The 90 percent syndrome is a natural consequence. Programmers, a plucky lot, always think that they are further along than they are.

It is difficult to foresee an obstacle on a road you have never traveled. Despite all efforts to the contrary, code is handcrafted. Third—and this gets to the heart of the matter—system specifications have the half-life of an adolescent friendship. Someone—the project manager, the team leader, a programmer, or, if the system is built on contract, the client—always has a new idea. It is as if a third of the way through building a bridge, the highway department decided it should have an additional traffic lane and be moved a half mile downstream.

Notice that not one of the reasons I have mentioned for failed software projects is technical. Researchers trying to develop a discipline of software engineering are fond of saying that there is no silver bullet: no single technical fix, no single software-development tool, no single, yet-to-be-imagined programming technique that will result in error-free, maintainable software. The reason for this is really quite simple. The problem with software is not technical. Remember my project. It fell into chaos because of foolish business decisions. Had Mark resisted the temptation to use the latest software-development products, a temptation he succumbed to not because they would produce a better system, but because they would seem flashier to prospective clients, we might have gone to market with only the usual array of problems.

Interestingly, the geek's geek, Bruce Schneier, in his recent book, *Secrets and Lies*, has come to similar conclusions about computer security: the largest problems are not technical. A computer security expert, Schneier has recanted his faith in the impermeability of cryptographic algorithms. Sophisticated cryptography is as resistant as ever to massive frontal attacks. The problem is that these algorithms are embedded in computer systems that are administered by real human beings with all their charms and foibles. People use dictionary entries or a child's name as passwords. They attach modems to their office computers, giving hackers easy access to a system that might otherwise be more thoroughly protected. They run versions of Linux with all network routines enabled, or they surreptitiously set up Web servers in their dormitory rooms. Cryptographic algorithms are no more secure than their contexts.

Until computing is organized like engineering, law, and medicine through a combination of self-regulating professional bodies, government-imposed standards, and the threat of litigation, inviting a computer into your house or office is to invite complexity masquerading as convenience.

Though the long march is far from over, we know a lot more about managing the complexity of software systems than we did twenty years ago. We have better programming languages and techniques, better design principles, clever software to keep track of changes, richly endowed procedures for moving from conception to system design to coding to testing to release. But systems still fail and projects are still canceled with the same regularity as in the bad old days before object-oriented techniques, before software engineering becomes an academic discipline. These techniques are administered by the same humans who undermine computer security. They include marketing staff who decree systems into existence; companies that stuff yet more features into already overstuffed software; designers and clients who change specifications as systems are being built; programmers who are more artist than engineer; and, of course, software itself that can be neither seen, nor touched, nor measured in any significant way.

There is no silver bullet. But just as the *Challenger* disaster might have been prevented with minimal common sense, so also with software failure. Keep it simple. Avoid exotic and new programming techniques. Know that an army of workers is no substitute for clear design and ample time. Don't let the fox, now disguised as a young man with a head full of acronyms, guard the chicken coop. Make only modest promises. Good advice, certainly, but no one is likely to listen anytime soon. Until computing is organized like engineering, law, and medicine through a combination of self-regulating professional bodies, government-imposed standards, and, yes, the threat of litigation, inviting a computer into your house or office is to invite complexity masquerading as convenience. Given the nature of computing, even these remedies may fall short of the mark.

But don't despair. If software engineering practice is out of reach, you still have options. For starters, you could just say no. You could decide that the ease of buying plane tickets online is not worth the hours you while away trying to get your printer to print or your modem to dial. Understand that saying no requires an ascetic nature: abstinence is not terribly attractive to most of us. On the other hand, you could sign up for broadband with the full knowledge that your computer, a jealous lover, will demand many, many Saturday afternoons. Most people are shocked when they learn that their computer requires more care than, say, their refrigerator. Yet I can tell you that its charms are immeasurably richer. First among them is the dream state. It's almost irresistible.

Paul De Palma is associate professor of mathematics and computer science at Gonzaga University. His essay "http://www.when_is_enough_enough?.com" appeared in the *Winter 1999* issue.