Chapter 1

Issues—The Software Crisis

1. Introduction to Chapter

The term “software crisis” has been used since the late 1960s to describe those recurring system development problems in which software development problems cause the entire system to be late, over budget, not responsive to the user and/or customer requirements, and difficult to use, maintain, and enhance. The late Dr. Winston Royce, in his paper Current Problems [1], emphasized this situation when he said in 1991:

The construction of new software that is both pleasing to the user/buyer and without latent errors is an unexpectedly hard problem. It is perhaps the most difficult problem in engineering today, and has been recognized as such for more than 15 years. It is often referred to as the “software crisis”. It has become the longest continuing “crisis” in the engineering world, and it continues unabated.

This chapter describes some of the current issues and problems in system development that are caused by software—software that is late, is over budget, and/or does not meet the customers’ requirements or needs.

Software is the set of instructions that govern the actions of a programmable machine. Software includes application programs, system software, utility software, and firmware. Software does not include data, procedures, people, and documentation. In this tutorial, “software” is synonymous with “computer programs.”

Because software is invisible, it is difficult to be certain of development progress or of product completeness and quality. Software is not governed by the physical laws of nature: there is no equivalent of Ohm’s Law, which governs the flow of electricity in a circuit; the laws of aerodynamics, which act to keep an aircraft flying stably in the air; or Maxwell’s Equations, which describe the radiation of energy from an antenna.
In addition, software is not manufactured like hardware; it does not have a production phase nor manufactured spare parts like hardware; it is typically custom-built, not assembled from existing components like hardware. Even in today’s society, software is viewed with suspicion by many individuals, such as senior managers and customers, as somewhat akin to “black magic.”

The result is that software is one of the most difficult artifacts of the modern world to develop and build.

2. Introduction to Papers

The opening paper fortuitously appeared in a recent issue of *Scientific American* as the editors were casting about for a way to incorporate a recent rash of high-publicity software problems into the motivation for this tutorial. The paper defines and presents essentially all the major issues currently plaguing software development and maintenance. The article is “popular” rather than technical in the sense that it is journalistic in style and focuses on popular perceptions of software as “black magic,” but it raises many issues that software professionals need to be familiar with. It is also worth noting that many of the problems described are partly or largely due to non-software issues such as politics, funding, and external constraints, but again the software professional needs to know that problems unrelated to software engineering must overcome if software projects are to be successful.

The term “software crisis” not unexpectedly originated with the military, for that is where large, complex “real-time” software was first developed. More recently, as civilian and commercial software systems have approached and exceeded military systems in size, complexity, and performance requirements, the “software crisis” has occurred in these environments as well. It is noteworthy that the *Scientific American* article mentions military systems only peripherally.

The article begins with a discussion of the highly-publicized and software-related failure of the baggage system at the new Denver International Airport. As of the date of the article, opening of the airport had been delayed four times, for almost a year, at a cost to the airport authority of over $1 million a day.

Almost as visible in recent months, and also mentioned in the article, are failures of software development for the Department of Motor Vehicles (DMV) of the State of California, and for the advanced air traffic control system of the US Federal Aviation Administration (FAA). The DMV project involved attempts to merge existing, separately developed systems that managed driver’s licenses and vehicle registrations. As has been pointed out in the press [2], the State of California has had problems with computer projects of over $1 billion in value, and the problems resulted from the acquisition policies of the State of California (how contractors and consultants are selected and managed by the State), and from hardware-software integration difficulties, as well as from causes strictly related to software development.

The article identifies the first use of the term “software engineering” in a 1968 conference of the NATO Science Committee in Garmisch, Germany. (See also the Bauer article in this Tutorial.) Many approaches that have been proposed to improve software development are discussed; the author feels that most of these ideas have not lived up to the expectations of their originators. Also discussed is the idea that there are no “silver bullets.” (See the article by Brooks in this chapter.)

The *Scientific American* article looks favorably on the use of formal specification methods to solve the problem of software quality, and on “software reuse” (the ability to use a software product developed for one application again later for another application) to solve the productivity or cost problem.

The Software Engineering Institute’s Capability Maturity Model was also favorably mentioned (see the article by Paulk, Curtis, Chrissis, and Weber in this Tutorial) as a motivation to software developers to improve their practices. The paper reports an SEI finding that approximately 75 percent of all software developers do not have any formal process or any productivity or quality metrics.

Because software development depends on an educated workforce and good communications rather than on a fixed plant of any kind, software is inherently a suitable export product for developing countries. Although the US is still strong in software design and project management, the article notes that third world countries—notably India and Far Eastern countries—are capable of producing many more “lines of code” per dollar.

A sidebar by Dr. Mary Shaw provides a view of software engineering’s history, and of how that history may serve as a roadmap for software engineering’s future. Finally, the paper urges education of computer science students in software engineering as an essential step toward resolving the software crisis.

The second and last article in this chapter, “No Silver Bullets: Essence and Accidents of Software Engineering,” is by Fred Brooks, one of the legendary figures in software engineering. He has been called the father of software engineering project management in the United States. He worked at IBM in the 1960s and was the software project manager for the OS/360 operating system.
This paper, which he wrote in 1987, states that “no single technique exists to solve the software crisis, that there is no silver bullet.” The easy problems (“accidents”) have been solved and the remaining difficulties are “essential.” He views the solution to the software crisis as a collection of many software engineering tools and techniques that, used in combination, will reduce or eliminate software problems. Although Brooks sees no single solution to the software crisis, no single technology or management technique, he does see encouragement for the future through disciplined, consistent efforts to develop, propagate, and exploit many of the software tools and techniques that are being developed today. (In a report, also written in 1987 [3], Brooks states his belief that most software development problems of the US Department of Defense are managerial rather than technical.)

Brooks believes the hard part of building software is the specification and design of a system, not the coding and testing of the final product. As a result, he believes that building software will always be hard. There is no apparent simple solution. Brooks describes the three major advances in software development as:

- The use of high level languages
- The implementation of time-sharing to improve the productivity of programmers and the quality of their products
- Unified programming environment

Brooks also cites the Ada language, object-oriented programming, artificial intelligence, expert systems, and “automatic” programming (automated generation of code from system specification and design) as technologies with the potential for improving software. From the perspective of another eight years, the AI-related technologies for the most part have yet to fulfill the potential that Brooks saw for them in 1987.

Denver's new international airport was to be the pride of the Rockies, a wonder of modern engineering. Twice the size of Manhattan, 10 times the breadth of Heathrow, the airport is big enough to land three jets simultaneously—in bad weather. Even more impressive than its girth is the airport's subterranean baggage-handling system. Tearing like intelligent coal-mine cars along 21 miles of steel track, 4,000 independent "telecars" route and deliver luggage between the counters, gates and claim areas of 20 different airlines. A central nervous system of some 100 computers networked to one another and to 5,000 electric eyes, 400 radio receivers and 56 bar-code scanners orchestrates the safe and timely arrival of every valise and ski bag.

At least that is the plan. For nine months, this Gulliver has been held captive by Lilliputians—errors in the software that controls its automated baggage system. Scheduled for takeoff by last Halloween, the airport's grand opening was postponed until December to allow BAE Automated Systems time to flush the gremlins out of its $193-million system. December yielded to March. March slipped to May. In June the airport's planners, their bond rating demoted to junk and their budget hemorrhaging red ink at the rate of $1.1 million a day in interest and operating costs, conceded that they could not predict when the baggage system would stabilize enough for the airport to open.

To veteran software developers, the Denver debacle is notable only for its visibility. Studies have shown that for every six new large-scale software systems that are put into operation, two others are canceled. The average software development project overshoots its schedule by half; larger projects generally do worse. And some three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all.

The art of programming has taken 50 years of continual refinement to reach this stage. By the time it reached 25, the difficulties of building big software loomed so large that in the autumn of 1968 the NATO Science Committee convened some 50 top programmers, computer scientists and captains of industry to plot a course out of what had come to be known as the software crisis. Although the experts could not contrive a road map to guide the industry toward firmer ground, they did coin a name for that distant goal: software engineering, now defined formally as "the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software."

A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently. "It's like musket making was before Eli Whitney," says Brad J. Cox, a professor at George Mason University. "Before the industrial revolution, there was a nonspecialized approach to manufacturing goods that involved very little interchangeability and a maximum of craftsmanship. If we are ever going to lick this software crisis, we're going to have to stop this hand-to-mouth, every-programmer-builds-everything-from-the-ground-up, preindustrial approach."

The picture is not entirely bleak. Intuition is slowly yielding to analysis as programmers begin using quantitative measurements of the quality of the software they produce to improve...
the way they produce it. The mathematical foundations of programming are solidifying as researchers work on ways of expressing program designs in algebraic forms that make it easier to avoid serious mistakes. Academic computer scientists are starting to address their failure to produce a solid corps of software professionals. Perhaps most important, many in the industry are turning their attention toward inventing the technology and market structures needed to support interchangeable, reusable software parts. "Un fortunately, the industry does not uniformly apply that which is well-known best practice," laments Larry E. Druffel, director of Carnegie Mellon University's Software Engineering Institute. In fact, a research innovation typically requires 18 years to wend its way into the repertoire of standard programming techniques. By combining their efforts, academia, industry and government may be able to hoist software development to the level of an industrial-age engineering discipline within the decade. If they come up short, society's headlong rush into the information age will be halting and unpredictable at best.

**Shifting Sands**

"We will see massive changes [in computer use] over the next few years, causing the initial personal computer revolution to pale into comparative insignificance," concluded 22 leaders in software development from academia, industry and research laboratories this past April. The experts gathered at Hed sor Park, a corporate retreat near London, to commemorate the NATO conference and to analyze the future directions of software. "In 1968 we knew what we wanted to build but couldn't," reflected Cliff Jones, a professor at the University of Manchester. "Today we are standing on shifting sands."

The foundations of traditional programming practices are eroding swiftly, as hardware engineers churn out ever faster, cheaper and smaller machines. Many fundamental assumptions that programmers make—for instance, their acceptance that everything they produce will have defects—must change in response. "When computers are em bedded in light switches, you've got to get the software right the first time because you're not going to have a chance to update it," says Mary M. Shaw, a professor at Carnegie Mellon.

"The amount of code in most consumer products is doubling every two years," notes Remi H. Bourgonjon, director of software technology at Philips Research Laboratory in Eindhoven. Already, he reports, televisions may contain up to 500 kilobytes of software; an electric shaver, two kilobytes. The power trains in new General Motors cars run 30,000 lines of computer code.

Getting software right the first time is hard even for those who care to try. The Department of Defense applies rigorous—and expensive—testing standards to ensure that software on which a mission depends is reliable. Those standards were used to certify Clementine, a satellite that the DOD and the National Aeronautics and Space Administration directed into lunar orbit this past spring. A major part of the Clementine mission was to test targeting software that could one day be used in a space-based missile defense system. But when the satellite was spun around and instructed to fix the moon in its sights, a bug in its program caused the spacecraft instead to fire its maneuvering thrusters continuously for 11 minutes. Out of fuel and spinning wildly, the satellite could not make its rendezvous with the asteroid Geographos.

Errors in real-time systems such as Clementine are devilishly difficult to spot because, like that suspicious sound in your car engine, they often occur only when conditions are just so [see "The Risks of Software," by Bev Littlewood and Lorenzo Strigini; SCIENTIFIC AMERICAN, November 1992]. "It is not clear that the methods that are currently used for producing safety-critical software, such as that in nuclear reactors or in cars, will evolve and scale up adequately to match our future expectations," warned Gilles Kahn, the scientific director of France's INRIA research laboratory, at the Hedsor Park meeting. "On the contrary, for real-time systems I think we are at a fracture point."

Software is buckling as well under tectonic stresses imposed by the inexorably growing demand for "distributed systems": programs that run cooperatively on many networked computers. Businesses are pouring capital into distributed information systems that they hope to wield as strategic weapons. The inconstancy of software development can turn such projects into Russian roulette.

Many companies are lured by goals that seem simple enough. Some try to reincarnate obsolete mainframe-based software in distributed form. Others want to plug their existing systems into one another or into new systems with which they can share data and a friendlier user interface. In the technical lingo, connecting programs in this way is often called systems integration. But Brian Randell, a computer scientist at the University of Newcastle upon Tyne, suggests that "there is a better word than integration, from old R. F. Scott: namely, 'to graunch,' which means 'to make to fit by the use of excessive force.'"

It is a risky business, for although...
software seems like malleable stuff, most programs are actually intricate plexuses of brittle logic through which data of only the right kind may pass. Like handmade muskets, several programs may perform similar functions and yet still be unique in design. That makes software difficult to modify and repair. It also means that attempts to grapple systems together often end badly.

In 1987, for example, California's Department of Motor Vehicles decided to make its customers' lives easier by merging the state's driver and vehicle registration systems—a seemingly straightforward task. It had hoped to unveil convenient one-stop renewal kiosks last year. Instead the DMV saw the projected cost explode to 6.5 times the expected price and the delivery date recede to 1998. In December the agency pulled the plug and walked away from the seven-year, $44.3-million investment.

Sometimes nothing fails like success. In the 1970s American Airlines constructed SABRE, a virtuosoic, $2-billion flight reservation system that became part of the travel industry's infrastructure. "SABRE was the shining example of a strategic information system because it drove American to being the world's largest airline," recalls Bill Curtis, a consultant to the Software Engineering Institute.

Intent on brandishing software as effectively in this decade, American tried to grapple its flight-booking technology with the hotel and car reservation systems of Marriott, Hilton and Budget. In 1992 the project collapsed into a heap of litigation. "It was a smashing failure," Curtis says. "American wrote off $165 million against that system."

The airline is hardly suffering alone. In June IBM's Consulting Group released the results of a survey of 24 leading companies that had developed large distributed systems. The numbers were unsettling: 55 percent of the projects cost more than expected, 68 percent overran their schedules and 88 percent had to be substantially redesigned.

The survey did not report one critical statistic: how reliably the completed programs ran. Often systems crash because they fail to expect the unexpected. Networks amplify this problem. "Distributed systems can consist of a great set of interconnected single points of failure, many of which you have not identified beforehand," Randall explains. "The complexity and fragility of these systems pose a major challenge."

The challenge of complexity is not only large but also growing. The bang that computers deliver per buck is doubling every 18 months or so. One result is "an order of magnitude growth in system size every decade—for some industries, every half decade," Curtis says. To keep up with such demand, programmers will have to change the way that they work. "You can't build skyscrapers using carpenters," Curtis quips.

Mayday, Mayday

When a system becomes so complex that no one manager can comprehend the entirety, traditional development processes break down. The Federal Aviation Administration (FAA) has faced this problem throughout its decade-old attempt to replace the nation's increasingly obsolete air-traffic control system [see "Aging Airways," by Gary Six; Scientific American, May].

The replacement, called the Advanced Automation System (AAS), combines all the challenges of computing in the 1990s. A program that is more than a million lines in size is distributed across hundreds of computers and embedded into new and sophisticated hardware, all of which must respond around the clock to unpredictable real-time events. Even a small glitch potentially threatens public safety.

To realize its technological dream, the FAA chose IBM's Federal Systems Company, a well-respected leader in software development that has since been purchased by Loral. FAA managers expected (but did not demand) that IBM would use state-of-the-art techniques to estimate the cost and length of the project. They assumed that IBM would screen the requirements and design drawn up for the system in order to catch mistakes early, when they can be fixed in hours rather than days. And the FAA conservatively expected to pay about $500 per line of computer code, five times the industry average for well-managed development processes.

According to a report on the AAS project released in May by the Center for Naval Analysis, IBM's "cost estimation and development process tracking used inappropriate data, were performed inconsistently and were routinely ignored" by project managers. As a result, the FAA has been paying $700 to $900 per line for its bold AAS software. One reason for the exorbitant price is that "on average every line of code developed needs to be rewritten once," be-
no way of knowing when they are on the wrong track or off the track altogether."

(The Center for Naval Analysis concluded that the AAS project at IBM Federal Systems "appears to be at a low 1 rating.") The remaining 24 percent of projects are at levels 2 or 3.

Only two elite groups have earned the highest CMM rating, a level 5. Motorola's Indian programming team in Bangalore holds one title. Loral's (formerly IBM's) on-board space shuttle software project claims the other. The Loral team has learned to control bugs so well that it can reliably predict how many will be found in each new version of the software. That is a remarkable feat, considering that 90 percent of American programmers do not even keep count of the mistakes they find, according to Capers Jones, chairman of Software Productivity Research. Of those who do, he says, few catch more than a third of the defects that are there.

Tom Peterson, head of Loral's shuttle software project, attributes its success to "a culture that tries to fix not just the bug but also the flaw in the testing process that allowed it to slip through." Yet some bugs inevitably escape detection. The first launch of the space shuttle in 1981 was aborted and delayed for two days because a glitch prevented the five on-board computers from synchronizing properly. Another flaw, this one in the shuttle's rendezvous program, jeopardized the Intelsat-6 satellite rescue mission in 1992.

Although the CMM is no panacea, its promotion by the Software Engineering Institute has persuaded a number of leading software companies that quantitative quality control can pay off in the long run. Raytheon's equipment division, for example, formed a "software engineering initiative" in 1988 after flunking the CMM test. The division began pouring $1 million per year into refining rigorous inspection and testing guidelines and training its 400 programmers to follow them.

Within three years the division had jumped two levels. By this past June, most projects—including complex radar and air-traffic control systems—were finishing ahead of schedule and under budget. Productivity has more than doubled. An analysis of avoided rework costs revealed a savings of $7.80 for every dollar invested in the initiative. Impressed by such successes, the U.S. Air Force has mandated that all its software developers must reach level 3 of the CMM by 1998. NASA is reportedly considering a similar policy.

**Mathematical Re-creations**

Even the best-laid designs can go awry, and errors will creep in so long as humans create programs. Bugs squashed early rarely threaten a project's deadline and budget, however. Devastating mistakes are nearly always those in the initial design that slip undetected into the final product.

Mass-market software producers, because they have no single customer to please, can take a belated and brute-force approach to bug removal: they release the faulty product as a "beta" version and let hordes of users dig up the glitches. According to Charles Simonyi, a chief architect at Microsoft, the new version of the Windows operating system will be beta-tested by 20,000 volunteers. That is remarkably effective, but also expensive, inefficient—and since mass-produced PC products make up less than 10 percent of the $92.8-billion software market in the U.S.—usually impractical.

Researchers are thus formulating several strategies to attack bugs early or to avoid introducing them at all. One idea is to recognize that the problem a system is supposed to solve always changes as the system is being built. Denver's airport planners saddled BAe with $20 million worth of changes to the design of its baggage system long after construction had begun. IBM has been similarly bedeviled by the indecision of FAA managers. Both companies naively assumed that once their design was approved, they would be left in peace to build it.

Some developers are at last shedding that illusion and rethinking software as something to be grown rather than built. As a first step, programmers are increasingly stitching together quick prototypes out of standard graphic interface components. Like an architect's scale model, a system prototype can help clear up misunderstandings between customer and developer before a logical foundation is poured.

Because they mimic only the outward behavior of systems, prototypes are of little help in spotting logical inconsistencies in a system's design. "The vast majority of errors in large-scale software are errors of omission," notes Laszlo A. Belady, director of Mitsubishi Electric Research Laboratory. And models do not make it any easier to detect bugs once a design is committed to code.

When it absolutely, positively has to be right, says Martyn Thomas, chairman of Praxis, a British software company, engineers rely on mathematical analysis to predict how their designs will behave in the real world. Unfortunately, the mathematics that describes physical systems does not apply within the synthetic binary universe of a computer program; discrete mathematics, a far less mature field, governs here. But using the still limited tools of set theory and predicate calculus, computer scientists have contrived ways to translate specifications and programs into the language of mathematics, where they can be analyzed with theoretical tools called formal methods.

**RAYTHEON HAS SAVED $17.2 million in software costs since 1988, when its equipment division began using rigorous development processes that doubled its programmers' productivity and helped them to avoid making expensive mistakes.**
Progress toward Professionalism

ENGINEERING EVOLUTION
PARADIGM

Skilled craftsmen
Established procedure
Pragmatic refinement
Training in mechanics
Economic concern for cost and supply of materials
Manufacture for sale

CHEMICAL ENGINEERING

1775: French Academy offers reward for method to convert brine (salt) to soda ash (alkali)
1808: John Dalton publishes his atomic theory
1823: Nicolas Leblanc's industrial alkali process first put into operation
1850s: Pollution of British Midlands by alkali plants
1857: William Henry Perkin founds synthetic dye industry

SOFTWARE ENGINEERING

1970s: Structured programming methods gain favor
1980s: Fourth-generation languages released
1990s: Reuse repositories founded

SCIENCE

1774: Joseph Priestley isolates oxygen
1808: John Dalton publishes his atomic theory
1887: George E. Davis identifies functional operations
1922: Hermann Staudinger explains polymerization

1956: IBM invents FORTRAN
1966: Donald E. Knuth publishes his theory of algorithms and data structures
1972: Smalltalk object-oriented language released
1980s: Formal methods and notations refined

PROFESSIONAL ENGINEERING

Educated professionals
Analysis and theory
Progress relies on science
Analysis enables new applications
Market segmentation by product variety

1915: Arthur D. Little refines and demonstrates unit operations
1994: Du Pont operates chemical megaplants

1994: Isolated examples only of algorithms, data structures, compiler construction

1930s: Programs are small and intuitive
1950s: SABRE airline reservation system is rare success
1990s: Most personal computer software is still handcrafted

1700s: Lye boiled to make soap
Most dyes made from vegetables

Like software developers, chemical engineers try to design processes to create safe, pure products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modeling, proven design solutions and rigorous quality-control methods—and their efforts usually succeed. Software, Shaw points out, is somewhat less mature, more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale programming to the commercial stage, computer science (which is younger than many of its researchers) has yet to build the experimental foundation on which software engineering must rest.

Engineers disciplines share common stages in their evolution, observes Mary M. Shaw of Carnegie Mellon University. She spies interesting parallels between software engineering and chemical engineering, two fields that aspire to exploit on an industrial scale the processes that are discovered by small-scale research.

Like software developers, chemical engineers try to design processes to create safe, pure products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modeling, proven design solutions and rigorous quality-control methods—and their efforts usually succeed. Software, Shaw points out, is somewhat less mature, more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale programming to the commercial stage, computer science (which is younger than many of its researchers) has yet to build the experimental foundation on which software engineering must rest.

Like software developers, chemical engineers try to design processes to create safe, pure products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modeling, proven design solutions and rigorous quality-control methods—and their efforts usually succeed. Software, Shaw points out, is somewhat less mature, more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale programming to the commercial stage, computer science (which is younger than many of its researchers) has yet to build the experimental foundation on which software engineering must rest.

Like software developers, chemical engineers try to design processes to create safe, pure products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modeling, proven design solutions and rigorous quality-control methods—and their efforts usually succeed. Software, Shaw points out, is somewhat less mature, more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale programming to the commercial stage, computer science (which is younger than many of its researchers) has yet to build the experimental foundation on which software engineering must rest.
Praxis recently used formal methods on an air-traffic control project for Britain's Civil Aviation Authority. Although Praxis's program was much smaller than the FAA's, the two shared a similar design problem: the need to keep redundant systems synchronized so that if one fails, another can instantly take over. "The difficult part was guaranteeing that messages are delivered in the proper order over twin networks," recalls Anthony Hall, a principal consultant to Praxis. "So here we tried to carry out proofs of our design, and they failed, because the design was wrong. The benefit of finding errors at that early stage is enormous," he adds. The system was finished on time and put into operation last October.

Praxis used formal notations on only the most critical parts of its software, but other software firms have employed mathematical rigor throughout the entire development of a system. GEC Alsthom in Paris is using a formal method called "B" as it spends $350 million to upgrade the switching- and speed-control software that guides the 6,000 electric trains in France's national railway system. By increasing the speed of the trains and reducing the distance between them, the system can save the railway company billions of dollars that might otherwise need to be spent on new lines.

Safety was an obvious concern. So GEC developers wrote the entire design and final program in formal notation and then used mathematics to prove them consistent. "Functional tests are still necessary, however, for two reasons," says Fernando Mejia, manager of the formal development section at GEC. First, programmers do occasion- also make mistakes in proofs. Secondly, formal methods can guarantee only that software meets its specification, not that it handle the surprises of real world.

Formal methods have other problems as well. Ted Ralston, director of strategy for Odyssey Research Associates in Ithaca, N.Y., points out that reading pages of algebraic formulas is even more stultifying than reviewing computer code. Odyssey is just one of several companies that are trying to automate formal methods to make them less onerous to programmers. GEC is collaborating with Digilog in France to commercialize programming tools for the B method. The beta version is being tested by seven companies and institutions, including Aerospatiale, as well as France's atomic energy authority and its defense department.

On the other side of the Atlantic, formal methods by themselves have yet to catch on. "I am skeptical that Americans are sufficiently disciplined to apply formal methods in any broad fashion," says David A. Fisher of the National Institute of Standards and Technology (NIST). There are exceptions, however, most notably among the growing circle of companies experimenting with the "clean-room" approach to programming.

The clean-room process attempts to meld formal notations, correctness proofs and statistical quality control with an evolutionary approach to software development. Like the microchip manufacturing technique from which it takes its name, clean-room development tries to use rigorous engineering techniques to consistently fabricate products that run perfectly the first time. Programmers grow systems one function at a time and certify the quality of each unit before integrating it into the architecture.

Growing software requires a whole new approach to testing. Traditionally, developers test a program by running it the way they intend it to be used, which often bears scant resemblance to real-world conditions. In a clean-room process, programmers try to assign a probability to every execution path—correct and incorrect—that users can take. They then derive test cases from those statistical data, so that the most common paths are tested more thoroughly. Next the program runs through each test case and times how long it takes to fail. Those times are then fed back, in true engineering fashion, to a model that calculates how reliable the program is.

Early adopters report encouraging results. Ericsson Telecom, the European telecommunications giant, used clean-room processes on a 70-programmer project to fabricate an operating system for its telephone-switching computers. Errors were reportedly reduced to just one per 1,000 lines of program code; the industry average is about 25 times higher. Perhaps more important, the company found that development productivity increased by 70 percent, and testing productivity doubled.

No Silver Bullet

Then again, the industry has heard tell many times before of "silver bullets" supposedly able to slay werewolf projects. Since the 1960s developers have peddled dozens of technological innova-
Third, Fisher says, "you can walk into a typical company and find two guys sharing an office, getting the same salary and having essentially the same credentials and yet find a factor of 100 difference in the number of instructions per day that they produce." Such enormous individual differences tend to swamp the much smaller effects of technology or process improvements.

After 25 years of disappointment with apparent innovations that turned out to be irreproducible or unscalable, many researchers concede that computer science needs an experimental branch to separate the general results from the accidental. "There has always been this assumption that if I give you a method, it is right just because I told you so," complains Victor R. Basili, a professor at the University of Maryland.

"People are developing all kinds of things, and it's really quite frightening how bad some of them are," he says.

Mary Shaw of Carnegie Mellon points out that mature engineering fields codify proved solutions in handbooks so that even novices can consistently handle routine designs, freeing more talented practitioners for advanced projects. No such handbook yet exists for software, so mistakes are repeated on project after project, year after year.

DeMillo suggests that the government should take a more active role. "The National Science Foundation should be interested in funding research aimed at verifying experimental results that have been claimed by other people," he says. "Currently, if it's not groundbreaking, first-time-ever-done research, program officers at the NSF tend to discount the work." DeMillo knows whereof he speaks. From 1989 to 1991 he directed the NSF's computer and computation research division.

Yet "if software engineering is to be an experimental science, that means it needs laboratory science. Where the heck are the laboratories?" Basili asks. "There is at the core of all commerce just does not work for things that can be copied in nanoseconds." When Cox tried selling the parts his programmers had created, he found that the price the market would bear was far too low for him to recover the costs of development.

Fisher favors the idea that components should be synthesized on the fly. Programmers would "basically capture how to do it rather than actually doing it," producing a recipe that any computer could understand. "Then when you want to assemble two components, you would take this recipe and derive compatible versions by adding additional elements to their interfaces. The whole thing would be automated," he explains.

Even with a $150-million incentive and market pressures forcing companies to find cheaper ways of producing software, an industrial revolution in software is not imminent. "We expect to see only isolated examples of these technologies in five to seven years—and we may not succeed technically either," Fisher hedges. Even when the technology is ready, components will find few takers unless they can be made cost-effective. And the cost of software parts will depend less on the technology involved than on the kind of market that arises to produce and consume them.

Brad Cox, like Fisher, once ran a software component company and found it hard going. He believes he has figured out the problem—and its solution. Cox's firm tried to sell low-level program parts analogous to computer chips. "What's different between software ICs [integrated circuits] and silicon ICs is that silicon ICs are made of atoms, so they abide by conservation of mass, and people therefore know how to buy and sell them robustly," he says. "But this interchange process that is at the core of all commerce just does not work for things that can be copied in nanoseconds." When Cox tried selling the parts his programmers had created, he found that the price the market would bear was far too low for him to recover the costs of development.

The reasons were twofold. First, recasting the component by hand for each customer was time-consuming; NIST hopes to clear this barrier with its Advanced Technology Program. The other factor was not so much technical as cultural: buyers want to pay for a component once and make copies for free.

"The music industry has had about a century of experience with this very problem," Cox observes. "They used to sell tangible goods like piano rolls and sheet music, and then radio and television came along and knocked all that into a cocked hat." Music companies adapted to broadcasting by setting up agencies to collect royalties every time a song is aired and to funnel the money back to the artists and producers.

Cox suggests similarly charging users each time they use a software compo-
A Developing World

Since the invention of computers, Americans have dominated the software market. Microsoft alone produces more computer code each year than do any of 100 nations, according to Capers Jones of Software Productivity Research in Burlington, Mass. U.S. suppliers hold about 70 percent of the worldwide software market.

But as international networks sprout and large corporations deflate, India, Hungary, Russia, the Philippines and other poorer nations are discovering in software a lucrative industry that requires the one resource in which they are rich: an underemployed, well-educated labor force. American and European giants are now competing with upstart Asian development companies for contracts, and in response many are forming subsidiaries overseas. Indeed, some managers in the trade predict that software development will gradually split between Western software engineers who design systems and Eastern programmers who build them.

"In fact, it is going on already," says Laszlo A. Bélady, director of Mitsubishi Electric Research Laboratory. AT&T, Hewlett-Packard, IBM, British Telecom and Texas Instruments have all set up programming teams in India. The Pact Group in Lyons, France, reportedly maintains a "software factory" in Manila. "Cadence, the U.S. supplier of VLSI design tools, has had its software development sited on the Pacific rim for several years," reports Martyn Thomas, chairman of Praxis. "ACT, a U.K.-based systems house, is using Russian programmers from the former Soviet space program," he adds.

So far India’s star has risen fastest. "Offshore development [work commissioned in India by foreign companies] has begun to take off in the past 18 to 24 months," says Rajendra S. Pawar, head of New Delhi-based NIIT, which has graduated 200,000 Indians from its programming courses. Indeed, India’s software exports have seen a compound annual growth of 38 percent over the past five years; last year they jumped 60 percent—four times the average growth rate worldwide.

About 58 percent of the $360-million worth of software that flowed out of India last year ended up in the U.S. That tiny drop hardly makes a splash in a $92.8-billion market. But several trends may propel exports beyond the $1-billion mark as early as 1997.

The single most important factor, Pawar asserts, is the support of the Indian government, which has eased tariffs and restrictions, subsidized numerous software technology parks and export zones, and doled out five-year tax exemptions to software exporters. "The opening of the Indian economy is acting as a very big catalyst," Pawar says.

It certainly seems to have attracted the attention of large multinational firms eager to reduce both the cost of the software they need and the amount they build in-house. The primary cost of software is labor. Indian programmers come so cheap—$125 per unit of software versus $925 for an American developer, according to Jones—that some companies fly an entire team to the U.S. to work on a project. More than half of India’s software exports come from such "body shopping," although tightened U.S. visa restrictions are stanching this flow.

Another factor, Pawar observes, is a growing trust in the quality of overseas project management. "In the past two years, American companies have become far more comfortable with the offshore concept," he says. This is a result in part of success stories from leaders like Citicorp, which develops banking systems in Bombay, and Motorola, which has a top-rated team of more than 150 programmers in Bangalore building software for its Iridium satellite network.

Offshore development certainly costs less than body shopping, and not merely because of saved airfare. "Thanks to the time differences between India and the U.S., Indian software developers can act the elves and the shoemaker," working overnight on changes requested by managers the previous day, notes Richard Heeks, who studies Asian computer industries at the University of Manchester in England.

Price is not everything. Most Eastern nations are still weak in design and management skills. "The U.S. still has the best system architects in the world," boasts Bill Curtis of the Software Engineering Institute. "At large systems, nobody touches us." But when it comes to just writing program code, the American hegemony may be drawing to a close.

<table>
<thead>
<tr>
<th>Year</th>
<th>India's Software Exports (Millions of U.S. Dollars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>16</td>
</tr>
<tr>
<td>1986</td>
<td>110</td>
</tr>
<tr>
<td>1987</td>
<td>39</td>
</tr>
<tr>
<td>1988</td>
<td>62</td>
</tr>
<tr>
<td>1989</td>
<td>67</td>
</tr>
<tr>
<td>1990</td>
<td>100</td>
</tr>
<tr>
<td>1991</td>
<td>1128</td>
</tr>
<tr>
<td>1992</td>
<td>164</td>
</tr>
<tr>
<td>1993</td>
<td>225</td>
</tr>
<tr>
<td>1994</td>
<td>360</td>
</tr>
<tr>
<td>1995</td>
<td>483</td>
</tr>
<tr>
<td>1996</td>
<td>NOT AVAILABLE</td>
</tr>
<tr>
<td>1997</td>
<td>1,000</td>
</tr>
</tbody>
</table>

Sources: NIIT, NASSCOM
"In fact," he says, "that model could work for software even more easily than for music, thanks to the infrastructure advantages that computers and communications give us. Record players don't have high-speed network links in them to report usage, but our computers do."

Or will, at least. Looking ahead to the time when nearly all computers are connected, Cox envisions distributing software of all kinds via networks that link component producers, end users and financial institutions. "It's analogous to a credit-card operation but with tentacles that reach into PCs," he says. Although that may sound ominous to some, Cox argues that "the Internet now is more like a garbage dump than a farmer's market. We need a national infrastructure that can support the distribution of everything from Grandma's cookie recipe to Apple's window managers to Addison-Wesley's electronic books." Recognizing the enormity of the cultural shift he is proposing, Cox expects to press his cause for years to come through the Coalition for Electronic Markets, of which he is president.

The combination of industrial process control, advanced technological tools and interchangeable parts promises to transform not only how programming is done but also who does it. Many of the experts who convened at Hedsor Park agreed with Belady that "in the future, professional people in most fields will use programming as a tool, but they won't call themselves programmers or think of themselves as spending their time programming. They will think they are doing architecture, or traffic planning or film making."

That possibility begs the question of who is qualified to build important systems. Today anyone can bill herself as a software engineer. "But when you have 100 million user-programmers, frequently they will be doing things that are life critical—building applications that fill prescriptions, for example," notes Barry W. Boehm, director of the Center for Software Engineering at the University of Southern California. Boehm is one of an increasing number who suggest certifying software engineers, as is done in other engineering fields.

Of course, certification helps only if programmers are properly trained to begin with. Currently only 28 universities offer graduate programs in software engineering; five years ago there were just 10. None offer undergraduate degrees. Even academics such as Shaw, DeMillo and Basili agree that computer science curricula generally provide poor preparation for industrial software development. "Basic things like designing code inspections, producing user documentation and maintaining aging software are not covered in academia," Capers Jones laments.

Engineers, the infantry of every industrial revolution, do not spontaneously generate. They are trained out of the bad habits developed by the craftsmen that preceded them. Until the lessons of computer science inculcate a desire not merely to build better things but also to build things better, the best we can expect is that software development will undergo a slow, and probably painful, industrial evolution.
No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed.

Skepticism is not pessimism, however. Although we see no startling break-throughs—and indeed, I believe such to be inconsistent with the nature of software—many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order-of-magnitude improvement. There is no royal road, but there is a road.

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

Does it have to be hard?—Essential difficulties

Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.
We cannot expect ever to see twofold gains every two years.

First, one must observe that the anomaly is not that software progress is so slow, but that computer hardware progress is so fast. No other technology since civilization began has seen six orders of magnitude in performance-price gain in 30 years. In no other technology can one choose to take the gain in either improved performance or in reduced costs. These gains flow from the transformation of computer manufacture from an assembly industry into a process industry.

Second, to see what rate of progress one can expect in software technology, let us examine the difficulties of that technology. Following Aristotle, I divide them into essence, the difficulties inherent in the nature of software, and accidents, those difficulties that today attend its production but are not inherent.

The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.

If this is true, building software will always be hard. There is inherently no silver bullet.

Let us consider the inherent properties of this irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility.

Complexity. Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into a subroutine—open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

Digital computers are themselves more complex than most things people build: They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do.

Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly.

The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence. For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.

Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From complexity of function comes the difficulty of invoking function, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors.

Not only technical problems, but management problems as well come from the complexity. It makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster.

Conformity. Software people are not alone in facing complexity. Physics deals
with terribly complex objects even at the "fundamental" particle level. The physicist labors on, however, in a firm faith that there are unifying principles to be found, whether in quarks or in unified-field theories. Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.

In many cases, the software must conform because it is the most recent arrival on the scene. In others, it must conform because it is perceived as the most conformable. But in all cases, much complexity comes from conformation to other interfaces; this complexity cannot be simplified out by any redesign of the software alone.

**Changeability.** The software entity is constantly subject to pressures for change. Of course, so are buildings, cars, computers. But manufactured things are infrequently changed after manufacture; they are superseded by later models, or essential changes are incorporated into later serial-number copies of the same basic design. Call-backs of automobiles are really quite infrequent; field changes of manufactured things are infrequent; field changes of computers have been really quite infrequent; field changes of computers somewhat less so. Both are much less frequent than modifications to fielded software.

In part, this is so because the software of a system embodies its function, and the function is the part that most feels the pressures of change. In part it is because software can be changed more easily—it is pure thought-stuff, infinitely malleable. Buildings do in fact get changed, but the high costs of change, understood by all, serve to dampen the whims of the changers.

All successful software gets changed. Two processes are at work. First, as a software product is found to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it.

Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity. In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.

**Invisibility.** Software is invisible and unvisualizable. Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious.

Despite progress in restricting and simplifying software structures, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools.

Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction. The reality of software is not inherently embedded in space. Hence, it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical. Indeed, one of the ways of establishing conceptual control over such structure is to enforce link cutting until one or more of the graphs becomes hierarchical.¹

In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds.

### Past breakthroughs solved accidental difficulties

If we examine the three steps in software-technology development that have been most fruitful in the past, we discover that each attacked a different major difficulty in building software, but that those difficulties have been accidental, not essential, difficulties. We can also see the natural limits to the extrapolation of each such attack.

**High-level languages.** Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility. What does a high-level language accomplish? It frees a program from much of its accidental complexity. An abstract program consists of conceptual constructs: operations, data types, sequences, and communication. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such. To the extent that the high-level language embodies the constructs one wants in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all.

The most a high-level language can do is to furnish all the constructs that the programmer imagines in the abstract program. To be sure, the level of our thinking about data structures, data types, and operations is steadily rising, but at an ever-decreasing rate. And language development approaches closer and closer to the sophistication of users.

Moreover, at some point the elaboration of a high-level language creates a tool-mastery burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs.

**Time-sharing.** Time-sharing brought a major improvement in the productivity of programmers and in the quality of their product, although not so large as that
brought by high-level languages.

Time-sharing attacks a quite different difficulty. Time-sharing preserves immediacy, and hence enables one to maintain an overview of complexity. The slow turnaround of batch programming means that one inevitably forgets the minutiae, if not the very thrust, of what one was thinking when he stopped programming and called for compilation and execution. This interruption is costly in time, for one must refresh one's memory. The most serious effect may well be the decay of the grasp of all that is going on in a complex system.

Slow turnaround, like machine-language complexities, is an accidental rather than an essential difficulty of the software process. The limits of the potential contribution of time-sharing derive directly from the development of whole toolbenches, programs that used the standard formats. As a result, conceptual structures that in principle could always call, feed, and use each other can indeed easily do so in practice.

Unified programming environments. Unix and Interlisp, the first integrated programming environments to come into widespread use, seem to have improved productivity by integral factors. Why?

They attack the accidental difficulties that result from using individual programs together, by providing integrated libraries, unified file formats, and pipes and filters. As a result, conceptual structures that in principle could always call, feed, and use one another can indeed easily do so in practice.

This breakthrough in turn stimulated the development of whole toolbenches, since each new tool could be applied to any programs that used the standard formats.

Because of these successes, environments are the subject of much of today's software-engineering research. We look at their promise and limitations in the next section.

Hopes for the silver

Now let us consider the technical developments that are most often advanced as potential silver bullets. What problems do they address—the problems of essence, or the remaining accidental difficulties? Do they offer revolutionary advances, or incremental ones?

Ada and other high-level language advances. One of the most touted recent de-

To slay the werewolf

Why a silver bullet? Magic, of course. Silver is identified with the moon and thus has magic properties. A silver bullet offers the fastest, most powerful, and safest way to slay the fast, powerful, and incredibly dangerous werewolf. And what could be more natural than using the moon-metal to destroy a creature transformed under the light of the full moon?

The legend of the werewolf is probably one of the oldest monster legends around. Herodotus in the fifth century BC gave us the first written report of werewolves when he mentioned a tribe north of the Black Sea, called the Neuri, who supposedly turned into wolves a few days each year. Herodotus wrote that he didn't believe it.

Sceptics aside, many people have believed in people turning into wolves or other animals. In medieval Europe, some people were killed because they were thought to be werewolves. In those times, it didn't take very little by a werewolf to become one. A bargain with the devil, using a special potion, wearing a special belt, or being cursed by a witch could all turn a person into a werewolf. However, medieval werewolves could be hurt and killed by normal weapons. The problem was to overcome their strength and cunning.

Enter the fictional, not legendary, werewolf. The first major werewolf movie, The Werewolf of London, in 1899 created the two-legged man-wolf who changed into a monster when the moon was full. He became a werewolf after being bitten by one, and could be killed only with a silver bullet. Sound familiar?

Actually, we owe many of today's ideas about werewolves to Lon Chaney Jr.'s unforgettable 1941 portrayal in The Wolf Man. Subsequent films seldom strayed far from the mythology of the werewolf shown in that movie. But that movie strayed far from the original mythology of the werewolf.

Would you believe that before fiction took over the legend, werewolves weren't troubled by silver bullets? Vampires were the ones who couldn't stand them. Of course, if you rely on the legends, your only salvation if unarmed and attacked by a werewolf is to climb an ash tree or run into a field of rye. Not so easy to find in an urban setting, and hardly recognizable to the average movie audience.

What should you watch out for? People whose eyebrows grow together, whose index finger is longer than the middle finger, and who have hair growing on their palms. Red or black teeth are a definite signal of possible trouble.

Take warning, though. The same symptoms may be found in people suffering from hypertrichosis (people born with hair covering their bodies) or porphyria. In porphyria, a person's body produces toxins called porphyrins. Consequently, light becomes painful, the skin grows hair, and the teeth may turn red. Worse for the victim's reputation, his or her increasingly bizarre behavior makes people even more suspicious of the other symptoms. It seems very likely that the sufferers of this disease unwittingly contributed to the current legend, although in earlier times they were evidently not accused of murder.

It is worth noting that the film tradition often makes the werewolf a rather sympathetic character, an innocent transformed against his (or rarely, her) will into a monster. As the gypsy said in The Wolf Man,

Even a man who is pure at heart,
And says his prayers at night,
Can become a wolf when the wolfbane blooms,
And the moon is full and bright.

—Nancy Hays
Assistant Editor

The Bettman Archive
velopments is Ada, a general-purpose high-level language of the 1980's. Ada not only reflects evolutionary improvements in language concepts, but indeed embodies features to encourage modern design and modularization. Perhaps the Ada philosophy is more of an advance than the Ada language, for it is the philosophy of modularization, of abstract data types, of hierarchical structuring. Ada is over-rich, a natural result of the process by which requirements were laid on its design. That is not fatal, for subsetted working vocabularies can solve the learning problem, and hardware advances will give us the cheap MIPS to pay for the compiling costs. Advancing the structuring of software systems is indeed a very good use for the increased MIPS our dollars will buy. Operating systems, loudly decried in the 1960's for their memory and cycle costs, have proved to be an excellent form in which to use some of the MIPS and cheap memory bytes of the past hardware surge.

Nevertheless, Ada will not prove to be the silver bullet that slays the software productivity monster. It is, after all, just another high-level language, and the biggest payoff from such languages came from the first transition—the transition up from the accidental complexities of the machine into the more abstract statement of step-by-step solutions. Once those accidents have been removed, the remaining ones will be smaller, and the payoff from their removal will surely be less.

I predict that a decade from now, when the effectiveness of Ada is assessed, it will be seen to have made a substantial difference, but not because of any particular language feature, nor indeed because of all of them combined. Neither will the new Ada environments prove to be the cause of the improvements. Ada's greatest contribution will be that switching to it occasioned training programmers in modern software-design techniques.

Object-oriented programming. Many students of the art hold out more hope for object-oriented programming than for other technical fads of the day.

Many students of the art hold out more hope for object-oriented programming than for other technical fads of the day.

Artificial intelligence. Many people expect advances in artificial intelligence to provide the revolutionary breakthrough that will give order-of-magnitude gains in software productivity and quality. I do not. To see why, we must dissect what is meant by "artificial intelligence."

D.L. Parnas has clarified the terminological chaos:

Two quite different definitions of AI are in common use today. AI-1: The use of computers to solve problems that previously could only be solved by applying human intelligence. AI-2: The use of a specific set of programming techniques known as heuristic or rule-based pro-

gramming. In this approach human experts are studied to determine what heuristics or rules of thumb they use in solving problems. The program is designed to solve a problem the way that humans seem to solve it.

The first definition has a sliding meaning. Something can fit the definition of AI-1 today but, once we see how the program works and understand the problem, we will not think of it as AI any more. Unfortunately I cannot identify a body of technology that is unique to this field. Most of the work is problem-specific, and some abstraction or creativity is required to see how to transfer it.

I agree completely with this critique. The techniques used for speech recognition seem to have little in common with those used for image recognition, and both are different from those used in expert systems. I have a hard time seeing how image recognition, for example, will make any appreciable difference in programming practice. The same problem is true of speech recognition. The hard thing about building software is deciding what one wants to say, not saying it. No facilitation of expression can give more than marginal gains.

Expert-systems technology, AI-2, deserves a section of its own.

Expert systems. The most advanced part of the artificial intelligence art, and the most widely applied, is the technology for building expert systems. Many software scientists are hard at work applying this technology to the software-building environment. Is what is the concept, and what are the prospects?

An expert system is a program that contains a generalized inference engine and a rule base, takes input data and assumptions, explores the inferences derivable from the rule base, yields conclusions and advice, and offers to explain its results by retracing its reasoning for the user. The inference engines typically can deal with fuzzy or probabilistic data and rules, in addition to purely deterministic logic.

Such systems offer some clear advantages over programmed algorithms designed for arriving at the same solutions to the same problems:

• Inference-engine technology is developed in an application-independent way, and then applied to many uses. One can justify much effort on the inference engines. Indeed, that technology is well advanced.

• The changeable parts of the application-peculiar materials are en-
coded in the rule base in a uniform fashion, and tools are provided for developing, changing, testing, and documenting the rule base. This regularizes much of the complexity of the application itself.

The power of such systems does not come from ever-fancier inference mechanisms, but rather from ever-richer knowledge bases that reflect the real world more accurately. I believe that the most important advance offered by the technology is the separation of the application complexity from the program itself.

How can this technology be applied to the software-engineering task? In many ways: Such systems can suggest interface rules, advise on testing strategies, remember bug-type frequencies, and offer optimization hints.

Consider an imaginary testing advisor, for example. In its most rudimentary form, the diagnostic expert system is very like a pilot's checklist, just enumerating suggestions as to possible causes of difficulty. As more and more system structure is embodied in the rule base, and as the rule base takes more sophisticated account of the trouble symptoms reported, the testing advisor becomes more and more particular in the hypotheses it generates and the tests it recommends. Such an expert system may depart most radically from the conventional ones in that its rule base should probably be hierarchically modularized in the same way the corresponding software product is, so that as the product is modularly modified, the diagnostic rule base can be modularly modified as well.

The work required to generate the diagnostic rules is work that would have to be done anyway in generating the set of test cases for the modules and for the system. If it is done in a suitably general manner, with both a uniform structure for rules and a good inference engine available, it may actually reduce the total labor of generating bring-up test cases, and help as well with lifelong maintenance and modification testing. In the same way, one can postulate other advisors, probably many and probably simple, for the other parts of the software-construction task.

Many difficulties stand in the way of the early realization of useful expert-system advisors to the program developer. A crucial part of our imaginary scenario is the development of easy ways to get from program-structure specification to the automatic or semiautomatic generation of diagnostic rules. Even more difficult and important is the twofold task of knowledge acquisition: finding articulate, self-explanatory experts who know why they do things, and developing efficient techniques for extracting what they know and distilling it into rule bases. The essential prerequisite for building an expert system is to have an expert.

The most powerful contribution by expert systems will surely be to put at the service of the inexperienced programmer the experience and accumulated wisdom of the best programmers. This is no small contribution. The gap between the best software engineering practice and the average practice is very wide—perhaps wider than in any other engineering discipline. A tool that disseminates good practice would be important.

"Automatic" programming. For almost 40 years, people have been anticipating and writing about "automatic programming," or the generation of a program for solving a problem from a statement of the problem specifications. Some today write as if they expect this technology to provide the next breakthrough.

Parnas \textsuperscript{4} implies that the term is used for glamour, not for semantic content, asserting,

In short, automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer.

He argues, in essence, that in most cases it is the solution method, not the problem, whose specification has to be given.

One can find exceptions. The technique of building generators is very powerful, and it is routinely used to good advantage in programs for sorting. Some systems for integrating differential equations have also permitted direct specification of the problem, and the systems have assessed the parameters, chosen from a library of methods of solution, and generated the programs.

These applications have very favorable properties:

* The problems are readily characterized by relatively few parameters.
* There are many known methods of solution to provide a library of alternatives.
* Extensive analysis has led to explicit rules for selecting solution techniques, given problem parameters.

It is hard to see how such techniques generalize to the wider world of the ordinary software system, where cases with such neat properties are the exception. It is hard even to imagine how this breakthrough in generalization could occur.

Graphical programming. A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design. Sometimes the promise held out by such an approach is postulated by analogy with VLSI chip design, in which computer graphics plays so fruitful a role. Sometimes the theorist justifies the approach by considering flowcharts as the ideal program-design medium and by providing powerful facilities for constructing them.

Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will.

In the first place, as I have argued elsewhere, \textsuperscript{8} the flowchart is a very poor abstraction of software structure. Indeed, it is best viewed as Burks, von Neumann, and Goldstine's attempt to provide a desperately needed high-level control language for their proposed computer. In the pitiful, multipage, connection-boxed form to which the flowchart has today been elaborated, it has proved to be useless as a design tool—programmers draw
flowcharts after, not before, writing the programs they describe.

Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram. The so-called "desktop metaphor" of today's workstation is instead an "airplane-seat" metaphor. Anyone who has shuffled a lap full of papers while seated between two portly passengers will recognize the difference—one can see only a very few things at once. The true desktop provides overview of, and random access to, a score of pages. Moreover, when fits of creativity run strong, more than one programmer or writer has been known to abandon the desktop for the more spacious floor. The hardware technology will have to advance quite substantially before the scope of our scopes is sufficient for the software design task.

More fundamentally, as I have argued above, software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview. The VLSI analogy is fundamentally misleading—a chip design is a layered two-dimensional description whose geometry reflects its realization in 3-space. A software system is not.

Program verification. Much of the effort in modern programming goes into testing and the repair of bugs. Is there perhaps a silver bullet to be found by eliminating the errors at the source, in the system-design phase? Can both productivity and product reliability be radically enhanced by following the profoundly different strategy of proving designs correct before the immense effort is poured into implementing and testing them?

I do not believe we will find productivity magic here. Program verification is a very powerful concept, and it will be very important for such things as secure operating-system kernels. The technology does not promise, however, to save labor. Verifications are so much work that only a few substantial programs have ever been verified.

Program verification does not mean error-proof programs. There is no magic here, either. Mathematical proofs also can be faulty. So whereas verification might reduce the program-testing load, it cannot eliminate it.

More seriously, even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.

Environments and tools. How much more gain can be expected from the exploding researches into better programming environments? One's instinctive reaction is that the big-payoff problems—hierarchical file systems, uniform file formats to make possible uniform program interfaces, and generalized tools—were the first attacked, and have been solved. Language-specific smart editors are developments not yet widely used in practice, but the most they promise is freedom from syntactic errors and simple semantic errors.

Perhaps the biggest gain yet to be realized from programming environments is the use of integrated database systems to keep track of the myriad details that must be recalled accurately by the individual programmer and kept current for a group of collaborators on a single system. Surely this work is worthwhile, and surely it will bear some fruit in both productivity and reliability. But by its very nature, the return from now on must be marginal.

Workstations. What gains are to be expected for the software art from the certain and rapid increase in the power and memory capacity of the individual workstation? Well, how many MIPS can one use fruitfully? The composition and editing of programs and documents is fully supported by today's speeds. Compiling could stand a boost, but a factor of 10 in machine speed would surely leave think-time the dominant activity in the programmer's day. Indeed, it appears to be so now.

More powerful workstations we surely welcome. Magical enhancements from them we cannot expect.

Promising attacks on the conceptual essence

Even though no technological breakthrough promises to give the sort of magical results with which we are so familiar in the hardware area, there is both an abundance of good work going on now, and the promise of steady, if unspectacular progress.

All of the technological attacks on the accidents of the software process are fundamentally limited by the productivity equation:

$$\text{time of task} = \sum \left(\text{frequency}\right) \times \left(\text{time}\right)$$

If, as I believe, the conceptual components of the task are now taking most of the time, then no amount of activity on the task components that are merely the expression of the concepts can give large productivity gains.

Hence we must consider those attacks that address the essence of the software problem, the formulation of these complex conceptual structures. Fortunately, some of these attacks are very promising.

Buy versus build. The most radical possible solution for constructing software is not to construct it at all.

Every day this becomes easier, as more and more vendors offer more and better software products for a dizzying variety of applications. While we software engineers have labored on production methodology, the personal-computer revolution has created not one, but many, mass markets for software. Every newsstand carries monthly magazines, which sorted by machine type, advertise and review dozens of products at prices from a few dollars to a few hundred dollars. More specialized sources offer very powerful products for the workstation and other Unix markets. Even software tools and environments can be bought off-the-shelf. I have elsewhere proposed a marketplace for individual modules. Any such product is cheaper to buy than to build afresh. Even at a cost of one hundred thousand dollars, a purchased piece of software is costing only about as much as one programmer-year. And delivery is immediate! Immediate at least for products that really exist, products whose developer can refer products to a happy user. Moreover, such products tend to be much better documented and somewhat better maintained than home-grown software.
The development of the mass market is, I believe, the most profound long-run trend in software engineering. The cost of software has always been development cost, not replication cost. Sharing that cost among even a few users radically cuts the per-user cost. Another way of looking at it is that the use of n copies of a software system effectively multiplies the productivity of its developers by n. That is an enhancement of the productivity of the discipline and of the nation.

The key issue, of course, is applicability. Can I use an available off-the-shelf package to perform my task? A surprising thing has happened here. During the 1950's and 1960's, study after study showed that users would not use off-the-shelf packages for payroll, inventory control, accounts receivable, and so on. The requirements were too specialized, the case-to-case variation too high. During the 1980's, we find such packages in high demand and widespread use. What has changed?

Not the packages, really. They may be somewhat more generalized and somewhat more customizable than formerly, but not much. Not the applications, either. If anything, the business and scientific needs of today are more diverse and complicated than those of 20 years ago.

The big change has been in the hardware/software cost ratio. In 1960, the buyer of a two-million dollar machine felt that he could afford $250,000 more for a customized payroll program, one that slipped easily and nondisruptively into the computer-hostile social environment. Today, the buyer of a $50,000 office machine cannot conceivably afford a customized payroll program, so he adapts the payroll procedure to the packages available. Computers are now so commonplace, if not yet so beloved, that the adaptations are accepted as a matter of course.

There are dramatic exceptions to my argument that the generalization of software packages has changed little over the years: electronic spreadsheets and simple database systems. These powerful tools, so obvious in retrospect and yet so late in appearing, lend themselves to myriad uses, some quite unorthodox. Articles and even books now abound on how to tackle unexpected tasks with the spreadsheet. Large numbers of applications that would formerly have been written as custom programs in Cobol or Report Program Generator are now routinely done with these tools.

Many users now operate their own computer 24 hours a day and day out on various applications without ever writing a program. Indeed, many of these users cannot write new programs for their machines, but they are nevertheless adept at solving new problems with them.

I believe the single most powerful software-productivity strategy for many organizations today is to equip the computer-naive intellectual workers who are on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs and then to turn them loose. The same strategy, carried out with generalized mathematical and statistical packages and some simple programming capabilities, will also work for hundreds of laboratory scientists.

Requirements refinement and rapid prototyping. The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification. Even the simple answer—"Make the new software system work like our old manual information-processing system"—is in fact too simple. One never wants exactly that. Complex software systems are, moreover, things that act, that move, that work. The dynamics of that action are hard to imagine. So in planning any software-design activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition.

I would go a step further and assert that it is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.

Therefore, one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements.

A prototype software system is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same hardware speed, size, or cost constraints. Prototypes typically perform the mainline tasks of the application, but make no attempt to handle the exceptional tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability.

Much of present-day software-acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software-acquisition problems...
spring from that fallacy. Hence, they cannot be fixed without fundamental revision—revision that provides for iterative development and specification of prototypes and products.

Incremental development—grow, don't build, software. I still remember the jolt I felt in 1958 when I first heard a friend talk about building a program, as opposed to writing one. In a flash he broadened my whole view of the software process. The metaphor shift was powerful, and accurate. Today we understand how like other building processes the construction of software is, and we freely use other elements of the metaphor, such as specifications, assembly of components, and scaffolding.

The building metaphor has outlived its usefulness. It is time to change again. If, as I believe, the conceptual structures we construct today are too complicated to be specified accurately in advance, and too complex to be built faultlessly, then we must take a radically different approach.

Let us turn to nature and study complexity in living things, instead of just the dead works of man. Here we find constructs whose complexities thrill us with awe. The brain alone is intricate beyond mapping, powerful beyond imitation, rich in diversity, self-protecting, and self-renewing. The secret is that it is grown, not built.

So it must be with our software systems. Some years ago Harlan Mills proposed that any software system should be grown by incremental development. That is, the system should first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then, bit by bit, it should be fleshed out, with the subprograms in turn being developed—into actions or calls to empty stubs in the level below. I have seen most dramatic results since I began urging this technique on the project builders in my Software Engineering Laboratory class. Nothing in the past decade has so radically changed my own practice, or its effectiveness. The approach necessitates top-down design, for it is a top-down growing of the software. It allows easy backtracking. It lends itself to early prototypes. Each added function and new provision for more complex data or circumstances grows organically out of what is already there.

The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one. Efforts re-double when the first picture from a new graphics software system appears on the screen, even if it is only a rectangle. One always has, at every stage in the process, a working system. I find that teams can grow much more complex entities in four months than they can build.

The same benefits can be realized on large projects as on my small ones. Great designers. The central question in how to improve the software art centers, as it always has, on people.

We can get good designs by following good practices instead of poor ones. Good design practices can be taught. Programmers are among the most intelligent part of the population, so they can learn good practice. Hence, a major thrust in the United States is to promulgate good modern practice. New curricula, new literature, new organizations such as the Software Engineering Institute, all have come into being in order to raise the level of our practice from poor to good. This is entirely proper.

Nevertheless, I do not believe we can make the next step upward in the same way. Whereas the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a creative process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge.

The differences are not minor—they are rather like the differences between Salieri and Mozart. Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort. The differences between the great and the average approach an order of magnitude.

A little retrospection shows that although many fine, useful software systems have been designed by committees and built as part of multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers. Consider Unix, APL, Pascal, Modula, the Smalltalk interface, even Fortran; and contrast them with Cobol, PL/I, Algol, MVS/370, and MS-DOS. (See Table 1.)

Hence, although I strongly support the technology-transfer and curriculum-development efforts now under way, I think the most important single effort we can mount is to develop ways to grow great designers.

No software organization can ignore this challenge. Good managers, scarce though they be, are no scarcer than good designers. Great designers and great managers are both very rare. Most organizations spend considerable effort in finding and cultivating the management prospects; I know of none that spends equal effort in finding and developing the great designers upon whom the technical excellence of the products will ultimately depend.

My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded. Not only salary, but the perquisites of recognition—office size, furnishings, personal technical equipment, travel funds, staff support—must be fully equivalent.

How to grow great designers? Space does not permit a lengthy discussion, but some steps are obvious:

- Systematically identify top designers as early as possible. The best are often not the most experienced.
- Assign a career mentor to be responsible for the development of the prospect, and carefully keep a career file.
- Devise and maintain a career-development plan for each prospect, including carefully selected apprenticeships with top designers, episodes of advanced formal education, and short courses, all interspersed with solo-design and technical-leadership assignments.
- Provide opportunities for growing designers to interact with and stimulate each other.
Acknowledgments

I thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick, and, most especially, David Parnas for their insights and stimulating ideas, and Rebekah Bierly for the technical production of this article.

References