Software's Chronic Crisis

by W. Wayt Gibbs, *staff writer*

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

"Software's Chronic Crisis" by W.W. Gibbs from *Scientific American*, Sept. 1994, pp. 86–95.
Reprinted with permission. Copyright © 1994 by Scientific American, Inc. All rights reserved.

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.

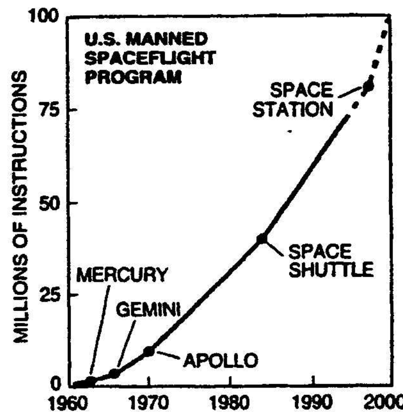
"Unfortunately, the industry does not uniformly apply that which is well-known best practice," laments Larry E. Druffel, director of Carnegie Mellon University's Software Engineering Institute. In fact, a research innovation typically requires 18 years to wend its way into the repertoire of standard 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.

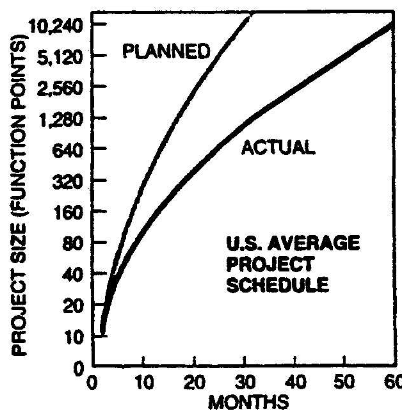
Shifting Sands

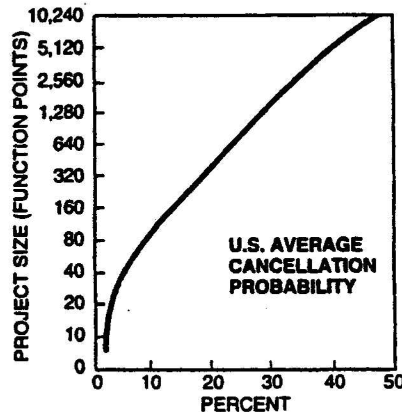
"We will see massive changes [in computer use] over the next few years, causing the initial personal computer revolution to pale into comparative insignificance," concluded 22 leaders in software development from academia, industry and research laboratories this past April. The experts gathered at Hedsor 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 embedded 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.







SOFTWARE IS EXPLODING in size as society comes to rely on more powerful computer systems (*top*). That faith is often rewarded by disappointment as most large software projects overrun their schedules (*middle*) and many fail outright (*bottom*)—usually after most of the development money has been spent.

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.A.F. slang: 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 graunch 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 virtuosic, $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 graunch 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," Randell 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 Stix; 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 the AAS software. One reason for the exorbitant price is that "on average every line of code developed needs to be rewritten once," bemoaned an internal FAA report.

Alarmed by skyrocketing costs and tests that showed the half-completed system to be unreliable, FAA administrator David R. Hinson decided in June to cancel two of the four major parts of the AAS and to scale back a third. The $144 million spent on these failed programs is but a drop next to the $1.4 billion invested in the fourth and central piece: new workstation software for air-traffic controllers.

That project is also spiraling down the drain. Now running about five years late and more than $1 billion over budget, the bug-infested program is being scoured by software experts at Carnegie Mellon and the Massachusetts Institute of Technology to determine whether it can be salvaged or must be canceled outright. The reviewers are scheduled to make their report in September.

Disaster will become an increasingly common and disruptive part of software development unless programming takes on more of the characteristics of an engineering discipline rooted firmly in science and mathematics [*see box on page 92*]. Fortunately, that trend has already begun. Over the past decade industry leaders have made significant progress toward understanding how to measure, consistently and quantitatively, the chaos of their development processes, the density of errors in their products and the stagnation of their programmers' productivity. Researchers are already taking the next step: finding practical, repeatable solutions to these problems.

Proceeds of Process

In 1991, for example, the Software Engineering Institute, a software think tank funded by the military, unveiled its Capability Maturity Model (CMM). "It provides a vision of software engineering and management excellence," beams David Zubrow, who leads a project on empirical methods at the institute. The CMM has at last persuaded many programmers to concentrate on measuring the process by which they produce software, a prerequisite for any industrial engineering discipline.

Using interviews, questionnaires and the CMM as a benchmark, evaluators can grade the ability of a programming team to create predictably software that meets its customers' needs. The CMM uses a five-level scale, ranging from chaos at level 1 to the paragon of good management at level 5. To date, 261 organizations have been rated.

"The vast majority—about 75 percent—are still stuck in level 1," Curtis reports. "They have no formal process,

no measurements of what they do and 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.

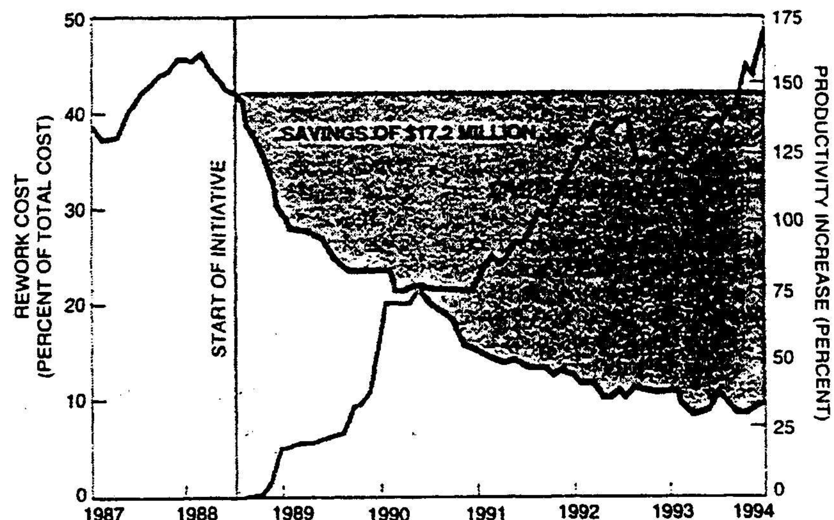
Mass-market software producers, because they have no single customer to please, can take a belated and brute-force approach to bug removal: they release the faulty product as a "beta" version and let hordes of users dig up the glitches. According to Charles Simonyi, a chief architect at Microsoft, the 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.



SOURCE: Raytheon

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

Engineering 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.

CHEMICAL ENGINEERING

SOFTWARE ENGINEERING

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 occasionally make mistakes in proofs. Secondly, formal methods can guarantee only that software meets its specification, not that it can handle the surprises of the real world.

Formal methods have other problems as well. Ted Ralston, director of strategic planning 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 innovations intended to boost productivity—many have even presented demonstration projects to "prove" the verity of their boasts. Advocates of object-oriented analysis and programming, a buzzword du jour, claim their approach represents a paradigm shift that will deliver "a 14-to-1 improvement in productivity," along with higher quality and easier maintenance, all at reduced cost.

There are reasons to be skeptical. "In the 1970s structured programming was also touted as a paradigm shift," Curtis recalls. "So was CASE [computer-assisted software engineering]. So were third-, fourth- and fifth-generation languages. We've heard great promises for technology, many of which weren't delivered."

Meanwhile productivity in software development has lagged behind that of more mature disciplines, most notably computer hardware engineering. "I think of software as a cargo cult," Cox says. "Our main accomplishments were imported from this foreign culture of hardware engineering—faster machines and more memory." Fisher tends to agree: adjusted for inflation, "the value added per worker in the industry has been at $40,000 for two decades," he asserts. "We're not seeing any increases."

"I don't believe that," replies Richard A. DeMillo, a professor at Purdue University and head of the Software Engineering Research Consortium. "There has been improvement, but everyone uses different definitions of productivity." A recent study published by Capers Jones—but based on necessarily dubious historical data—states that U.S. programmers churn out twice as much code today as they did in 1970.

The fact of the matter is that no one really knows how productive software developers are, for three reasons. First, less than 10 percent of American companies consistently measure the productivity of their programmers.

Second, the industry has yet to settle on a useful standard unit of measurement. Most reports, including those published in peer-reviewed computer science journals, express productivity in terms of lines of code per worker per month. But programs are written in a wide variety of languages and vary enormously in the complexity of their operation. Comparing the number of lines written by a Japanese programmer using C with the number produced by an American using Ada is thus like comparing their salaries without converting from yen to dollars.

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. Because attempts to scale promising technologies to industrial proportions so often fail, small laboratories are of limited utility. "We need to have places where we can gather data and try things out," DeMillo says. "The only way to do that is to have a real software development organization as a partner."

There have been only a few such partnerships. Perhaps the most successful is the Software Engineering Laboratory, a consortium of NASA's Goddard Space Flight Center, Computer Sciences Corp. and the University of Maryland. Basili helped to found the laboratory in 1976. Since then, graduate students and NASA programmers have collaborated on "well over 100 projects," Basili says, most having to do with building ground-support software for satellites.

Just Add Water

Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be reused at many different scales. Programmers have for decades used libraries of subroutines to avoid rewriting the same code over and over. But these components break down when they are moved to a different programming language, computer platform or operating environment. "The tragedy is that as hardware becomes obsolete, an excellent expression of a sorting algorithm written in the 1960s has to be rewritten," observes Simonyi of Microsoft.

Fisher sees tragedy of a different kind. "The real price we pay is that as a specialist in any software technology you cannot capture your special capability in a product. If you can't do that, you basically can't be a specialist." Not that some haven't tried. Before moving to NIST last year, Fisher founded and served as CEO of Incremental Systems. "We were truly world-class in three of the component technologies that go into compilers but were not as good in the other seven or so," he states. "But we found that there was no practical way of selling compiler components; we had to sell entire compilers."

So now he is doing something about that. In April, NIST announced that it was creating an Advanced Technology Program to help engender a market for component-based software. As head of the program, Fisher will be distributing $150 million in research grants to software companies willing to attack the technical obstacles that currently make software parts impractical.

The biggest challenge is to find ways of cutting the ties that inherently bind programs to specific computers and to other programs. Researchers are investigating several promising approaches, including a common language that could be used to describe software parts, programs that reshape components to match any environment, and components that have lots of optional features a user can turn on or off.

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. Belady, 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.

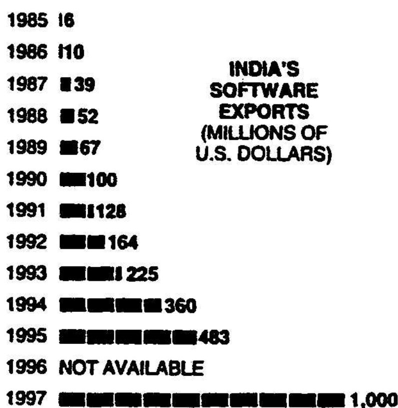
The single most important factor, Pawar asserts, is the support of the Indian government, which has eased tariffs and restrictions, subsidized numerous software technology parks and export zones, and doled out five-year tax exemptions to software exporters. "The opening of the Indian economy is acting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



nent. "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.

FURTHER READING

ENCYCLOPEDIA OF SOFTWARE ENGINEERING. Edited by John J. Marciniak. John Wiley & Sons, 1994.

SOFTWARE 2000: A VIEW OF THE FUTURE. Edited by Brian Randell, Gill Ringland and Bill Wulf. ICL and the Commission of European Communities, 1994.

FORMAL METHODS: A VIRTUAL LIBRARY. Jonathan Bowen. Available in hypertext on the World Wide Web as http://www.comlab.ox.ac.uk/archive/formal-methods.html