

No Silver Bullet

Essence and Accidents of Software Engineering

Computer Magazine; April 1987

by Frederick P. Brooks, Jr.,

University of North Carolina at Chapel Hill

This article was First Published In Information Processing 1986, ISBN No. 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V. (North-holland) IFIP 1986. Scanned from a poor-quality faxed copy by Brad Cox which is the reason for occasional OCR errors.

I ([Brad Cox](#)) have written several articles ([No Silver Bullet Revisited](#)) and a book ([Superdistribution: Objects as Property on the Electronic Frontier](#)), and founded a company ([Superdistribution Inc](#)) based on a disagreement with the main conclusion of this article.

Briefly, Brooks argues that software disappoints because it is so different from tangible goods, being made of bits not atoms, that the solutions that make tangible goods meet our expectations cannot possibly apply. This assumption is deeply buried and only evident in the fact that Brooks neglected to even mention it as a solution (e.g. as a "silver bullet") in his article.

I argue that robust markets can and will apply to software once we've provided the mechanisms described in the book and markets have evolved that rely on them. This would put software on the same price-performance curve that tangible goods enjoy today.

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 non-technical 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 breakthroughs--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 numerous 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 two fold 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, and 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" level [BJC:]. The physicist labors on, however, in a firm faith that there are unifying principles to be found, whether in quarks or in unified field theory. Einstein argued that there must be simplified [BJC: undecipherable] 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.

Change:

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 [BJC: indecipherable] copies of the same basic design. Callbacks of automobiles are ready 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 changes.

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, than 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 re*ic'dng and dmPlifYinB [bjc: indecipherable] software structures, they remain inherently unvisualizable, and they 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, and 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, and 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.

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 involved 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.

Timesharing

Time-sharing brought a major improvement in the productivity of programmers and in the quality of their products, 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. The principal effect of time-sharing is to shorten system response time. As this response time goes to zero, at some point it passes the human threshold of noticeability, about 100 milliseconds. Beyond that threshold, no benefits are to be expected.

Integrated Program Development 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?

One of these developments 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^[1].

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 any of the other technical fads of the day[2]. I am among them. Mark Sherman of Dartmouth notes on CSnet News that one must be careful to distinguish two separate ideas that go under that name: abstract data types and hierarchical types. The concept of the abstract data type is that an object's type should be defined by a name, a set of proper values, and a set of proper operations rather than by its storage structure, which should be hidden. Examples are Ada packages (with private types) and Modula's modules.

Hierarchical types, such as Simula-7's classes, allow one to define general interfaces that can be further refined by providing subordinate types. The two concepts are orthogonal--one may have hierarchies without hiding and hiding without hierarchies. Both concepts represent real advances in the art of building software.

Each removes yet another accidental difficulty from the process, allowing the designer to express the essence of the design without having to express large amounts of syntactic material that add no information content. For both abstract types and hierarchical types, the result is to remove a higher-order kind of accidental difficulty and allow a higher-order expression of design.

Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential, and such attacks make no change whatever in that. An order-of-magnitude gain can be made by object-oriented programming only if the unnecessary type specification underbrush still in our programming language is itself nine-tenths of the work involved in designing a program product. I doubt it.

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[3]. I do not. To see why, we must dissect what is meant by "artificial intelligence."

D.L. Parnas has clarified the terminological chaos[4]: 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 programming. 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 in

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[[3]-[5]]. 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 tracing 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 encoded 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 difference engine available, it may actually reduce the labor of generating bring-up test cases, and bdp [bjc: indecipherable] 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 analytical 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 5

Parnas 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 [[sterling]] 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 Ph.D. dissertations in software engineering is graphical, or

visual, programming--the application of computer graphics to software design[[6][7]] 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[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

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 * argument that the generation of software packages has changed little over the years: electronic spreadsheets and simple database systems. These powerful tools, so obvious in retrospect and *t 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 formally 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 computers day in 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-naïve 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 up the detail necessary for specification. Even the simple answer--"Make the new software system work like our old manual information-processing system" --is in far 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 [\[9\]](#) [\[10\]](#) [\[11\]](#)[\[12\]](#)

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.

Frederick P. Brooks is Kenan Professor of Computer Science at the University of North Carolina in Chapel Hill. He is best known as the "father of the IBM System/360 computer family," having served as project manager for the System/360 hardware and later as project manager for the Operating System/360 software.

At Chapel Hill, Brooks founded the UNC Dept. of Computer Science and has participated in the establishment and guiding of the Microelectronics Center of North Carolina, the Triangle Universities Computation Center, and the North Carolina Educational Computing Service. He has received the National Medal of Technology, a Guggenheim fellowship, and the McDowell and Computer Pioneer awards of the Computer Society of the IEEE.

Brooks received his Ph.D. (in what is today computer science) from Harvard, where he was a student of Howard Aiken.

Readers may write to F.P. Brooks at the University of North Carolina, Dept. of Computer Science, Chapel Hill, NC 27514.

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.

Skeptics 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 being bitten 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 stealth and cunning.

Enter the fictional, not legendary, werewolf. The first major werewolf movie, *The Werewolf of London*, in 1935 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 mark 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 murderous tendencies.

It is worth noting that the film tradition often makes the werewolf a rather sympathetic character, an Innocent transformed against 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.

References

- [1] D.L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Trans. Software Engineering, Vol. 5, No. 2, Mar. 1979, pp. 128-138.
- [2] G. Booch, "Object-oriented Design," Software Engineering with Ada, 1983, Benjamin/Cummings, Menlo Park, Calif.
- [3] IEEE Trans. Software Engineering (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., Vol. 11, No. 11, Nov. 1985.
- [4] D.L. Parnas, "Software Aspects of Strategic Defense Systems," American Scientist, Nov. 1985.
- [5] R. Baker, "A 15-Year Perspective on Automatic Programming," IEEE Trans. Software Engineering (special issue on artificial intelligence and software engineering), *. Mostow, guest ed., Vol. 11, No. 11, Nov. 1985, pp. 1257-1267.
- [6] Computer (special issue on visual programming), R.B. Gaphton and T. Ichikawa, guest eds., Vol. 18, No. 8, Aug. 1985.
- [7] G. Raeder, "A Survey of Current Graphical Programming Techniques," Computer (special issue on visual programming), R.B. Gaphton and T. Ichikawa, guest eds., Vol.18, No. 8, Aug. 1985, pp. 11-25.
- [8] F.P. Brooks, The Mythical Man-Month, 1975, Addison-Wesley, Reading, Mass., New York, Chapter 14.
- [9] Defense Science Board, Report of the Task Force on Military Software, in press.
- [10] H.D. Mills, "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Ruskin, ed., Prentice-Hall, Englewood Cliffs, N.*., 1971.
- [11] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," 1985, TRW tech. report 21-371-85, TRW, Inc., I Space Park, Redondo Beach, CA 90278.
- [12] H. Sackman, W.J. Erikson, and E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," CACM, Vol. II, No. I, Jan. 1968, pp.3-11.