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 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 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 humour 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 processes 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 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 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 whisks 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. 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.

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 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 strength 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 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

April 1987
velopments is Ada, a general-purpose high-level language of the 1980's. Ada not
only reflects evolutionary improvements in language concepts, but indeed em-
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 sub-
setted working vocabularies can solve the
learning problem, and hardware advances
will give us the cheap MIPS to pay for the
compiling costs. Advancing the structur-
ing 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 hard-
ware 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 big-
gest 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 ac-
cidents 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 dif-
ference, 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 contribu-
tion will be that switching to it occasion-
ioned 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 Dart-
mouth 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 opera-
tions 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-67's
classes, allow one to define general in-
faces that can be further refined by pro-
viding subordinate types. The two con-
cepts are orthogonal—one may have hierarchies without hiding and hiding
without hierarchies. Both concepts repre-
sent 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

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

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 expres-
sion of design.

Nevertheless, such advances can do no
more than to remove all the accidental dif-
ficulties from the expression of the design.

The complexity of the design itself is essen-
tial, and such attacks make no change
whatever in that. An ordjer-of-magnitude
gain can be made by object-oriented pro-
gramming only if the unnecessary type-
specification underbrush still in our pro-
gramming language is itself nine-tenths of
the work involved in designing a program
product. I doubt it.

Artificial intelligence. Many people ex-
pect 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 termin-
ological 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 apply-
ing 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 ex-
erts 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 mean-
g. . . Something can fit the definition of
AI-1 today but, once we see how the
program works and understand the prob-
lem, we will not think of it as AI any
more. . . . Unfortunately I cannot iden-
tify a body of technology that is unique
to this field. . . . Most of the work is prob-
lem-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 recogni-
tion 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 pro-
gramming practice. The same problem is
ture of speech recognition. The hard thing
about building software is deciding what
one wants to say, not saying it. No facilita-
tion of expression can give more than mar-
ginal 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 soft-
ware scientists are hard at work applying
this technology to the software-building
evironment. 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 retraction its reasoning
for the user. The inference engines typ-
ically can deal with fuzzy or probabilistic
data and rules, in addition to purely deter-
ministic logic.

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

• Inference-engine technology is de-
veloped in an application-dependent
way, and then applied to many uses.
One can justify much effort on the in-
ference 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, semi-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 glamor, 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:

- 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.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 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 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 pro-

Language-specific smart editors promise at most freedom from syntactic errors and simple semantic errors.

gram 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:

\[ \text{time of task} = \sum_i (frequency)_i \times (time)_i \]

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 newstand 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. 9

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 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 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 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 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. 10 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-

---

**Table 1. Exciting vs. useful but unexciting software products.**

<table>
<thead>
<tr>
<th>Yes Exciting Products</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix</td>
<td>Cobol</td>
</tr>
<tr>
<td>APL</td>
<td>PL/1</td>
</tr>
<tr>
<td>Pascal</td>
<td>Algol</td>
</tr>
<tr>
<td>Modula</td>
<td>MVS/370</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>MS-DOS</td>
</tr>
</tbody>
</table>

...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. 11

**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. 12 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/1, 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 Biery for the technical production of this article.

References


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 1961 at 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.
Biting the Silver Bullet

Toward a Brighter Future for System Development

David Harel, Weizmann Institute of Science

In an eloquent and thoughtful 1986 article, Frederick Brooks expresses his feelings about the illusions and hopes software engineering offers. He argues that many proposed ideas are not "silver bullets" that will deliver us from the horrors of developing complex systems.

Brooks' article is reminiscent of Parnas' series of minipapers that accompanied his widely publicized resignation from the Strategic Defense Initiative Organization (SDIO) Panel on Computing in 1985. Parnas claims that current proposals are vastly inadequate to build reliable software as complex as that required for the SDI project.

We thus have two rather discouraging position papers, authored by two of the most influential figures in the software world. Neither is a critique of software engineering per se, although both make an effort to dissolve myths of magical power that people have cultivated concerning certain trends in the field.

This article was triggered by those of Brooks and Parnas. It is not a rebuttal. Indeed, I agree with most of the specific points made in both papers. Instead, the goal of this article is to illuminate the brighter side of the coin, emphasizing developments in the field that were too recent or immature to have influenced Brooks and Parnas when they wrote their manuscripts.

The two main aspects of these developments have to do with a carefully wrought "vanilla" approach to system modeling and the emergence of powerful methods to execute and analyze the resulting models. It can be argued that the combined effect of these and other ideas is already showing positive signs and appears to have the potential to provide a truly major improvement in our present abilities — profoundly affecting the essence of the problem. This might take more than the 10 years Brooks focuses on. It will surely be a long time before reliable software for the likes of the SDI project can be built. Such a system remains an order of magnitude too large and too critical to construct today, mainly because of its first-time-must-work nature. But I also believe that we are on the royal (main) road and that the general impression you get from reading the Brooks and Parnas articles is far too bleak.

A "vanilla" approach to modeling, together with powerful notions of executability and code generation, may have a profound impact on the "essence" of developing complex systems.
Past versus present

Brooks' arguments. The main problem, as Brooks rightly sees it, is in specifying, designing, and testing the "conceptual construct" underlying the system being developed, and not in "the labor of representing it and testing the fidelity of the representation."

"The hard thing about building software," he claims, "is deciding what one wants to say, not saying it." In elaborating, he mentions the superlinear growth in the number of system states, the difficulty of comprehending the conceptual construct and communicating it to others, and what he believes to be its inherent unvisualizable character.

Brooks further argues that, in contrast to their apparent appeal, several proposed ideas in the field do not constitute magical solutions to the essential problems. Among the "nombullets" he discusses are high-level languages, object-oriented programming, artificial intelligence and expert systems, automatic programming, graphical languages, program verification, and hardware improvements.

In his introduction, Brooks says that although he sees no startling breakthroughs in the next decade, "many encouraging innovations are under way," and eventually they will be exploited to "yield an order-of-magnitude improvement."

Brooks mentions two sets of innovations. The first set includes those of the above proposals that he doesn't totally discard (for example, high-level languages and object-oriented programming). However, he claims that they deal only with representation issues, which constitute the accidental part of the problem.

The second set of innovations, the ones Brooks claims will influence the essence, include

- buying sufficiently general ready-made software, instead of having it tailor-made;
- refining the requirements iteratively and interactively with the client, using increasingly better prototypes;
- enhancing the design in an iterative, top-down fashion, adding lower-level details at each step; and
- finding, hiring, and cultivating extremely talented designers.

Despite the encouraging way the points are expressed, we come away feeling distinctly uncomfortable. Apart from ideas that deal with the accidental parts of the problem, we are told to buy good software from others, hire better people than we already have, and continue with the well-established practices of prototyping and iterative design. All the rest is marginal.

I have discussed Brooks' article with many people over the past few years. Most stated that while they agree with many of its individual points, the paper presents a far gloomer assessment of the situation than seems appropriate. I feel that this is rooted in some of its underlying adopted themes.

The first is the sharp separation between the accidental and essential aspects of the problem, relegating everything related to representation, language, and levels of abstraction to the former and only the process of thinking about the concepts to the latter.

The second is the treatment of each proposed idea in isolation, with the accompanying claim that most of the proposals address representation, so that they cannot help with the essence.

The third involves concentrating on only 10 years of the future, which is probably too short a period in which to expect any significant improvement. (About half of this period is already behind us.)

Finally, the discussion is presented as a search for a miracle-working silver bullet that will slay the werewolf of constructing complex software. By arguing that current motifs will not bring about that miracle, at least not within the next few years, we are left with the troubling feeling that the werewolf is here to stay.

We've been there before. Since this article takes a longer term point of view, it is instructive to carry out a brief thought experiment. Let's go back, say 40 years. That was the time when instead of grappling with the design of large, complex systems, programmers were in the business of developing conventional one-person programs (which would be on the order of 100-200 lines in a modern programming language) that were to carry out limited algorithmic tasks.

Given the technology and methodology available then, such tasks were similarly formidable. Failures, errors, and missed deadlines were all around.

Imagine an article appearing then and claiming the essence of the problem to be deciding what one wants to say, that is, conceiving the algorithm. Writing the program is the accidental part. Such an article might have asked about the availability of a one-stroke solution that deals with the essence. From the way the issue is presented, it would follow that any ideas that relate to representation and levels of detail can be discounted, because they deal with the nonessential parts of the problem. The article would have very well go on to argue that ideas like high-level programming languages, compilation, and algorithmic paradigms can be safely set aside, since they do not deal with the essence.

However, while none of these ideas alone has solved the problem, and while it did take more than 10 years for the situation to change, we have indeed witnessed an order-of-magnitude advance in our ability to tackle the very

On biting bullets

There are two opinions about the origin of the phrase "Biting the bullet." One is that it came from the need to bite the top off the paper cartridge prior to firing a certain kind of British rifle used in the mid 19th century. This often had to be done under enemy fire and required keeping a cool head.

The other is that it is an old American phrase, rooted in the folklore of the US Civil War. It supposedly emerged from the practice of encouraging a patient who was to undergo field surgery to bite down hard on a lead bullet to "divert the mind from pain and prevent screaming" (R.L. Chapman, American Slang, Harper and Row, New York, 1986).

In more recent years, the phrase has come to signify having to do something painful but necessary, or to undertake an activity despite criticism or opposition, while exhibiting a measure of courage and optimism.
More on the vanilla approach

It is impossible to provide a detailed account of the vanilla approach to modeling in this article. The discussion of it in the text is thus extremely brief. As mentioned, the ideas are based on much early work on the specification and design of nonreactive systems, suitably extended.

The three independent efforts that led to this approach are described, respectively, in the Ward and Mellor book, the Hatley and Pirbhai book, and in the Harel et al. publication related to the Statemate system.

The latter is less informative on the modeling aspects of the approach than the two books; its main intention was to describe the supporting tool. However, a more detailed description of this modeling framework appears in the following manuscript, which should appear in book form in due time:


The following paper compares and evaluates these three research efforts (as well as a related fourth one). It is quite illuminating and emphasizes the differences between them, particularly those relevant to modeling behavior:


The following book contains interesting discussions and comparisons of these and other modeling approaches. It also features a valuable annotated bibliography of some 600 items:


The rest of this article is restricted to a class of systems that has been termed reactive.

The climate suggests that we stand to witness a grand-scale improvement in the current situation is similar, except that we are now in the business of developing very complex systems. These systems are to consist of large amounts of software and hardware and are often of a distributed nature. Their size and complexity, as Brooks and Parnas observe, is formidable when compared to one-person programs. By their very nature, they also involve large numbers of technical personnel.

Vanilla frameworks. Most instrumental in triggering the revolution in one-person programming has been the evolution of a fitting, general-purpose conceptual framework, which we shall call "vanilla." Its main contribution was to free the programmer from having to think on an inappropriate level of detail, enabling him or her to conceive of an idea for solving an algorithmic problem and to map it easily from the mind into an appropriate high-level medium.

The cornerstone of this framework is a collection of fundamental notions and concepts that includes the basic dichotomy between data and control and convenient means for structuring and combining them into an algorithmic whole. Thus, elementary control structures, data types, and data structures were identified, and we learned how to wield them. A rich variety of algorithmic methods was devised, including divide and conquer, dynamic programming, and greedy paradigms; these were adapted to fit a variety of problem sets. Notions of correctness and efficiency were introduced, together with methods for establishing the former and estimating the latter.

In parallel, and based on these concepts, a corresponding set of vanilla high-level programming languages evolved, supported by powerful and sophisticated tools for testing and analyzing. We learned to rely on the theory of computational complexity to help us find efficient algorithms or to detect our stumbling upon an intractable problem; we have begun to understand the great virtues of parallelism, approximation, and randomization in obtaining even better solutions.

Thus, for one-person programs, accidental and essential issues were intimately and unavoidably intertwined.

Of course, as time went by, other flavors, more exotic than vanilla, naturally emerged, such as applicative, functional, and logic programming styles, as well as more esoteric approaches like systolic arrays and neural nets. For each, the basic notions and concepts have had to be redefined, and new languages and tools have been developed. The arsenal has thus grown considerably and has become richer and more varied — a sure sign of healthy evolution.

Back to the future. I believe the current situation is similar, except that we are now in the business of developing very complex systems. These systems are to consist of large amounts of software and hardware and are often of a distributed nature. Their size and complexity, as Brooks and Parnas observe, is formidable when compared to one-person programs. By their very nature, they also involve large numbers of technical personnel.

The rest of this article is restricted to a class of systems that has been termed reactive.
the process of constructing such systems. It is hard to predict a time frame for this, but the scope of the benefits it will bring about could very well match the striking changes we have witnessed in solving one-person algorithmic problems.

We now discuss the two components of these developments: means for modeling the system, and techniques for inspecting and analyzing the model.

Modeling the system

To model systems, we need an underlying set of fundamental concepts and notions — some call them "abstractions" — that, in Brooks' terminology, capture the "conceptual construct" of complex systems. Deciding what they are and how they relate is analogous to the separation of data and control in the vanilla approach to one-person programs and the identification of appropriate ways of structuring, expressing, and combining them. For a nonexotic first cut at the problem, these concepts must be sufficiently general to be widely applicable, even at the expense of being somewhat mediocre. To be amenable to inspection and analysis, they must also be rigorous and precise, with underlying formal semantics.

The vanilla approach is rooted in the early work of Parnas and others on modularization and information hiding,1 and in that of several researchers on structured analysis and structured design2 that dealt mainly with data-intensive systems. The backbone of the system model should be a hierarchy of activities, as we'll call them, that capture the functional capabilities of the system — suitably decomposed to a level with which the designer is happy. (The activities need not be arranged in strict hierarchies. The breakup, or decomposition, may have overlappings, with elements on any level being shared by multiple parent elements. The term "hierarchy" used here thus carries a more flexible connotation.)

Data elements and data stores are also specified therein, and are associated as inputs and outputs that flow between the activities on the various levels. The semantics of this kind of functional description is dynamically noncommitting in that it merely asserts that activities can be active, information can flow, and so on. It does not contain information about what will happen, when it will happen, or why it will happen. As a consequence, this hierarchy can only serve as part of a conceptual model for truly reactive systems — such as control and communication systems or embedded real-time systems, which have a crucial behavioral side that has to be addressed, too.

Some time ago, a number of independent research groups extended these widely accepted ideas to deal with reactive systems.3,4 Their efforts resulted in a surprisingly similar set of conclusions. Using their own terminology and emphasis, they each recommended that the hierarchy of activities be enriched with behavioral descriptions we'll call control activities, which potentially appear on all levels.

Control activities serve as the central nervous system, so to speak, of the model. They are meant to sense and control the dynamics of that portion of the functional description on their level. This includes the ability to activate and deactivate activities, cause data to be read and written, and sense when such things have happened — thus affecting subsequent behavior. The resulting combination is the system's conceptual model.

The recommendations also call for a structural, or architectural, description of the system to deal with such notions as subsystems and modules, channels and physical links, and storage components. This description can thus be considered the system's physical model. The conceptual and physical models are related by a mapping that assigns implementation responsibility for the various parts of the former to those of the latter.

Modeling behavior. While the functional description is the backbone of the conceptual model, the behavioral descriptions (that is, the control activities) are, in a crucial sense, its heart and soul. Behavior over time is much less tangible than either functional- ity or physical structure, and more than anything else, this is the aspect that renders reactive systems so slippery and error-prone.

In the realm of dynamic behavior, there is a particularly dire need for approaches that are sufficiently clear and well-structured to enable designers to capture their thinking in a coherent and comprehensive fashion. Moreover, behavioral descriptions must possess rigorous underlying semantics; all too often, insufficient attention has been paid to semantics. The discussion of analysis below will show how important this is.

The aforementioned research groups5 more or less agree that behavioral controllers should be described using modes, or states, together with control elements, such as events and conditions, that trigger transitions between them. Implicitly, they have also adopted a subtle abstraction, termed the synchrony hypothesis,6 according to which everything takes zero time unless explicitly prescribed otherwise. However, there is no agreement as to exactly how this is to be turned into a workable medium for modeling reactive behavior. It is clear that conventional finite-state machines will not do, due to their lack of structure, their verbosity, and the notorious state-explosion phenomenon.

The basic elements of reactive behav-
ior (states, transitions, events, conditions, and time) must be allowed to be properly and naturally conceptualized, structured, and combined so that fundamental patterns of reactive behavior—like sequentiality, concurrency, and synchronization—will mesh smoothly with the functional decomposition.

A number of solutions have been suggested by these groups. They range from variants of communicating finite-state machines, through combinational decision tables and other similar means, to a relative newcomer—statecharts. Other formalisms, such as Petri nets, temporal logic, or certain languages especially tailored for real-time systems, would be reasonable choices, too.

**Modeling data.** Although data-intensive systems are not the subject of this article, a few words regarding the issue of incorporating data modeling into the vanilla framework are in order.

Conventional data elements and data structures can be specified and manipulated in standard ways within behavioral descriptions or in bottom-level activities. To deal with large-scale pools of data, such as databases or knowledge-bases, we would have to use a separate data-modeling medium, such as a suitably adapted version of Chen's entity-relationship approach. The resulting descriptions would then be associated with the data stores.

Incorporating data-modeling techniques into the present framework could serve as an excellent melting pot for combining ideas from the world of data-intensive systems with ones from the world of reactive systems.

**Strata of conceptual models.** Brooks states that descriptions of software that abstract away its complexity often also abstract away its essence—the complexity itself being part of the essence. Obviously, he is right. Indeed, it is important to use the vanilla approach in a way that does not hide the system's essential complexity. Proper use actually enables harnessing and taming that complexity by allowing the designer to capture the system's inherent conceptual structure in a natural way.

Regardless of how well devised it might be, one conceptual model might not be enough to take us from our initial thoughts to a final working implementation. While it is possible to construct a good functional hierarchy, interweaved with its controlling activities, the mapping we specify between that model and the physical model often turns out to be naive, rarely constituting a satisfactory full-fledged implementation. Consequently, we must often add a new dimension to the modeling process by repeatedly refining the conceptual model.

This can be done by preparing a new tier, or stratum, of functional hierarchies, one for each of the subsystems appearing in the structural view, and providing a lower-level mapping between these refined models and the subsystems themselves. This process may continue downward until a satisfactory level of design is reached.

These ideas are quite in line with Brooks' sympathetic discussion of top-down design. Of course, two crucial parts of this process concern the methodological issue of providing guidelines and heuristics for actually carrying it out, and the technical issue of showing consistency between the resulting strata. These are briefly discussed below.

**Visual representation.** Most issues of representation have been skirted above; indeed, some justification could be found in giving them second-class status. However, I believe that convenient media for representing the concepts and structures inherent in a model impact the very thinking that goes into constructing that model. In the one-person programming world, the availability of programming languages such as Pascal and C and even their precursors like FORTRAN, ALGOL, and PL/I has had a profound influence on a programmer's ability to conceive of good algorithms. Moreover, good representation is also instrumental in communicating those algorithms and their underlying ideas to others.

I agree with Brooks that flowcharts have become pitiful visualizations of programs, and even with a more general claim concerning the hopelessness of finding a general-purpose visual programming language that could replace conventional languages. But this opinion comes to a screeching halt where complex reactive systems are concerned.

Much of the conceptual construct underlying a complex reactive system is inherently topological in nature, and this is reflected in the vanilla approach outlined above. Hierarchies, with or without overlapping, and multilevel relationships, whether they concern structure, function, or behavior, can be captured naturally by simple, rigorous, and well-known notions from set theory and topology; these, in turn, have natural counterparts as spatial/graphical representations.

As argued elsewhere, this fact gives rise to visual formalisms, in which encapsulation, connectedness, and adjacency play central roles, and lesser features, such as size, shape, and color, can also be exploited. Furthermore, all these graphical features come complete with rigorous mathematical semantics.

Visual formalisms have indeed been proposed for representing the various aspects of vanilla models. From several years of following their application in large real-world projects, I have become convinced that using appropriate visual formalisms can have a spectacular effect on engineers and programmers. (An example of this is the avionics system for the state-of-the-art Lavi fighter at Israel Aircraft Industries, where the visual language of statecharts was used for specification. Although these experiences are too recent to have yielded statistics, some comparisons and evaluations have already appeared.)

Moreover, this effect is not limited to mere accidental issues; the quality and expedition of their very thinking was found to be improved. Successful system development in the future will revolve around visual representations. We will first conceptualize, using the "proper" entities and relationships, and then formulate and reformulate our conceptions as a series of increasingly more comprehensive models represented in an appropriate combination of visual languages. A combination it must be, since system models have several facets, each of which conjures up different kinds of mental images.
Of course, the job is far from complete. Some aspects of the modeling process have not been as forthcoming as others in lending themselves to good visualization. Algorithmic operations on variables and data structures, for example, will probably remain textual. In addition, as Brooks aptly observes, some of the less obvious connections between the various parts of system models are not easily visualized. However, for a number of years, we have been doing far, far better than the "several, general directed graphs, superimposed one upon another" Brooks describes. The graphical languages currently used are still two-dimensional, whereas some of the concepts could definitely do with higher-dimensional visualization. This may still happen. In fact, realistic motion-based 3D techniques are rapidly coming into reach. A new aspect of visual languages that will have to be addressed is computerized support for the "nice looking" layout of diagrams. This is a difficult and challenging problem in which only marginal progress has been made.

Regarding hardware, our scopes are currently of limited scope, to use Brooks' captivating phrase, making the extent to which we can comfortably display very large visual models dependent on the availability of dramatically improved graphical hardware. Rather than taking this as a reason to abandon visual approaches, we should find it enlightening. For once, concepts and software ideas are ahead, waiting for the development of matching hardware. If the past record of hardware improvements is any measure, these developments will not be long in coming.

It is our duty to forge ahead to turn system modeling into a predominantly visual and graphical process. I believe this is one of the most promising trends in our field.

Methods and guidelines. In addition to thinking with the "right" concepts and representing the resulting thoughts in appropriate programming languages, a programmer can call on a variety of well-established methods, guidelines, and techniques to help formulate a good solution to a one-person algorithmic problem. These constitute a large reservoir of knowledge accumulated over years, embodying the experience and expertise of generations of programmers, algorithm designers, and computer scientists. As might be expected, there has always been a great deal of cross-fertilization between the world of methods and techniques and the world of concepts and languages.

The story for complex reactive systems is no different, except that it is at a far more embryonic stage. Despite the proliferation of so-called methodologies, it is still too early to see a wide-ranging and well-understood collection of guidelines and techniques for the step-by-step process of system development.

Many of the proposed methodologies are not methodologies at all in that they do not contain recommendations about how to actually do things. For that matter, the vanilla approach described here is not a methodology either. However, what is worse is that many do prescribe recipes, but these often suffer from being too restrictive, hard to apply, or downright wrong.

One of the most unfortunate trends has been in presenting a method as exclusive, that is, preaching about its being the step-by-step way to develop entire systems. This can be compared in naïveté to someone advocating divide-and-conquer or branch-and-bound as the method to write programs.

The availability of a solid, general-purpose framework within which one can conceptualize, capture, and represent a system model seems to be far more important right now. All-encompassing recipes for how to get the work done simply do not exist; guidelines and techniques that work in special cases do exist, and more will surface in time. Obviously, they will be influenced by the choice of the framework and will, in turn, influence that framework and its evolution. And they will draw heavily on our experience in wielding the notions, concepts, languages, and tools.

Among the guidelines suggested are top-down and bottom-up approaches, which prescribe the raw order of things, as well as approaches related to the nature of the elements that drive the process, such as state-driven, function-driven, or data-driven.

In principle, all of these can be followed quite smoothly within the vanilla framework, though constructing really good models, as well as choosing which of these guidelines to use for what systems, will obviously remain something of an art.

Some methods are further removed from the vanilla framework, since they advocate a somewhat different set of basic concepts. One of the better-known examples is the object-oriented approach, in which objects and their capabilities take precedence over activities and states. While it is possible to follow this approach within the confines of our basic framework, it's perhaps not as smooth-going as one would like. This is an excellent example of a more specialized, or exotic, flavor, which is already resulting in correspondingly specialized advances in languages and tools.

In addition to guidelines for the overall process of development, a number of heuristics have been addressed at the nontrivial process of mapping the conceptual model onto the physical one. They are often based on taking subtle advantage of the cohesion and coupling of activities. For pure software systems, this task is usually less perplexing, since the structure of the final product can be taken to correspond reasonably well with that of the conceptual model. However, embedded systems are different. In them, the physical breakup into components and subcomponents might be acutely orthogonal to the conceptual structure. These cases require more iterations in the design process, giving rise to several strata of physical and conceptual models, as discussed above. The importance of such heuristics stems from such cases. (These heuristics could well find their way into useful expert-system support tools, as envisioned by Brooks.)

Designers would do well to master all of these techniques, guidelines, and heuristics, and to use them to devise models in a manner that they deem most natural. In time, I'm certain we will outgrow the deep convictions we have cultivated around the various methodologies. We will stop trying to get everyone to use
Tools for model execution

Although model execution is not a new idea, its vast potential has not yet been fully exploited. All the executability features discussed in the text are available in the Statemate tool, the first release of which was developed between 1984 and 1987. It is currently being used mainly in the areas of avionics, telecommunication, and process control.

A number of additional tools support some of these features. A couple of them are commercially available, and others are still in research and development stages. Here are a few additional publications describing techniques and tools for executing models:


Analyzing the model

The preceding sections have repeatedly invoked the analogy between conventional algorithms and models of complex systems. When it comes to semantics and analysis, this analogy takes on a particularly interesting twist.

Although the importance of testing and analyzing one-person algorithms has always been acknowledged, the world of complex systems has long suffered from something of an indifference to such needs. By analogy, the situation was as if we were asked to solve one-person algorithmic problems without the possibility of running programs, and hence without being able to test and debug them at all.

Indeed, many past approaches to system development provided no means for capturing behavior, being centered instead on the functional aspects and dataflow. The approaches that did provide such means were informal, lacking the rigorous semantics necessary for even beginning to analyze the dynamics. Hence, it was impossible to predict in early stages how the system would behave if constructed according to the model.

Not until actual code was written — usually at a very late stage in the project by people other than those responsible for and capable of developing the "conceptual construct" and at much greater expense — could one expect to get reliable answers to "what if?" questions. This, of course, has had a deplorable effect on the expedition and quality of development efforts for large and complex systems.

As a consequence, most computerized tools that flourished around such methods (computer-aided software engineering — CASE — tools, as they are often called) concentrated on providing mere graphic-editing capabilities, sometimes accompanied by document generation, version control, and project management facilities. Their proponents heralded the ability of these tools to check model "consistency and completeness," which is really just a grand form of syntax checking.

To use my analogy again, it is like making sure, in a conventional program, that the begins and ends match, that procedure calls have the right number and types of parameters, and that all declared variables are indeed used. In the complex system arena, such checking includes the consistency of level-labeling schemes and of inputs and outputs within the hierarchies, the nonredundancy of flow elements, and so on: it is analogous to the checking carried out in one-person programming environments on-the-fly or in simple precompilation stages.

Since designing a complex reactive system is so much more massive and intricate an undertaking than writing a conventional one-person program, testing for consistency and completeness in system modeling is far more important than syntax checking in programs. Nevertheless, it remains a mere test of the syntactic integrity of the model and has very little to do with that model's conceptual and logical aspects.

Checking that a model is consistent and complete cannot prevent logical errors that cause a missile to fire unintentionally or a stock market system to run amok — exactly the kinds of mishaps that are at the heart of our problem. For this, we need the ability to carry out real testing and analysis.

You may feel the following discussion is unrealistically futuristic. Not so. All the possibilities we mention have been implemented in a computerized tool that supports the vanilla approach and that is being used in the development of real systems. Several other tools also support some of these possibilities (see the "Tools for model execution" sidebar).

None of the implementations is perfect; each requires improvements and extensions. However, they do corroborate the feasibility of the ideas summarized below. In fact, since many of these ideas are standard practice in the world of conventional programming, the tools appear to be the first complex-system analogs of useful general-pur-
pose programming environments.

**Model execution.**

One of the most interesting notions to come out of recent work in systems engineering is that of **executable specifications** or, to fit in better with the terminology used here, **executable models**. Executing a model is analogous to running a program directly, with the aid of an interpreter. Unfortunately, the term has been erroneously equated with the animation of diagrams only. However, executability is in fact many-sided and far more significant.

A prerequisite to executing complex system models is the availability of a formal semantics for those models — most notably, for the medium that captures the behavioral view. Thus, while the adjective "visual" in the term "visual formalism" was justified earlier on grounds pertaining to model representation, the word "formalism" is justified now on grounds pertaining to model analogies.

The core of model execution is the ability to carry out a single step of the system's dynamic operation, with all consequences taken into account. During a step, the environment can generate external events, change the truth values of conditions, and update variables and other data elements. Such changes then affect the status of the system; they trigger state changes in the controllers, activate and deactivate activities, modify conditions and variables, and so on. In turn, each of these changes can cause many others, often yielding intricate chain reactions.

A semantics for the model must contain sufficient information to capture these ramifications precisely. Given the current status and the changes made by the environment, calculating the effect of a step usually involves complicated algorithmic procedures, which are derived from, and reflect, that semantics.

**Interactive and batch execution.** The simplest way to execute, or "run," the model using a computerized tool is in a step-by-step interactive fashion. At each step, the user emulates the system's environment by generating events and changing values. The tool, in turn, responds by transforming the system into the new resulting status. If the model is represented visually, the change in status will also be reflected visually, say, by changes in color or emphasis in the diagrams.

Once we have the basic ability to execute a step, our appetite grows. We might now want to see the model executing noninteractively. To check, for example, that a telephone call connects when it should, we can prepare the relevant sequence of events and signals in a batch file, set up the model to start in the initial status, and ask our tool to execute steps iteratively, reading in the changes from the file. The graphic feedback from such a batch execution becomes an (often quite appealing) animation of the diagrams.

By executing scenarios that reflect the way we expect our system to behave, we are able to verify that it will indeed do so — long before final implementation. If we find that the system's response is not as expected, we may go back to the model, change it, and run the same scenario again. This is analogous to single-step — or batch- debugging of conventional programs.

It should be emphasized that scenarios can be run at any time in the development effort, as long as the portion of interest is syntactically legal. During an execution, the user plays the role of all parts of the model that are external to the portion being executed, even if those parts will eventually be specified and thus become internal.

Again, from several years of seeing such execution capabilities used, mainly in large aerospace and electronic industries, I have become convinced of their value (as statistics are yet available to quantify this impression, although a couple of preliminary case studies have appeared). These execution capabilities appear to introduce an entirely new and powerful dimension into the task of verifying and debugging system models.

I have seen model executions uncovering hitherto unknown patterns of behavior, when the members of the development team thought they had covered everything. As a result, these people were able to discuss deep behavioral issues that would otherwise have been swept under a rug of enormous unreadable specification documents.

I have seen engineers use executability to tackle crucial problems, and, very early in the project, correct subtle conceptual errors — ones that could otherwise go undiscussed or undetected until it was too late. And typically, all these phenomena start to take place as soon as the first executions are run.

Customer representatives are often involved in these stages, which further supports what Brooks and others have urged: extensive prototyping and simulation of the system early on with the client.

**Programmed execution.** Our appetite now becomes even greater. We now might ask ourselves: If the tool can execute the model in detail, reading events in from a file, why should we be satisfied with merely witnessing the run in action and inspecting the final status? We would like to be able to incorporate breakpoints, causing the execution to suspend and the tool to take certain actions when particular situations come up. These actions can range from tempo-
rarily entering interactive mode (in order to monitor careful step-by-step progress) to executing a piece of ready-made code that describes a bottom-level activity.

In fact, we need not restrict ourselves to running self-devised scenarios. We might want to see the model executing under circumstances that we do not care to specify in detail. We might like to see its performance under random conditions and in both typical and less-than-typical situations. Such a capability gets to the heart of the need for an executable model: to minimize the unpredictable in the development of complex systems.

This more powerful notion of inspecting a model is achieved by the idea of programmed executions, using a special metalanguage supported by the tool. Programs in this language (which might be appropriately termed an execution control language) can be set up to look out for predefined breakpoints and accumulate information regarding the system's progress as it takes place.

As a simple example, in a typical flight of an aircraft we are specifying, we might want to know how many times the radar loses a locked-on target. Since it might be difficult for the engineer to put together a typical flight scenario, we can tap the power of our tool by instructing it to run many typical scenarios, using the accumulated results to calculate average-case information.

The tool follows typical scenarios by generating random numbers to select new events according to predefined probability distributions. The statistics are then gathered using appropriate breakpoints and simple arithmetical operations. The ideas behind these techniques are, of course, well known. However, the point is to extend them to conceptual models of complex systems, long before the costly final-implementation stages.

In a similar vein, we can use programmed executions to apply other, more powerful kinds of dynamic tests to system models. For example, we might set up an execution control program to carry out performance analysis. If we want to check whether an operating system we are modeling will ever require more main memory than some maximum allowed value, we can associate with the relevant activities in the functional view values that represent our knowledge about their memory consumption. We can then program the tool to run many typical scenarios, calculating the maximum memory consumption of all activities that are active simultaneously.

Despite its being applied to a system model, and not to a final implementation, this approach to analysis is far more informative than the extraction of worst-case estimates from simple graphs of process dependencies. The model being analyzed will (hopefully) be realistic and detailed, and executing it reflects precisely what would have happened had we run the real system instead.

If our analysis shows that the memory limit might indeed be exceeded, the tool can support that prediction by supplying the actual sequence of events that would cause it. Clearly, by replacing memory values with time information, similarly meaningful timing analysis can be carried out as well.

In general, then, carefully programmed executions can be used to inspect and debug the system model under a wide range of test data to emulate both the environment and the as-yet-unspecifed parts of the system and to analyze the model for performance and efficiency.

**Exhaustive executions.** When executing the model, we might detect such unpleasant anomalies as deadlocks or behavioral ambiguities (nondeterminism). However, finding and eliminating these in the cases that we happen to encounter does not ensure they will never occur in the lifetime of the system. It would be extremely useful to be able to run through all possible scenarios in search of such situations, by generating all possible external events and all changes in the values of conditions and variables.

We might also be interested in reachability tests, which would determine whether—when started in some given initial situation—the system can ever reach a situation in which some specified condition becomes true. This condition can be made to reflect desired or undesired situations. Moreover, we could imagine the test's being set up to report on the first scenario it finds that leads to the specified condition, or to report on all possible ones, producing the details of the scenarios themselves. We thus arrive at the idea of exhaustive executions.

Are such tests realistic? Could we subject the model to an exhaustive reachability test, for example, after which we will know for sure whether there is any possibility of its occurring under any possible circumstances? The answer, in principle, is yes, but with serious reservations. The number of possibilities that might have to be considered in an exhaustive execution can easily become incredibly large, even if we ensure that it is finite by limiting the possible values of the variables.

To get a feel for the sizes involved, a behavioral model that contains about 40 concurrent components, each with about 10 states, has more state configurations and, hence, might have more possible scenarios than the number of elementary particles in the entire universe. There can never be a language, method, or tool with which one can, in general, consider all of these in any reasonable amount of time.

This doesn't mean, however, that such tests are a bad idea.

First, the above numbers denote worst-case asymptotic estimates; a real system might very well have far fewer scenarios that can actually happen, and a careful process of considering only those that are feasible will take far less time than the worst-case estimate.

In fact, just such a reachability test was recently applied to a model of the firing mechanism of a certain, already deployed, ballistic missile system (The Statemate system was used for this. The main part of the underlying model consisted of a statechart with about 80 states. However, since these included parallel state components, the real number of states was much larger.) In less than three hours on a standard workstation, the test terminated, in the
process discovering a new sequence of events, unknown to the design team, that leads to the firing of the missile!

Second, exhaustive tests can be run on small, critical, and well-isolated parts of the model. We can instruct the tool to ignore some of the external events or to avoid simulating the details of certain activities. Clearly, this can cause it to overlook crucial situations, but the advantage is that the set of scenarios it considers is greatly reduced. To maximize the test’s effectiveness, such limiting constraints should be prepared very carefully, using as much knowledge of the modeled system as possible. This is another place where expert systems might come in handy.

Third, even if exhaustive tests cannot always be completed in reasonable amounts of time, it would be wise to have them run in the background, perhaps at night or on weekends, for as long as we can afford. There is nothing wrong with routinely submitting large system models to powerful supercomputers for exhaustive testing, even non-exhaustively. Since the tool can be set up to report on phenomena as they are discovered, the more time we observe such tests running without surprises the more confidence we have in the integrity of our model.

Watchdogs and temporal verification. Often, we are interested in establishing properties of the model that are of a global nature but are more involved than reachability or freedom from deadlock. Suppose we want to make sure that a certain party in a communication protocol never sends two consecutive messages unless a special item, say, an acknowledgment, is sent in the interim.

Although seemingly more complicated, this query can be cast in the form of a reachability test in the following way. First, construct a small special-purpose "piece" of behavioral specification that is carefully set up to enter a special state if and when the offending situation occurs. Next, attach this watchdog, as it is called, to the original model as a concurrent behavioral component and run a reachability test on the extended model to find out whether the special state can ever be entered. Since the watchdog runs in parallel with the rest of the model, the effect will be as desired.

Watchdogs can be used to verify the model against a wide variety of properties. Temporal logic, one of the most useful and well-known media for specifying global constraints on the behavior of a system, nicely complements modeling approaches that specify behavior in a more local, operational fashion. Under certain technical conditions, any temporal logic formula can be systematically translated into a watchdog, reducing the problem of verifying the complicated formula to that of establishing a much simpler property, such as reachability. The watchdog is then attached to the high-level controlling activity of the original model, resulting in a modified model, and an appropriate exhaustive test is run.

Actually, such verification need not be based solely on exhaustive executions. Research into the theory and technology of automatic verification of very large (but finite-state) systems against properties in temporal logic is already showing promising results. The techniques being developed in this area are far more subtle and efficient than brute-force exhaustive executions, and I believe that they will eventually find their way into system analysis tools. This direction of work might very well bring true system verification into the living room, so to speak.

Code generation. Even in its most advanced forms, executability is analogous to running conventional programs using interpretation. Complex systems are also amenable to the analog of compilation — that is, translating a model into runnable code in a lower level language. We call this ability code generation, although the term is often used to denote the more humble ability to re-
The code can be linked to "soft" panels — graphical mock-ups of control boards, complete with images of display screens, switches, dials, and gauges — that represent the actual user interface of the final system. These panels appear on the screen and can be manipulated with mouse and keyboard.

In the past few years, a number of companies have used this approach in design reviews involving customers and contractors, and it has proved to be extremely helpful — much more so than the typical documentation that accompanies such reviews.

It's important to point out that these system interface panels are not driven by hastily written code prepared especially for prototype purposes, but by code that was generated automatically from a model that is typically thoroughly tested and analyzed before being subjected to code generation. Moreover, when parts of the real target environment are available, they too can be linked to the code, and the runs become even more realistic.

Code generation is thus to be used for goals that go beyond the development team, in that code-driven mock-ups can be used as part of the standard communication between customer and contractor or subcontractor. It is not unreasonable that such a running version of the system model be a required deliverable in certain development stages.

A good code-generation facility would also have a debugging mechanism, with which the user can trace the executing parts of the code back up to the system model. Breakpoints can be inserted to stop the run when specified events occur, at which point the model's status can be examined and elements can be modified on-the-fly before resuming the run.

If substantial problems arise, changes can be made in the original model, which is then recomplied down into code and rerun. As in executions, trace files can be requested, recording crucial information for future inspection. Carrying the analogy between compilation and code generation a step further, this ability is tantamount to source-level debugging.

In addition to compiling, or codifying, the model itself, we can automatically produce code from specially prepared segments of behavior, such as watchdogs or test suites that are not part of the model but are used to execute and analyze it. For these, of course, the code generator output is actually final code.

An interesting variation calls for replacing high-level programming languages as the target medium for generated code by hardware description languages. A particular example is VHDL (which stands for VHSIC hardware description language, with VHSIC the abbreviation for very high-speed integrated circuit). In this way, hardware designers can also benefit from the virtues of the modeling and analysis techniques discussed above, and then translate their models into VHDL code, which can be subjected to silicon compilation or other appropriate procedures.

Verifying consistency between levels. Recall the process of preparing tiers, or strata, of conceptual models according to the physical model of the system. How can we establish the consistency of one level with the next?

There are a number of ways in which model-analysis techniques can help. The basic idea is to redirect the efforts from the task of inspecting and debugging a single model to the task of comparing two models. This applies to all manner of analysis and verification: interactive, programmed, and exhaustive execution; watchdogs and temporal verification; and code generation.

For example, we may execute the conceptual model prepared for a subsystem under the same conditions used to execute the original model of the entire system, and compare the results. One way to do this involves preparing scenarios for executing the new model directly from trace files of executions run on the original model. Clearly, this is not as simple as it sounds, and much research on this topic is still needed.

As far as code generation goes, we can often replace parts of the code generated from the original model by code generated from the newly designed subsystem models. If the final system is to be implemented in software, this has the effect of gradually bringing the original prototype code down toward a real implementation.

As subsystems are remodeled, their generated code is incorporated into the code that was generated one level higher. As more and more of the code becomes final production code, the entire system comes closer to its final form. It is not out of the question that this process will also become amenable to computerization. We can envision a user making restructuring decisions in the design stages (perhaps aided by an expert system) and the tool taking over from there, reorganizing the generated code in new, more efficient ways that reflect those decisions.

Combined with optimization procedures, which are badly needed and will hopefully be developed in the future, code generation has a chance to go far beyond prototyping, further justifying its role as the true complex system analog of conventional compilation.

The vanilla framework for system modeling outlined above is far from being universally accepted. Many of its facets are rooted in well-established and familiar ideas, but others are more recent and immature and require further work and experience.

On some issues, there is little agreement among researchers and practitioners, such as how to best approach the specification of behavior. I believe that the general framework is a good one and that there are also adequate proposals for behavioral specification. However, even the overall mold could easily turn out to be inadequate.

If it does not become the accepted analog of good old vanilla programming, then some other approach will. The precise form the winning effort takes on will be secondary, though I am fully convinced that reactive behavior will be one of its most crucial and delicate components, rigorous semantics included, and that visuality will play center stage.

From this basic framework, will evolve more specialized and exotic ones, for
which appropriate modeling languages will be designed and implemented and methods and guidelines conceived and mastered.

Things are far clearer in the analysis realm, where most of the abilities we have discussed are, to some extent, independent of the idiosyncrasies of the particular modeling approach. I believe that system development tools that lack powerful execution and code-generation capabilities will all but disappear.

Whenever people ultimately use to conceptualize and model their systems, the ability to thoroughly execute the resulting models and to compile them down into conventional high-level code will become indispensable. In a way, this too is vanilla. I believe that in time more exotic kinds of executability features will emerge, such as ones tailored to carry out timing and performance analysis, gather statistics, or compare the behavioral aspects of separate models.

A number of research directions present themselves, and in some there is already a promising body of work. Among the most important are

1. improving the techniques for generating high-quality code from conceptual models, and providing (semi)automated help to make design decisions in the process, and
2. enabling truly useful computerized verification of conceptual models against global constraints.

One of the crucial ingredients for success in these areas is extensive cooperation and collaboration of researchers in software and systems engineering with those in compilers, optimization, and heuristics for item (1) and in logic, semantics, and verification for item (2).

The current situation and the prospects for significant improvement indicate that we are at the start of a new and exciting era.

Acknowledgments

Discussions over the years with Amir Pnueli, Michal Politi, Rivi Sherman, and Moshe Cohen were extremely beneficial in helping me form the opinions voiced in this article. I am indebted to Derek J. Hatley, C.A.R. Hoare, Daniel Jackson, Ray Moritz, David L. Parnas, and the anonymous referees for commenting on a preliminary version of this material. This research was supported in part by grants from the Gutwirth Foundation and the Yeda Foundation for Applied Research.

References


---

Apply Yourself

with New Releases from Academic Press

Mathematica by Example
Martha Abell and James Braselton
January 1992, 672 pp., $32.50
ISBN: 0-12-041540-2

Fractal Attraction™
A Fractal Design System for the Macintosh®
Kevin D. Lee and Yosef Cohen
November 1991, $49.95
ISBN: 0-12-440740-4
Includes 80-page manual and one floppy disk.

Fundamentals of Spatial Information Systems
Robert Laurini and Derek Thompson
$49.95 (tentative)/ISBN: 0-12-438380-7

Fuzzy Systems Theory and Its Applications
Toshiro Terano, Kyozo Asai, and Michio Sugeno
$49.95 (tentative)/ISBN: 0-12-685245-6

Order from your local bookseller or directly from

ACADEMIC PRESS

CALL TOLL FREE
1-800-321-5068

Reader Service Number 1


---

IEEE CONFERENCE PREVIEW

1992 International Zurich Seminar on Digital Communications

March 16-19, 1992 — ETH Zurich, Switzerland

The Zurich Seminar is an event which has taken place every second year since 1970. It is organized by the IEEE Switzerland Chapter on Digital Communications. The focus in 1992 will be

Intelligent Networks and Their Applications

The area covered by IN (Intelligent Network) services will reach across all network types, such as terrestrial, satellite, cellular radio, Universal Personal Communications, and will include telephony, ISDN, video, etc.

Seminar: March 17-19, 1992, in a single session format. The main topics covered by the seminar will be: IN features and functionalities, IN architectures, IN modelling and evaluation systems, performance analysis, IN software environments. Tutorial: Monday March 16, 1992: "Object-oriented Design and Modelling of OSI and Distributed Processing Applications".

A full tutorial and seminar program with registration form is available from:

Secretariat IZS '92
P.O. Box
CH-8340 Hinwil
Switzerland

Tel: +41-1-938 15 56 (Mrs. Anne Schnicker)
Fax: +41-1-938 15 57
E-mail: izs@ks.id.ethz.ch

Readers can contact Harel at the Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. His e-mail address is harel@wisdom.weizmann.ac.il.
Panel

“No Silver Bullet” Reloaded – A Retrospective on “Essence and Accidents of Software Engineering”

Steven D. Fraser
Director (Engineering)
Cisco Research Center
Cisco Systems, San Jose

Frederick P. Brooks, Jr.
Kenan Professor
Dept of Computer Science
UNC, Chapel Hill

Martin Fowler
Chief Scientist,
ThoughtWorks
Boston

Ricardo Lopez
Principal Engineer
Qualcomm
San Diego

Aki Namioka
Development Manager
Cisco Systems
Seattle

Linda Northrop
Director
Product Line Systems
SEI (CMU), Pittsburgh

David Lorge Parnas
Director, Software Quality Research Lab
University of Limerick

David Thomas
Founder
Bedarra Labs
Ottawa

Abstract
Twenty years after the paper No Silver Bullet: Essence and Accidents of Software Engineering by Frederick P. Brooks first appeared in IEEE Computer in April 1987 (following its 1986 publication in Information Processing, ISBN 0444-7077-3) does the premise hold that the complexity of software is not accidental? How have the “hopes for silver” which included high-level language advances, object-oriented programming, artificial intelligence, expert systems, great designers, etc. – evolved? Panelists will discuss what has changed and/or stayed the same in the past twenty years – and the paper’s influence on the community.

Categories & Subject Descriptors:
D.2.9 Management
H.4 Information Technology and Systems
K.0 Computing Milieux
K.4.3 Organizational Impacts

General Terms: Design, Management

Keywords: Complexity, Silver Bullet, Design, Software

1. Steven Fraser (panel impresario), sdfraser@acm.org

STEVEN FRASER recently joined the Cisco Research Center in San Jose California as a Director (Engineering) with responsibilities for developing and managing university research collaborations. Previously, Steven was a member of Qualcomm’s Learning Center in San Diego, California with responsibilities for technical learning and development and creating the corporation’s internal technical conference – the QTech Forum. Steven also held a variety of technology management roles at Nortel/NT/BNR including: Process Architect, Senior Manager (Disruptive Technology and Global External Research), and Design Process Engineering Advisor. In 1994 he spent a year as a Visiting Scientist at the Software Engineering Institute (SEI) collaborating with the Application of Software Models project on the development of team-based domain analysis (software reuse) techniques. Fraser is the Corporate Support Chair for OOPSLA’07 and was the General Chair for XP2006. Fraser holds a doctorate in EE from McGill University in Montréal – and is a member of the ACM and a senior member of the IEEE. The IEEE Computer version of the paper “No Silver Bullet” memorably appeared the morning of his doctoral defense.

Fred Brooks’ seminal “No Silver Bullet” paper – often referenced simply by its initials “NSB” – begins with the chilling introduction:

“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...”

Panelists will discuss what “werewolves” remain, as systems grow in complexity and age – and what “hopes for silver” have evolved or have been laid to rest.

2. Frederick P. Brooks, Jr. brooks@cs.unc.edu

FREDERICK P. BROOKS, JR. received an A.B. summa cum laude in physics from Duke and a Ph.D. in computer science from Harvard. He joined IBM, working in Poughkeepsie and Yorktown, NY, 1956-1965. He was an architect of the Stretch and Harvest computers and later the project manager for the development of IBM’s System/360 family of computers. For this work he received a National Medal of Technology jointly with Bob O. Evans and Erich Bloch Brooks. With Dura Sweeney in 1957 – Brooks patented an interrupt system for the IBM Stretch computer that introduced most features of today’s interrupt systems. He coined the term computer architecture. His System/360 team first achieved strict compatibility, upward and downward, in a computer family. His early concern for word processing led to his selection of the 8-bit byte and the lowercase alphabet for the System/360, engineering of many new 8-bit input/output devices, and introduction of a character-string data type in the PL/I programming language.

In 1964 he founded the Computer Science Department at the University of North Carolina at Chapel Hill and chaired it for 20 years. Currently, he is Kenan Professor of Com-
puter Science. His principal research is in real-time, three-dimensional, computer graphics—"virtual reality." Brooks distilled the successes and failures of the development of Operating System/360 in The Mythical Man-Month: Essays in Software Engineering, (1975, 20th Anniversary Edition, 1995). He further examined software engineering in a 1986 paper, No Silver Bullet. He is a member of the National Academy of Engineering, the National Academy of Science, and the American Academy of Arts and Sciences. He has received the ACM A.M. Turing Award, the IEEE John von Neumann Medal, the IEEE Computer Society’s McDowell and Computer Pioneer Awards, the ACM Allen Newell and Distinguished Service Awards, the AFIPS Harry Goode Award, and an honorary Doctor of Technical Science from the Swiss Federal Institute of Technology (ETH-Zürich).

"No Silver Bullet" argues that:

1. The difficulties of making big software systems consist of essential difficulties and accidental (or incidental) difficulties.
2. The great leaps of progress in the past were accomplished by eliminating accidental difficulties, e.g., by high-level languages, time-sharing, and workstations.
3. Of the remaining difficulties, at least half seem to me to be essential, the very inherent complexity of what we build.
4. Therefore, no attack on accidental difficulties can bring an order-of-magnitude improvement—indeed, more than a factor of 2.
5. And yet, most of the proposed "radical improvements" proposed continue to address only accidental difficulties.

The soft spot in this argument is the estimate that at least half of our remaining trouble is essential. In the many challenges to the paper that have been published, none have challenged this point, even though it would seem easy to do so if one could enumerate a lot of particular accidental difficulties remaining.

The 1986 paper predicted that there would be "no order-of-magnitude improvement within a decade in software productivity, reliability, or simplicity." 1996 has come and gone. 2006 has come and gone. Few have stepped forward to claim such an improvement. In the 1995 Anniversary Edition of The Mythical Man-Month, Chapter 17 discusses progress in the first decade after "NSB". What has happened in the next decade?

Of the candidates enumerated in "NSB", object-oriented programming has made the biggest change, and it is a real attack on the inherent complexity itself. The most promising attack continues to be re-use, and re-use of COTS programs in particular. And we see much progress there; a 2004 paper by Boehm, Brown, Basili, and Turner describes the substantial difficulties remaining. It remains true that "The most radical possible solution for constructing software is not to construct it at all."

3. Martin Fowler, fowler@acm.org

Martin Fowler is an author, speaker, consultant, and general loud-mouth on software development. He concentrates on the issues of designing enterprise software, in particular: object-oriented technology, refactoring, patterns, agile methodologies, domain modeling, the Unified Modeling Language, and Extreme Programming. He has written five books and writes regularly at martinfowler.com. Martin was born in Walsall, England and lived in London for a decade before coming to New England where he has enjoyed living in the US even though he misses the beer, the deep pointlessness of Cricket, and the English countryside.

No Silver Bullet came out not long after I graduated. It was an extremely influential paper on me, in no small part due to the big influence it had on my early mentors. The term silver bullet has entered our industry's vocabulary, although in my experience it's never something directly claimed. It's common for people to criticize a technology on the basis that it isn't a silver bullet, but I can't remember anyone claiming their technology to be a silver bullet. (Amusingly tons of technologies claim an order of magnitude improvement, which is what this paper defines as a silver bullet.)

My greatest lessons from this paper were the importance of separating essential and accidental difficulties, and the central role of iterative development. Over the years the essential/accidental separation has become fuzzier for me - it's impossible to think of the essence of a problem without some form of representation, and in enterprise software the accidental difficulties increasingly appear as the essence of our problems. In contrast the strengths of iterative development have grown clearer and stronger. Within the agile movement we've seen huge developments in rapid turn-around of ideas into running tested code. I have no idea how to measure the consequences of iterative development on productivity, but I would place it as one of the great leaps of the last two decades.

A similarly large leap over this time foreshadowed by this paper is the presence of pre-built software, either in packages or libraries. These days nobody can propose a new language environment without access to a wide range of libraries. The open-source community has made an astonishing array of software available which I can use by just typing 'apt-get install'. When I compare what I'm working on now to what I had 20 years ago, I think I'd say that this software makes a greater difference to me than even the hardware that it drives.

If a message from this paper has been missed, it's the central importance of "great designers". It is something accepted in places like this, but when I think of our corporate clients I still see a huge gap between what they say and what they do. Our industry still has a long way to go here.
In the end core message behind the paper for me is that programming is fundamentally a hard task and that no single technology is going to leap up and make that difficulty go away. As a result we must distrust anyone who claims such a thing. I think technologists instinctively know this to be true, which is why silver bullet is only used as an accusation in our circles. Sadly many big purchasing decisions in the corporate world are not made by technologists - these days the werewolves are on the golf courses.

4. Ricardo Lopez, rlopez@qualcomm.com

RICARDO LOPEZ is a Principal Engineer in the Office of the Chief Scientist at QUALCOMM. He is responsible for software architecture, software process, and sometime Just Plain Old Software (JPOS). Architecting and designing software for over thirty years, he has been an evangelist for OO technology for the last twenty years and he has the arrow heads to prove it.

There is a “silver bullet” – it is the pursuit of personal and professional excellence – this when achieved, easily gives us an order of magnitude improvement in software productivity. There is no “silver bullet” from without – it must come from within.

5. Aki Namioka, anamioka@cisco.com

AKI NAMIOKA has been a Software Development Manager for more than 9 years. She is a member of Cisco Systems’ Voice Technology Group in Seattle. Prior to Cisco, Namioka held roles with IBM Global Services and Boeing’s Advanced Technology Center.

In Fred Brook's seminal article, he names several promising technologies that have become standard in current software development projects. We use code generation, third-party packages – notably open source projects, static analysis tools, etc. All of these have made our jobs easier.

However, the biggest problem I am faced with isn't fixing accidental errors or managing essential complexity - rather, it's creating products that meet expectations. Design by Defect is time consuming and wasteful. We rely on our Product Managers, Quality Assurance, and other members of the development team, to indirectly represent the interests of our real customers. However, no amount of requirements writing will create a thorough representation of our final output. There have been several suggestions presented at past OOPSLAs to help mitigate the lack of on-site customers in the development process, e.g. customer stand-ins, iterative design, etc. I can share some positive experiences we've had during this panel.

6. Linda Northrop, lnn@sei.cmu.edu

LINDA NORTHROP has more than 35 years of experience in software development as a practitioner, researcher, manager, consultant, and educator. She currently is director of the Product Line Systems Program at the SEI where she leads the work in software architecture, software product lines, and predictable component engineering. She recently led a yearlong study including leaders in the software community to define technical and social challenges to the creation of ultra-large-scale systems that will evolve in the next generation. The report, Ultra-Large-Scale Systems: The Software Challenge of the Future (ISBN 0-9786956-0-7), has just been published. She is coauthor of Software Product Lines: Practices and Patterns and chaired both the first and second international Software Product Line Conferences (SPLC1 and SPLC2). She is a past chair of the OOPSLA Steering Committee and OOPSLA 2001 conference Chair. Before joining the SEI, she was associated with both the United States Air Force Academy and the State University of New York as professor of computer science, and with both Eastman Kodak and IBM as a software engineer. As a private consultant, Linda also worked for an assortment of companies covering a wide range of software systems. She is a recipient of the Carnegie Science Award of Excellence for Information Technology and the New York State Chancellor’s Award for Excellence in Teaching.

When Brooks’ “No Silver Bullet” article was published I was mid-way through the college professor stage of my career. I immediately made it required reading for Lecture 3 of my undergraduate software engineering course. That lesson’s student outcomes included: awareness that software engineering involves more than programming; awareness of the software crisis; and familiarity with innovations, approaches, and movements introduced to combat the software crisis. Over the years, the article inspired scores of students, software continued to be in crisis state, and the article’s key phrases lodged in my memory. I find myself consistently drawing on those phrases and their underlying meaning in my post-professor career. In fact, as time progresses they have more meaning, not less. I have not seen a silver bullet.

Progress has been made in software engineering in the last twenty years and we have indeed constructed increasingly complex systems. In cases where we have focused on the essence, the results have been breathtaking. Yet conceptual errors still abound and crisis mode continues in far too many quarters. Innovations have too often focused on the accidents not the essence and in some cases have added greater complexity to software production.

At the same time, our global appetite for complex software and software-intensive systems continues to increase at a rate comparable to the increases in computational capacity of hardware. Sandwiched in the middle is our need to understand the problem domains and model solutions that harness the computational power as well as the attendant power of today’s sensor and wireless technologies. Current trends are leading us to systems of unthinkable scale in not only lines of code, but in the amount of data stored, accessed, manipulated, and refined; the number of connections and interdependencies; the number of hardware and computational elements, the number of system purposes and user perception of these purposes, the number of rou-
tine processes, interactions, and “emergent behaviors,” and the sheer number of people involved in some way. Brooks’ essential software difficulties of complexity, conformity, changeability, and invisibility are ever more poignant. In such systems the “grow not build” imperative is a fact of life, the boundary between developers and those we have called users blurs, and our accidental innovations help little. To wrestle these future werewolves we still need great designers and still have too few, but we also need to cultivate an interdisciplinary perspective that takes us uncomfortably out of our coding world.

7. **David Lorge Parnas, david.parnas@ul.ie**

David Lorge Parnas is Professor of Software Engineering, SFI Fellow, Director of the Software Quality Research Laboratory at the University of Limerick, Professor Emeritus at McMaster University and Adjunct Professor at Carleton University. Parnas received his B.S., M.S. and Ph.D. in Electrical Engineering – Systems and Communications Sciences from Carnegie Mellon University and honorary doctorates from the ETH in Zurich and the Catholic University of Louvain. He is a Fellow of the Royal Society of Canada and of the Association for Computing Machinery (ACM) and a Member of the Royal Irish Academy. He is licensed as a Professional Engineer in Ontario. Parnas won an ACM ‘Best Paper’ award in 1979, two ‘Most Influential Paper’ awards from the International Conference on Software Engineering, the 1998 ACM SIG-SOFT ‘Outstanding Research Award’, the ‘Practical Visionary Award’ given in honour of the late Dr. Harlan Mills, and the ‘Component and Object Technology’ award presented at TOOLS99. He was the first winner of the Norbert Wiener Prize from Computing Professionals for Social Responsibility and received the FifF prize from the Forum Informatiker für Frieden und Verantwortung in Germany. He is the author of more than 240 papers and reports. Many of his papers have been repeatedly republished and are considered classics. A collection of his papers can be found in Hoffman, D.M., Weiss, D.M. (eds.), “Software Fundamentals: Collected Papers by David L. Parnas”.

**Lead Bullets, Why are they not used?**

The attractive thing about the mythical silver-bullets is that they are easy to use. With a silver bullet in your arsenal, you need not work hard or hone skills, just point and whoosh - the enemy is defeated. Anyone who has the bullet can kill a werewolf, no education or training is required. Building software will never be like that. As Fred Brooks said, the essential difficulties will remain. However, I object to the phrase “accidental difficulties”. The word “accident” is often used in an attempt to escape blame. “It was an accident” is often a weak apology, a statement that the damage was unintentional and a claim that it could not be avoided. In fact, the accidents that I know about have all resulted from a combination of negligence, momentary carelessness, haste, greed, and poor training. This is also true of the many poor software products that I have examined. In each of them, the mistakes that led to the complexity and errors are obvious violations of principles and techniques that were known many years ago. These principles and techniques are not easy answers; they are hard work, but they are what we need to do.

It is a poor workman that blames his tools. In software, there are many who blame their programming languages and believe that new ones will be the silver bullet that so many seek. Programming languages are undoubtedly the great contribution of computer science to computer system development. The concept of (what were then called) higher level languages and the first implementations of that concept were incredibly useful. It is interesting that these useful tools were developed by people who had never studied about programming languages but knew what they were trying to do with them. Today, programming languages seem to be the province of people who have been taught all about them but know little else. Many bring to mind another old proverb, “to the man with a hammer, every problem is a nail”. If you examine the systems that consist of millions of lines of poor code, you will see that the problem is not “low level tools” but poor use of those tools. While I have no doubt that today’s languages can be improved (mostly by unification and simplification), they are not the cause of our problems and no new language will be the “silver bullet” that developers seek.

One of the stranger silver bullets that I see popularized should be called “business as usual”. For as long as I have observed software development, we have practised forms of iterative development, taken short-cuts when it comes to documentation and reviews, based our decisions on informal meetings, solved our problems by sitting next to others at a keyboard, and avoided “wasting time” by planning for the future. Where documentation was produced, it was seen as a required evil, and not taken seriously. Today I hear gurus advocating these old practices as if they were novel.

Another silver bullet has been the belief that we must depend on “great designers” to be “chief programmers” or “master architects” by means of great insight and creativity. If we were relying on such people to build our bridges, the ferry industry would be in great shape. There is a lot of routine work to be done and it is essential that it be done properly.

The belief that we can find a silver bullet is kept alive by another myth, the belief that we have made great progress in software engineering because we are managing to build increasingly complex systems. In fact, if there is progress it will because we will be building increasingly simple systems.

The belief that we have made progress is also encouraged by the impressive capabilities of the tools on our desktops. In fact, if you look closely you will see that these are made possible by highly improved hardware, not improved soft-
ware engineering. Many of the things we enjoy today (e.g. multiple-window displays) were invented and implemented in the 70s but only on very expensive specially developed hardware. If we tried to use today’s software on the standard hardware available then, we would be sharpening our pencils as we wait for a response.

The only solution to the never-ending software “crisis” is to try to emulate the science-based, disciplined, document based, development we see in good engineering projects. Our road builders (who are far from perfect) follow a fairly rigid process in which increasingly precise documents (often annotated drawings) are prepared, reviewed, analysed mathematically where needed, and then carefully used in the next phase of the process. Entry to the profession is also controlled; professional titles are restricted to people who have had a professional education and passed a series of exams. These restrictions are backed up by a regulatory regime in which those who are negligent or incompetent are examined and lose the right to practice. Introducing such professionalism in the software field is not easy, but it is the only road forward.

8. Dave Thomas, dave@bedarra.com

Dave Thomas (www.davethomas.net) is the founder of Bedarra Research Labs and is associated with Object Mentor, Carleton University and the Queensland University of Technology. Dave was a principle in the object conspiracy which sold OO to unsuspecting application developers who got lost in hopelessly complexity and technologists who used them to create hopelessly complex middleware frameworks. He helped create tools for those most addicted. Then he conspired with the Agile Alliance to try to convince companies that the best software is actually made by craftsmen whose work practices are not driven by a big process water wheel. Dave still labors under the illusion that it is possible for knowledgeable end users to create much of their own software if only our industry would enable and empower them.

Accidental Complexity – Footsteps in the Sand:

By many objective measures we have significantly increased rather than decreased the accidental complexity of software development.

- Languages, while supporting better abstraction mechanisms, have greatly increased complexity with generics, attributes, XML etc.
- Software professionals are increasingly being certified rather than apprenticed into true craftsmen.
- Class Libraries and Frameworks present an increasingly complex and constantly changing sea of APIs that must be mastered to create even the simplest of applications. There are too many ways to do the same thing.
- Tools are much more complex, often unnecessarily so.
- Competing fads and pseudo standards force market decisions to override sound technical decisions. Constant change is the enemy of quality.
- Open source frequently obligates the developer to understand an even larger code base and all too often maintain some of it.

Essential Complexity – We have seen the enemy and it is us!

- We ignore the KISS (Keep it Simple, Stupid) principle, ever introducing more concepts, frameworks and tools.
- We tackle too many domain intensive problems without the proper domain expertise.
- We jump to the new language, technology or release without considering the impact, often orphaning perfectly good assets because they are in the older version or language.
- We maintain a dedication to low level languages, albeit OO, when we know that their KLOCS will eventually kill us.
- We insist on taking our technology further from the end users who are increasingly more computationally literate than some of their developers.
- We are always in too much of a rush to build quality in and are therefore constantly trying to test defects out.
- We constantly break the rules of software physics only to relearn them down the road as the product or application goes into its 3rd release.

Lessons:

- Emphasize craftsmanship and mentor to really instill best practices. Good code really does need to be written at least 3 times! Writing compact high function code is a challenge for even the best developers.
- Written and verbal communication should be an essential part of education and professional development which is as valued as efficient code. We need Literate Programmers.
- In education we need emphasize skills such as discovery since the majority of development is enhancing or repairing an existing code base.
- The most important decisions for a requirement, architecture or design are “Is this really necessary?” and “Can we leave it out?”
- Exploit simpler languages with clear semantics for both the language and the libraries.
- Consider using a more powerful mechanism only where it will give you substantial leverage in terms of reduced code or greatly improved quality.