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 But manufactured things are infrequently changed after manufacture; they are superseded by later models, or essential changes are incorporated into later-serial-number copies of the same basic design.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 whims of the changers.

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

Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity.

In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.

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

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

Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction.

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

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

Past breakthroughs solved accidental difficulties

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

High-level languages. Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility.

What does a high-level language accomplish? It frees a program from much of its accidental complexity. An abstract program consists of conceptual constructs: operations, data types, sequences, and communication. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such. To the extent that the high-level language embodies the constructs one wants in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all.

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

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

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

brought by high-level languages.

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

Slow turnaround, like machine-language complexities, is an accidental rather than an essential difficulty of the software process. The limits of the potential contribution of time-sharing derive directly. 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.

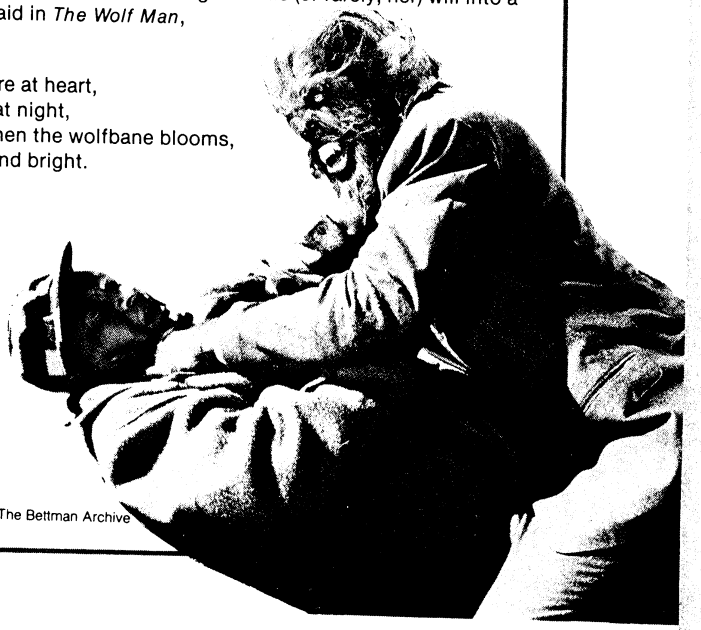
What should you watch out for? People whose eyebrows grow together, whose index finger is longer than the middle finger, and who have hair growing on their palms. Red or black teeth are a definite signal of possible trouble.

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



The Bettman Archive