There *Is* a Silver Bullet

A software industrial revolution based on reusable and interchangeable parts will alter the software universe

Brad J. Cox

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest. The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

—Brooks (see reference 1)

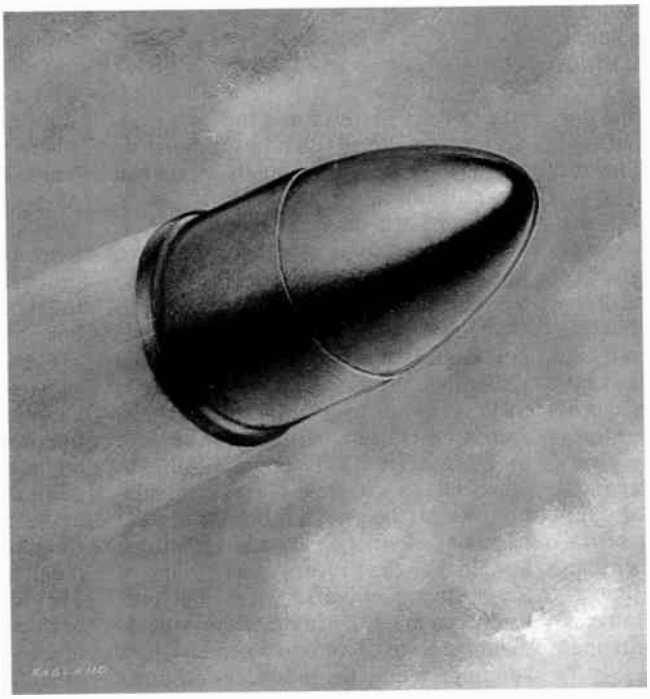


ILLUSTRATION: GREG RAGLAND © 1990

Two centuries after its birth in the industrial revolution, the age of manufacturing has matured and is showing signs of decline. And a new age, the information age, is emerging, born of the phenomenal achievements that the age of manufacturing brought to transportation, communication, and computing.

By eliminating time and space as barriers, however, the very achievements that put us within reach of a truly global economy are burying us in irrelevant and useless data, in mountains of low-quality ore that must be laboriously refined for relevant information—the signal hidden in the noise. The critical resource for turning this raw data into useful information is computer software, as strategic a resource in the information age as petroleum is today.

More than 20 years ago, the NATO Software Engineering Conference of 1968 coined the term *software crisis* to indicate that software was already scarce, expensive, of insufficient quality, hard to schedule, and nearly impossible to manage.

For example, in *The Mythical Man-month*, one of the seminal works of these two decades, Fred Brooks observed that adding more people to a late software project only makes matters worse. And in "No Silver Bullet: Essence and Accidents of Software Engineering," he argues that the difficulties are inevitable, arising from software's inescapable essence—not from accident, but from some deficiency in how programmers build software today.

But if you view these same facts from a new perspective, a more optimistic conclusion emerges. The software crisis is not an immovable obstacle but an irresistible force—a vast economic incentive that will grow toward infinity as the global economy moves into the information age.

To turn Brooks' own metaphor in a new direction, there *is* a silver bullet. It is a tremendously powerful weapon,

continued

propelled by vast economic forces that mere technical obstacles can resist only briefly. But as Brooks would agree, it is not a technology, a whiz-bang invention that will slay the software werewolf without effort on our part or vast side effects on our value systems and the balance of power between software producers and consumers.

The silver bullet is a *cultural* change rather than a technological change. It is a paradigm shift—a software industrial revolution based on reusable and interchangeable parts that will alter the software universe as surely as the industrial revolution changed manufacturing.

Object-Oriented Technologies

The term *object-oriented* keeps turning up. There are object-oriented environments, object-oriented applications, object-oriented databases, architectures, and user interfaces, and object-oriented specification, analysis, and design methods. And, of course, there are object-oriented programming languages, from conservative Ada to radical Smalltalk, with C++ and Objective-C somewhere in between. You may well wonder what, if anything, all these different technologies have in common. What does the adjective *object-oriented* really mean?

Who can say with certainty what *any* adjective means? No one is confused when an adjective such as *small* means entirely different things for cars, molecules, and galaxies. But in the software domain, words often cloud as much as they illuminate.

Object-oriented fails to distinguish between the low-level modularity/binding technologies of Ada and C++ and the higher-level ones of Smalltalk, and between these three languages and a hybrid environment like Objective-C. Moreover, purists summarily exclude ultra-high-level modularity/binding technologies like Fabrik or Metaphor (see reference 2) from the object-oriented domain because they are iconic rather than textual and because they do not support inheritance, forgetting that the same is true of things like tables and chairs that are indisputably "objects."

The confusion is understandable. The inhabitants of the software domain, from the simplest BASIC statement to a million-line application program, are as intangible as a ghost. And because programmers invent and build them all from first principles, everything in the software domain is unique and therefore unfamiliar, composed of modules and routines that have never been seen before and will never be seen again.

These component parts of the domain of software obey laws—promulgated by the programmers—that are specific to a unique instance. There are no general guidelines to bring order to the software domain. As Brooks put it, software is a world of werewolves and silver bullets. When all a programmer knows for certain is what he or she put there in the last few days, mystical belief will always win out over scientific reason. Thus, terms like *computer science* and *software engineering* remain oxymorons, tripping up those who don't recognize their inherent contradictions.

> ***Everything in the software domain is unique, composed of modules and routines that have never been seen before and will never be seen again.***

To get a grip on *object-oriented* means coming to the realization that it is an end, not a means—an objective rather than the technologies for achieving it. It means changing how we view software, shifting our emphasis to the objects we build rather than the processes we use to build them. It means using all available tools, from COBOL to Smalltalk and beyond, to make software as tangible—and as amenable to common-sense manipulation—as are the everyday objects in a department store. *Object-oriented* means abandoning the process-centric view of the software universe where the programmer-machine interaction is paramount in favor of a product-centered paradigm driven by the producer-consumer relationship.

But since reverting to this broader meaning might confuse the terminology even further, I use a separate term, *software industrial revolution*, to mean what *object-oriented* has always meant to me: transforming programming from a solitary cut-to-fit craft, like the cottage industries of colonial America, into an organizational enterprise like manufacturing is today. It means *enabling* software consumers, making it possible to solve your own specialized software problems the same way that homeowners solve plumbing problems: by assembling solutions from a robust market in off-the-shelf, reusable subcomponents, which are in turn supplied by multiple lower-level echelons of producers.

The problem with the old paradigm is illustrated with a simple question. When building a house, would you consider buying a plumbing system built entirely from unique, customized parts?

Yet this is just what the software community, with its infatuation with process and the pursuit of perfection from first principles, expects programmers to do whenever they build software. This would be considered ludicrous in a mature domain such as plumbing, yet it is business as usual in software.

To illustrate further, contrast the enormous interest that is generated by advances in process-oriented technologies—for example, structured programming, object-oriented programming, CASE, and Cleanroom—with the woeful lack of interest in the development of a robust market of fine-grained reusable software components.

The key element of the software industrial revolution is the creation of such a standard-parts marketplace, a place where those who specialize in the problem to be solved can purchase low-level, pluggable software components to assemble into higher-level solutions. The assemblers, of course, have as little interest in the processes used to build the reusable components as plumbers have in how to manufacture thermostats.

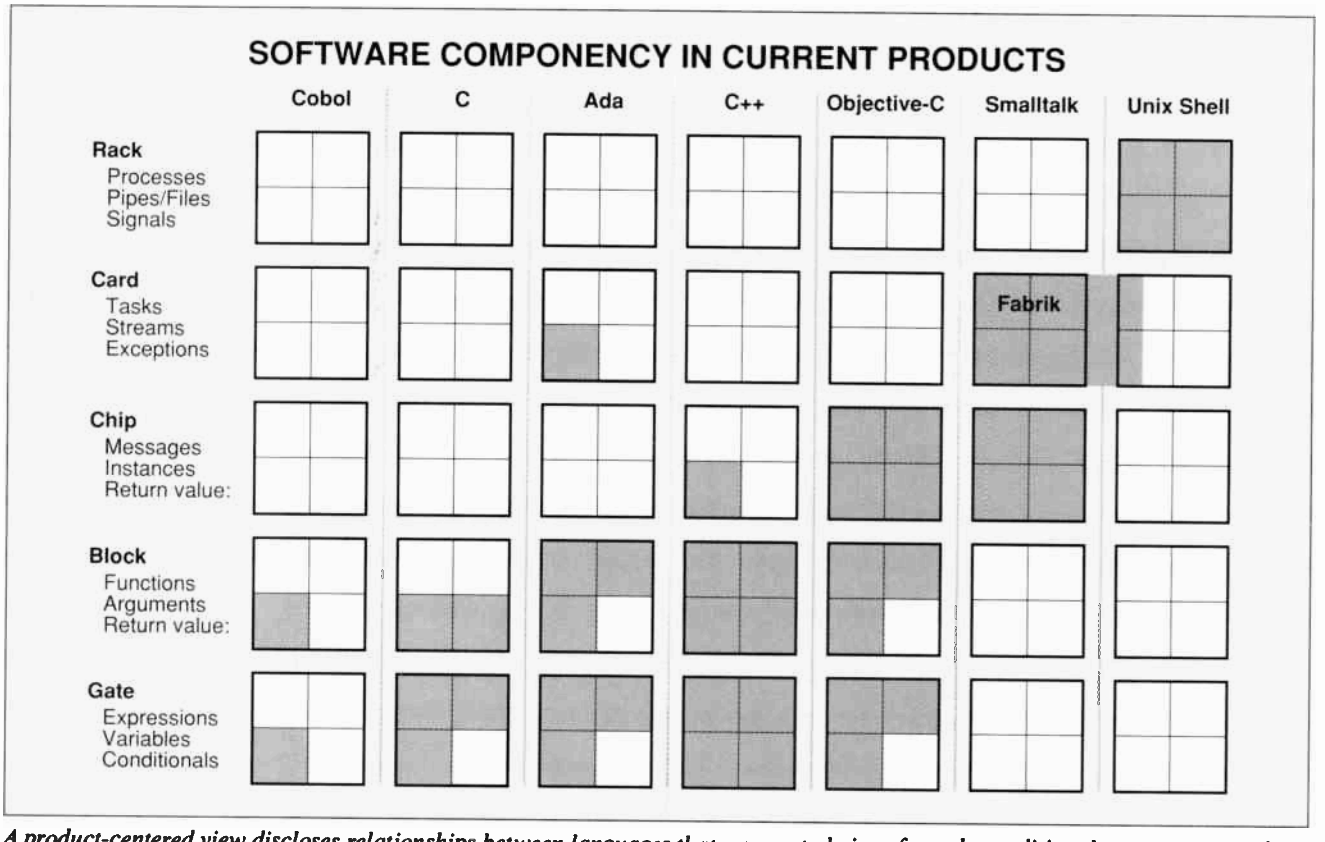
Clearly, software products are not the same as tangible products like plumbing supplies, and the differences are not minor. However, I shall avoid dwelling on the differences and emphasize a compelling similarity. Except for small programs that a solitary programmer builds for personal use, both programming and plumbing are organizational activities. That is, both are engaged in by ordinary people with the common sense to organize as producers and consumers of each other's products rather than reinvent everything from first principles. The goal of the software industrial revolution in general, and object-oriented technologies in particular, is to bring common sense to bear on software.

The Copernican Revolution

Let us assume that crises are a necessary precondition for the emergence of novel theories, and next ask how scientists

continued

THERE *IS* A SILVER BULLET



A product-centered view discloses relationships between languages that were not obvious from the traditional process-centered view. It implies a multilayered architecture of reusable, interchangeable software components analogous to the multilayered architecture of hardware engineering. It also implies that the critical path to escaping the software crisis does not involve discovering new modularity/binding technologies, but integrating those that are already known.

respond to their existence. Part of the answer, as obvious as it is important, can be discovered by noting first what scientists never do when confronted by even severe and prolonged anomalies. Though they may begin to lose faith and then to consider alternatives, they do not renounce the paradigm that has led them into crisis. They do not, that is, treat anomalies as counter-instances, though in the vocabulary of philosophy of science, that is what they are. The decision to reject one paradigm is always simultaneously the decision to accept another, and the judgment leading to that decision involves the comparison of both paradigms with nature and with each other.

—Kuhn (see reference 3)

Aristotle's universe had the earth and mankind at the center, with the sun, moon, planets, and stars circling around on ethereal spheres. The second-century astronomer Ptolemy amended this model by adding *epicycles* to account for observed discrepancies in planetary motion. By the sixteenth century, 90 such epicycles were needed, and the resulting complexity created an astronomy crisis.

The Aristotelian cosmological model as extended by Ptolemy was once as entrenched and "obvious" as today's process-centered model of software development. Given any particular discrepancy, astronomers were invariably able to eliminate it by making some adjustment in Ptolemy's system of epicycles, just as programmers can usually overcome specific difficulties within today's software-development paradigm.

But like today's software engineers, the astronomers could never quite make Ptolemy's system conform to the best observations of planetary position and precession of the equinoxes. As increasingly precise observations poured in, it became apparent that astronomy's complexity was increasing more rapidly than its accuracy and that a discrepancy corrected in one place was likely to show up in another. The problem could not be confined to the astronomers and ignored by everyone else, because the Julian calendar, based on the Ptolemaic model, was several days wrong—a discrepancy that any believer could see from the behavior of the moon.

This was as serious a problem for that era as the software crisis is today, since missing a saint's day lessened a worshipper's chances of salvation. By the sixteenth century, the sense of crisis had developed to the point that you can well imagine an early astronomer, frustrated beyond limit with keeping Ptolemaic models up to date, venting his despair in an article titled "No Silver Bullet: Essence and Accidents of Astrophysics."

In 1514, the Pope asked Copernicus to look into calendar reform, and over the next half century, Copernicus, Galileo, Kepler, and others obliged by eliminating the astronomy crisis, once and for all. But their silver bullet was not what the church had in mind. It was not a new process, some whiz-bang computer or programming language for computing epicycles more efficiently. It was a cultural change, a paradigm shift, a "mere" shift in viewpoint as to whether the heavens rotate around the earth or the sun.

The consequences to all the beliefs, value systems, vested interests, and

continued

power balances of that era were anything but minor. The astronomer's silver bullet removed mankind from its accustomed place at the center of the universe and placed us instead at the periphery, mere inhabitants of one of many planets circling around the sun.

The software industrial revolution involves a similar paradigm shift, with a similar assault on entrenched value systems, power structures, and sacred beliefs about the role of programmers in relation to consumers. It is also motivated by practical needs that an older paradigm has been unable to meet, resulting in a desperate feeling of crisis.

Just as the church's need for calendar reform escalated the astronomy crisis to where change became inevitable, the need for reliable software in the information age is escalating the software crisis to where a similar paradigm shift is no longer a question of whether, but of when and by whom.

The Industrial Revolution

It does not diminish the work of Whitney, Lee, and Hall to note the relentless support that came from the government, notably from Colonel Wadsworth and Colonel Bomford in the Ordnance Department and from John C. Calhoun in Congress. The development of the American system of interchangeable parts manufacture must be understood above all as the result of a decision by the United States War Department to have this kind of small arms whatever the cost.
—Hawke (see reference 4)

The cottage-industry approach to gunsmithing was in harmony with the realities of colonial America. It made sense to all parties, producers and consumers alike, to expend cheap labor as long as steel was imported at great cost from Europe. But as industrialization drove materials costs down and demand exceeded what the gunsmiths could produce, they began to experience pressure for change.

The same inexorable pressure is happening in software as the cost of computer hardware plummets and the demand for software exceeds our ability to supply it. As irresistible force meets immovable object, you experience the pressure as the software crisis: the awareness that software is too costly and of insufficient quality, and its development is nearly impossible to manage.

The software industrial revolution will occur, sometime, somewhere, whether programmers want it to or not, because it will be the software consumers who determine the outcome. It is only a question of when and by whom—whether the present software development community will be able to change its value system quickly enough to service the relentless pressure for change.

Contrary to what a casual understanding of the industrial revolution may suggest, it didn't happen overnight, and it didn't happen easily. In particular, the revolutionaries were not the cottage-industry gunsmiths; they actually seem to have played no role whatsoever, for or against. They stayed busy in their workshops and left it to their consumers to find another way.

The software industrial revolution will occur, sometime, somewhere, whether programmers want it to or not, because the consumers determine the outcome.

Judging from a letter written by Thomas Jefferson in 1785, it was actually he who found the solution in the workshop of a French inventor, Honoré Blanc. Key technical contributions were made by entrepreneurs (e.g., Eli Whitney, John Hall, and Roswell Lee) attracted from outside the traditional gunsmith community by incentives that the gunsmiths' consumers laid down to foster a new approach.

Those with the most to gain, and nothing to lose—the consumers—took control of their destiny by decisively wielding the behavioral modification tool of antiquity: money. They created an economic incentive for those vendors who would serve their interest (i.e., the ability to reuse, interchange, and repair parts) instead of the cottage-industry gunsmiths' interest in fine cut-to-fit craftsmanship.

Although it took consumers nearly 50 years to make their dream a reality, they moved the center of the manufacturing universe from the process to the product, with the consumers at the center and the producers circling around the periphery. Of course, the gunsmiths were "right" that interchangeable parts were far more expensive than cut-to-fit parts. But high-precision interchangeable parts ultimately proved to be the silver bullet for the manufacturing crisis: the paradigm shift that launched the age of manufacturing.

Reusable Software Components

A crucial test of a good paradigm is its ability to reveal simplifying structure to what previously seemed chaotic. Certainly, the software universe is chaotic today, with object-oriented technologies fighting traditional technologies, Ada fighting Smalltalk, C++ fighting Objective-C, and rapid prototyping fighting traditional methods, such as Milspec 2167 and Cleanroom.

Only one process must win and be adopted across an entire community, and each new contender must slug it out with older ones for the coveted title, "standard." Different levels of the producer-consumer hierarchy cannot seek specialized tools and reusable components for their specialized tasks, skills, and interests, but must fit themselves to the latest do-it-all panacea.

By focusing on the product rather than the process, a simpler pattern emerges, reminiscent of the distinct integration levels of hardware engineering (see the figure on page 212). On the card level, you can plug off-the-shelf cards to build custom hardware solutions without having to understand soldering irons and silicon chips. On the chip level, vendors can build cards from off-the-shelf chips without needing to understand the minute gate- and block-level details that *their* vendors must know to build silicon chips. Each modularity/binding technology encapsulates a level of complexity so that its consumer needn't know or care how components from a lower level were implemented, but only how to use them to solve the problem at hand.

Building software applications (rack-level modules) solely with tightly coupled technologies like subroutine libraries (block-level modules) is logically equivalent to wafer-scale integration, which is something that hardware engineering can barely accomplish to this day. Yet this is just what every software developer must do.

So, seven years ago, I cofounded The Stepstone Corp. to play a role analogous to that of silicon chip vendors by providing chip-level software components, or Software-ICs, to the system-building community. The goal was to create an enabling technology for a multilevel

continued

THERE *IS* A SILVER BULLET

marketplace in reusable software components. (It is called the Objective-C System-building Environment.)

Stepstone's experience amounts to a large-scale study of the reusable-software-components-marketplace strategy in action. With substantial Software-IC libraries now in the field and others on the way, the chip-level software-components-marketplace concept has been tried commercially and proven sound for an amazingly diverse range of customer applications.

However, this study has also revealed how difficult it still is, even with state-of-the-art object-oriented technologies, to design and build components that are both useful and genuinely reusable, to document their interfaces so that consumers can understand them, to port them to an unceasing torrent of new hardware platforms, to ensure that recent enhancements or ports haven't violated some preexisting interface, and to market them to a culture whose value system, like that of the colonial gunsmith, encourages building everything from first principles to avoid relying on somebody else's work.

A particularly discouraging example of this value system is that, in spite of the time and money invested in libraries and environmental tools like browsers, Objective-C continues to be thought of as a language to be compared with Ada and C++, rather than as the tiniest part of a much larger environment of ready-to-use software components and tools.

Another lesson of the last few years is that chip-level objects are only a beginning, not an end. The transition from how the machine forces programmers to think to how everyone expects tangible objects to behave is not a single step, but many, as shown in the figure. Just as there is no universal meaning for adjectives like *small* or *fast*, there is no single, narrow meaning for *object-oriented*.

At the gate and block levels of the figure, *object-oriented* means encapsulation of data and little more. The dynamism of everyday objects has been relinquished in favor of machine-oriented virtues, such as computational efficiency and static type checking. At the intermediate (chip) level, objects also embrace the open-universe model of everyday experience where all possible interactions between the parts and the whole are not declared in advance, as opposed to the closed universe of early binding and compile-time type checking.

But on the scale of any large system, gate-, block-, and even chip-level objects are extremely small units of granularity: They are grains of sand where bricks are needed. Since even chip-level objects are as procedural as conventional expressions and subroutines, they are just as alien to nonprogrammers. Until invoked by passing them a thread of control, these objects are as inert as conventional data, quite unlike the objects of everyday experience.

What Next?

In the stampede to force the world's round, dynamic objects into square, static languages like Ada or C++, you must never forget that such low-level languages—and even higher-level environments like Objective-C and Smalltalk—are highly unlikely to be accepted by mainstream information-age workers. They are more likely to insist on nontextual, nonprocedural visual "languages"

continued

THERE *IS* A SILVER BULLET

that offer a higher-level kind of "object," a card-level object, of the sort that programmers know as coroutines, lightweight processes, or data-flow modules. Since these objects encapsulate a thread of control alongside whatever lower-level objects were used to build them, they admit a tangible user interface that is uniquely intuitive for nonprogrammers.

By definition, these systems are more fundamentally "object-oriented" than the procedural, single-threaded "object-oriented" languages of today. Like the tangible objects of everyday experience, card-level objects provide their own thread of control internally. They don't "communicate by messages"; they don't "support inheritance"; and their user interface is iconic, not textual.

By introducing these and probably many other architectural levels, where the modularity/binding technologies at each level are oriented to the skills and interests of a distinct constituency of the reusable-software-components market, the programmer shortage can be solved as the telephone-operator shortage was solved, by making every computer user a programmer. ■

REFERENCES

1. Brooks, Fred. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, April 1987.
2. Ingals, Dan, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. "Fabrik: A Visual Programming Environment." *OOPSLA '88 Proceedings*. Metaphor is an office automation product of Metaphor Computer Systems (Mountain View, CA).
3. Kuhn, Thomas. *The Structure of Scientific Revolutions*. The University of Chicago press, 1962.
4. Hawke, David Freeman. *Nuts and Bolts from the Past: A History of American Technology 1776–1860*. New York: Harper and Row, 1988.

ACKNOWLEDGMENT

This article draws upon a longer, more detailed piece I wrote: "Planning the Software Industrial Revolution: The Impact of Object-Oriented Technologies," to be published in IEEE Software, *November 1990.*

Brad J. Cox is a cofounder and chief technical officer of The Stepstone Corp. (Sandy Hook, CT). He is also the author of Object-Oriented Programming: An Evolutionary Approach, *and the originator of the Objective-C System-building Environment. He can be reached on BIX c/o "editors."*