What if there's a Silver Bullet...

And the Competition Gets it First?

by Brad 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 [1].

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 Age of Information, is emerging, born of the phenomenal achievements that the Age of Manufacturing brought to transportation, communication, and computing.

However, by eliminating time and space as barriers, the very achievements that put us within reach of a truly global economy are burying us in irrelevant and useless data; mountains of low quality ore that must be 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, which is becoming the limiting strategic resource of the Age of Information much as petroleum is a strategic resource today.

As early as two decades 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, even with the techniques that made the Age of Manufacturing so successful. 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 and not from accident; some deficiency in how programmers build software today.

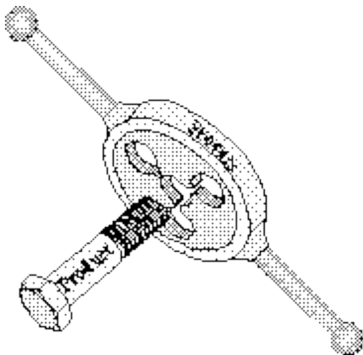

Figure 1: Whereas mature engineering domains define standard products such as this bolt and expect diverse processes to be used in making them, in software we still do the reverse, defining standard languages and methodologies rather than standard components to be implemented in many ways.

This article takes the opposite viewpoint. The software crisis is not an immovable obstacle but an infinite force, a vast economic incentive that will grow towards infinity as the global economy moves into the Age of Information. To turn Brooks' own metaphor in a new direction, there *is* a silver bullet. It is a tremendously powerful weapon, 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, nor vast side-effects on our value systems and power balances. The silver bullet is a *cultural* change rather than a technological change. It is a paradigm shift; a software industrial revolution that will change

the software universe as the industrial revolution changed manufacturing.

Object Technology

Object-oriented is turning up all over! There are object-oriented environments, object-oriented applications, object-oriented databases, object-oriented drawing programs, object-oriented architectures, object-oriented telephone switching systems, object-oriented user interfaces, and object-oriented specification, analysis, and design methods. And oh yes, there are object-oriented programming languages. Lots of them, of every description, from Ada at the conservative right to Smalltalk at the radical left, with C++ and Objective-C[TM] somewhere in between. And everyone is asking "What could such different technologies possibly have in common? Do they have *anything* in common? What does `object-oriented' really *mean?*"

What does *any* adjective mean in this murky swamp called software? No one is confused when adjectives like small or fast mean entirely different things in nuclear physics, gardening, and geology. But in the abstract murk of the software domain it is all too easy to lose one's bearings, to misunderstand the context, to confuse the very small with the extremely large. So the low-level modularity/binding technologies of Ada and C++ are confused with the higher-level ones of Smalltalk, and all three with hybrid environments like Objective-C. And ultra-high-level object-oriented technologies like Fabrik and Metaphor[2] are summarily excluded for being iconic rather than textual and for not supporting inheritance, forgetting that the same is true of tangible everyday entities that are indisputably `objects'.

Such confusion is not surprising. The denizens of the software domain, from the tiniest expression to the largest application, are as intangible as any ghost. And since programmers invent and build them all from first principles, everything we encounter there is unique and unfamiliar, composed of sub-components that have never been seen before and will never be seen again, and that obey laws unique to that encounter that don't generalize to future encounters. Software is a domain where dreams are planted and nightmares harvested, an irrational, mystical swamp where terrible demons compete with magical panaceas, or as Brooks put it, a world of werewolves and silver bullets. So long as all that anyone can know for certain is whatever we ourselves wrote during the last week or so, mystical belief will reign supreme over scientific reason. Terms like `computer science' and `software engineering' will remain oxymorons; at best content-free twaddle spawned of wishful thinking and at worst a cruel and selfish fraud on the consumers who pay our salaries.

In the broadest sense, `object-oriented' refers to the war and not the weapons, the ends and not the means, an objective rather than technologies for achieving it. It means orienting on the *objects* rather than on processes for building them; wielding all the tools programmers can muster, from well-proven antiques like Cobol to as-yet-missing ones like specification/testing languages, to *enable* software consumers, letting them reason about software products with the common-sense skills we all use to understand the tangible objects of everyday experience. It means relinquishing the traditional *process*-centered paradigm with the programmer-machine relationship at the center of the software universe (Figure 1) in favor of a *product*-centered paradigm with the producer-consumer relationship at the center.

But since reverting to this broader meaning might confuse terminology even further, I will 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. This means *enabling* software consumers; making it possible for them to solve their *own* specialized software problems as homeowners solve plumbing problems, by assembling their own solutions from a robust commercial marketplace in off-the-shelf sub-components, which are in turn supplied by multiple lower-level echelons of producers.

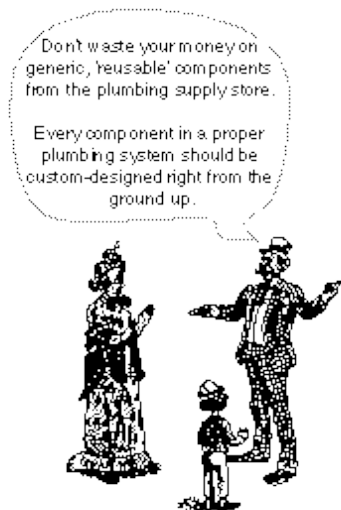

Figure 2: Mrs. Kahootie's Plumber: Imagine the complexity, conformity, and changeability problems he'd be getting her into, should she lack the common sense to stop him.

The problem with the old paradigm is shown in Figure 2. Although Mrs. Kahootie would be best served with standard products from a plumbing supply marketplace, her plumber shares the software community's infatuation with *process,* the pursuit of perfection from first principles. Although his proposal would seem ludicrous in a mature domain such as plumbing, it is the norm in software. For example, observe the tremendous interest in each new advance in software development process - structured programming, object-oriented programming, CASE, Cleanroom, Milspec 2167, Ada, and C++, and the lack of interest in, or disdain for, the prospect of robust marketplaces in fine-granularity software components like the plumbing supply marketplace. The key element of the software industrial revolution is the creation of such a marketplace, a place where low-level software components can be purchased for subsequent assembly into higher-level solutions by those whose specialty is the problem to be solved, but with as little interest in the processes used to build them as plumbers have in how water pumps and thermostats are manufactured.

Clearly, software products are not the same as tangible products like plumbing supplies, and the differences are not small. However I shall avoid dwelling on the differences to 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 *people* like those who raised the pyramids, fly to the moon, build computer hardware, and repair their plumbing; ordinary people with the common sense to organize as producers and consumers of each other's products rather than reinventing everything from first principles. The goal of the software industrial revolution as the war, and object-oriented technologies as a weapon, is bringing common sense to bear on software.

Copernican Revolution

Let us then assume that crises are a necessary precondition for the emergence of novel theories and ask next how scientists 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 judgement leading to that decision involves the comparison of both paradigms with nature and with each other3.

The Aristotelian cosmological model as extended by Ptolemy (Figure 3) 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 the software development paradigm of today.

But 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 could be seen by any believer 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 we could 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*.

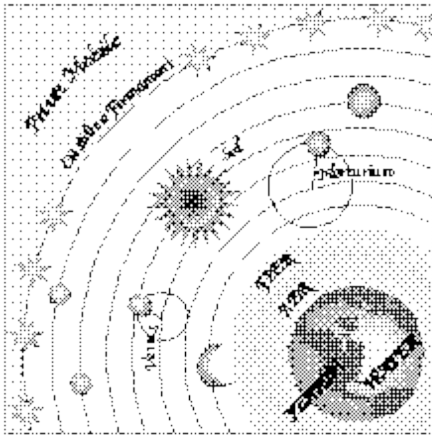

Figure 3: The Aristotelian model of the universe, as amended by Ptolemy, had the heavens circling the earth on crystaline spheres, with obsomalies in planetary motion explained by postulating secondary spheres, called 'epicycles'.

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 quite what the Pope 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 on the beliefs, value systems, vested interests and power balances of that era were anything but minor. Their silver bullet removed mankind from his accustomed place at the center of the universe at the focus of God's vision and the primary object of his concern, 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 our 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 software in the Age of Information is escalating the software crisis to where a similar paradigm shift is no longer a question of whether, but of how quickly and by whom.

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."4

The gunsmith shop in Colonial Williamsburg, Virginia, is a fascinating place to watch gunsmiths build guns as programmers build software, fabricating each part from raw materials and hand-fitting each part to each assembly. When I was last there, the gunsmith was filing a beautifully proportioned wood screw from a wrought iron rod that he'd forged on the anvil behind his shop, cutting its threads by hand and measuring entirely by eye. I was fascinated by his demonstration of testing a newly forged gun barrel, charging it with four times the normal load, strapping it to a log, and letting `er rip from behind a sturdy shelter, not the least hindered by many programmers' paralyzing obsession that such testing `only' reveals the presence of defects, not their absence.

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 demand for software exceeds our ability to supply it. As irresistible force meets immovable object, we experience the pressure as the software crisis; the awareness that software is too costly and of insufficient quality, and its development nearly impossible to manage.

Insofar as this pressure is truly inexorable, nothing we think or do can stand in its path. The software industrial revolution *will* occur, sometime, somewhere, whether programmers want it to or not, because it is our consumers who determine the outcome. It is not a question of whether, but only of how quickly, and of whether the present software development community will be the ones who will service the inexorable 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, who actually played almost no role whatsoever, for or against. The value system of the craftsman culture, so palpable in that Williamsburg gunsmith, was too strong to overcome from within. They stayed busy in their workshops, filing on their iron bars. It was their *consumers* who found a better way.

Judging from a letter by Thomas Jefferson in 1785, it was actually the nation's president-to-be, the ultimate consumer of ordnance products, who found the solution in the workshop of a French inventor, Honoré Blanc. Key technical contributions were made by entrepreneurs like Ely Whitney, John Hall, and Roswell Lee, attracted from outside the traditional gunsmith community by incentives that the gunsmith's consumers laid down to foster a new approach.

The real heroes of this revolution were those with the most to gain, and nothing to loose, the *consumers*. They took control of their destiny by decisively wielding the behavior modification tool of antiquity, *money*. They created an economic incentive for a new group of producers prepared to service the *consumers*' interest in interchangeability and repairability, rather than the cottage industry gunsmiths' interest in fine cut-to-fit craftsmanship.

Although it took half a century for the consumers' dream to become a reality, they moved the center of the manufacturing universe from the *process* to the *product*, with the consumers at the center with the producers circling around the periphery. Of course, the gunsmiths' were `right' that interchangeable parts were initially 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. 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 for their specialized tasks, skills, and interests, but must fit themselves to the latest does-all panacea.

By focusing on the *product* rather than on the *process*, a simpler pattern emerges, reminiscent of the distinct integration levels of hardware engineering (Figure 4). Card-level pluggability lets end-users plug off-the-shelf cards to build custom hardware solutions without having to understand soldering irons and silicon chips. Chip-level pluggability lets vendors 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 the 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.

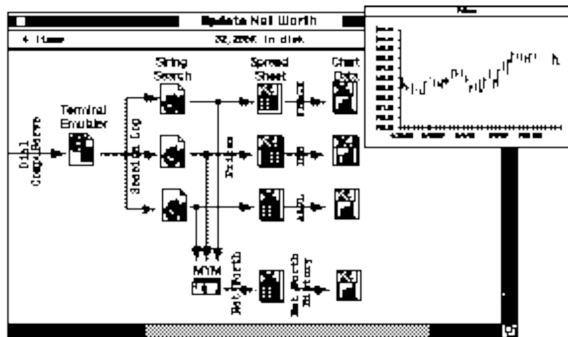

Figure 4: Graphical example of programming with "card-level" objects

Building applications (rack-level modules) solely with tightly-coupled technologies like subroutine libraries (block-level modules) is logically equivalent to wafer-scale integration, something that hardware engineering can barely accomplish to this day. So seven years ago, Stepstone began to play a role analogous to the silicon chip vendors, providing chip-level software components, or Software-ICs[TM], to the system-building community. But since C did not support pluggability - the ability to bind a pre-existing component into its new environment *dynamically*, when the component is used, rather than *statically*, when it is produced, we had to first extend C by adding a pre-processor that could provide the requisite loosely coupled chip-level modularity/binding technology of Smalltalk within C's gate and block-level technologies. The goal of this extended language, Objective-C[TM], was not to be yet another programming language, and as originally envisioned, not even to repair C's long-standing deficiencies[5]. It was to be the enabling technology for a multi-level marketplace in software components; the Objective-C System-building Environment.

Our experience amounts to a large-scale studyof the 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 we have also discovered 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 customers 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 pre-existing interface, and to market them to a culture whose value system, like the Williamsburg 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 we've invested in libraries and environmental tools such as browsers, Objective-C continues to be thought of as a gate- and block-level *language* to be compared with Ada and C++ rather than as the tiniest part of a much larger *environment* of ready-to-use chip-level software components and tools.

We have also learned that chip-level objects are only a beginning, not the 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 Figure 4. Just as there is no universal meaning for adjectives like small or fast, there is no single narrow meaning for object-oriented other than summarily dictated ones. At the gate and block levels of Figure 4, 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 level, chip-level objects also embrace the open universe model of everyday experience where all possible interactions between parts and the whole are not declared in advance, when the universe is created by the compiler.

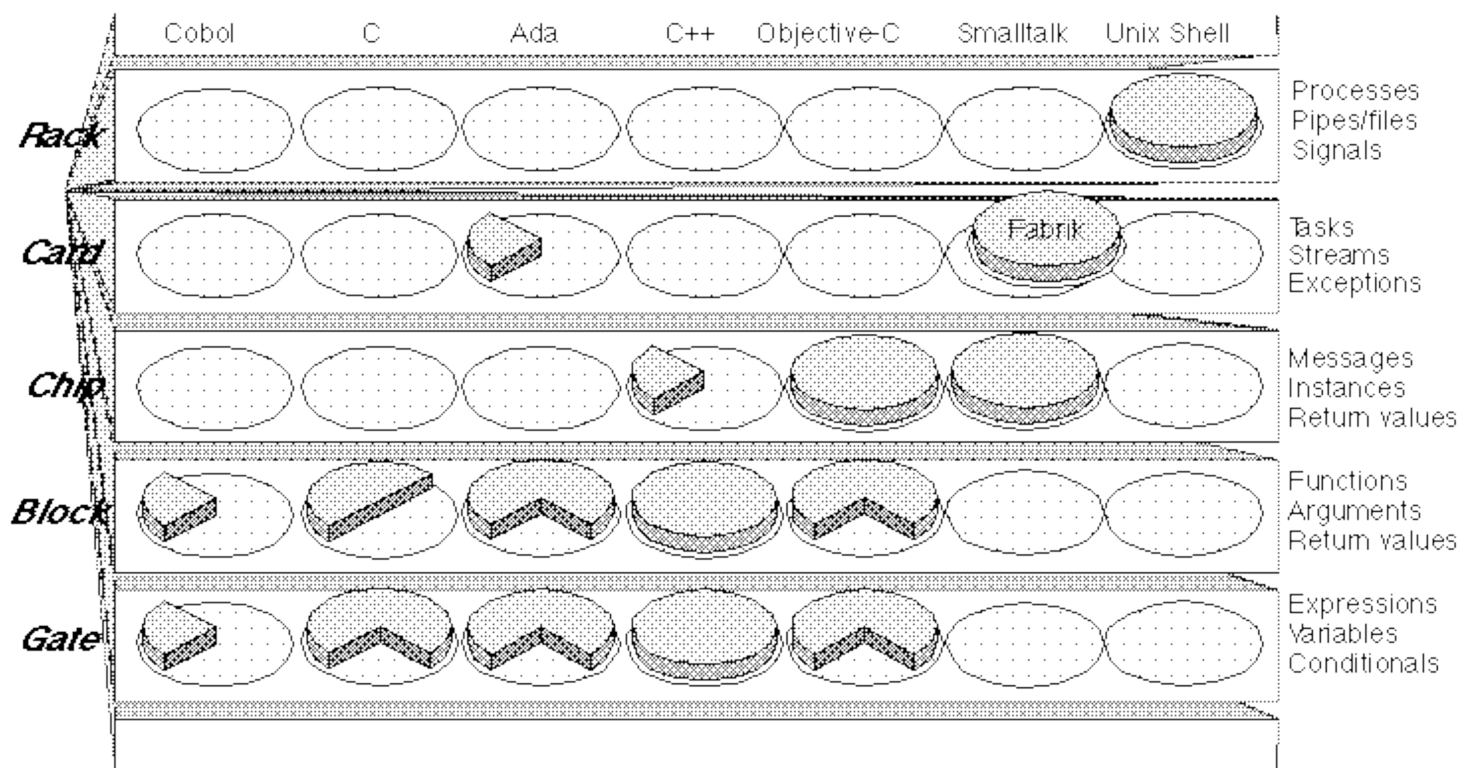

Figure 5: A multi-level programming environment offers diverse programming metaphors for the skills, interests and abilities of diverse constituencies of any large marketplace. In the foreground window an application specialist uses a visual programming `language' to build an application (rack-level module) by connecting lightweight tasks (card-level modules) and conventional objects (chip-level modules). The chip- and card-level objects were built in a traditional textual object-oriented environment such as Objective-C, typically by assembling even lower-level objects (gate- and block-level modules) that were written in even lower-level languages such as Ada or C++.

But on the scale of any large-scale system, gate, block, and even chip-level objects are extremely small units of granularity; 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 non-programmers. Until invoked by passing them a thread of control from outside, they are as inert as conventional data, quite unlike the objects of everyday experience.

In the stampede to force the world's round, dynamic objects into square, static languages like Ada or C++, we must never forget that such low-level languages - nor even higher-level environments like Objective-C and Smalltalk - are highly unlikely to be accepted by mainstream Age of Information workers. They are more likely to insist on non-textual, non-procedural visual `languages' 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. Card-level objects encapsulate a thread of control alongside whatever lower-level objects were used to build them, so they admit a tangible user interface that is uniquely intuitive for non-programmers (see Figure 5). By definition, card-level 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 software components marketplace, the programmer shortage can be solved as the telephone operator shortage was solved, by making

every computer user a programmer.

References

- [1]*No Silver Bullet; Essence vs Accidents of Software Engineering*, Fred Brooks; IEEE Computer Magazine; April 1987
- [3] *Fabrik; A Visual Programming Environment;* Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle; OOPSLA `88 Proceedings. Metaphor is an office automation product of Metaphor Computer Systems; Mountain View, CA.
- [4] Thomas Kuhn; *The Structure of Scientific Revolutions*; The University of Chicago press; 1962
- [5] *Nuts and Bolts of the Past; A History of American Technology 1776-1860;* David Freeman Hawke; Harper & Row Publishers, New York.

Dr. Cox is a co-founder and Chief Technical Officer of The Stepstone Corporation, the author of *Object-oriented Programming, An Evolutionary Approach,* and originator of the Objective-C System-building Environment. He is presently writing *Object-oriented System-building; A Revolutionary Approach* along the general lines of this article. He can be reached at The Stepstone Corporation; 75 Glen Road; Sandy Hook CT 06482, by telephone at 203 426 1875, or as cox@stepstone.com.

[TM] Objective-C, Software-IC, and ICpak are registered trademarks of The Stepstone Corporation.