Planning the Software Industrial Revolution

Brad J. Cox, *Stepstone Corp.*

Software must stop being a process-centered cottage industry. A product-centered approach that gives equal weight to specification can move software engineering into its industrial revolution.

The possibility of a software industrial revolution, in which programmers stop coding everything from scratch and begin assembling applications from well-stocked catalogs of reusable software components, is an enduring dream that continues to elude our grasp. Although object-oriented programming has brought the software industrial revolution a step closer, commonsense organizational principles like reusability and interchangeability are still the exception, not the rule.

According to historian Thomas Kuhn,[1] science does not progress continuously, by gradually extending an established paradigm. It proceeds as a series of revolutionary upheavals. The discovery of unreconcilable shortcomings in an established paradigm produces a crisis that may lead to a revolution in which the established paradigm is overthrown and replaced.

The software crisis is such a crisis, and the software industrial revolution is such a revolution. The familiar process-centric paradigm of software engineering, where progress is measured by advancement of the software-development process, entered the crisis stage 23 years ago when the term "software crisis" was first coined. The paradigm that may launch the Information Age is the same one that launched the Manufacturing Age 200 years ago. It is a product-centric paradigm in which progress is measured by the accretion of standard, interchangeable, reusable components, and only secondarily by advancing the processes used to build them.

Software crisis

The gunsmith shop in colonial Williamsburg, Va., is a fascinating place to watch gunsmiths build guns as we build software: by 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

0740-7459/90/0900/0025/$01.00 © 1990 IEEE

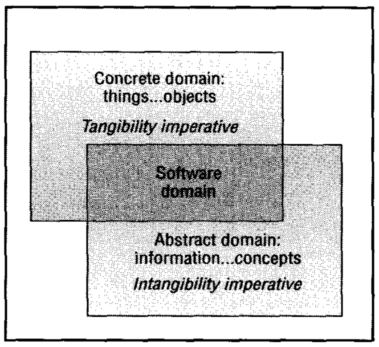


Figure 1. Software is a hybrid, halfway between an abstract idea and a physical, tangible thing. It lies at the overlap of two imperatives: tangibility and intangibility.

iron rod that he'd forged on the anvil behind his shop, cutting its threads entirely by hand and by eye. I was fascinated by how he tested a newly forged gun barrel — charging it with four times the normal load, strapping it to a log, and letting it rip from behind a sturdy shelter — not the least hindered by academia's 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 economic, technological, and cultural realities of colonial America. It made sense 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 to replace the cottage-industry gunsmith's process-centered approach with a product-centered approach: interchangeable parts to address the consumer's demand for less costly, easily repaired products.

The same inexorable pressure is happening today as the cost of 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, of insufficient quality, and its development nearly impossible to manage.

Insofar as this pressure is truly inexorable, nothing we think or do will stand in its path. The software industrial revolution will occur, sometime, somewhere — whether our value system is for it or against it — because it is our consumers' values that govern the outcome. It is only a question of how quickly and of whether we or our competitors will service the inexorable pressure for change.

Software industrial revolution

Contrary to what a casual understanding of the Industrial Revolution may suggest, the displacement of cut-to-fit craftsmanship by high-precision interchangeable parts didn't happen overnight and it didn't happen easily. The heroes of this revolution 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. They stayed busy in their workshops, filing on their iron bars, and left it to their consumers to find another way.

It was actually the ultimate consumer of ordnance products, Thomas Jefferson, who found the solution in 1785 during a visit to France before his presidency. Congress supported his proposal with remarkable steadfastness through 25 years of unsuccessful attempts, such as Ely Whitney's pioneering effort, until John Hall finally succeeded in 1822. An additional 24 years were to elapse before what was then called armory practice spread to private contractors — a half century! This may be a fundamental time constant of such revolutions, since similar lags occurred in the telephony, phonograph, automobile, steamboat, railroad, and other industries. On this time scale, the distinction between evolution and revolution can be exceedingly hard to discern.

Two imperatives. Although I certainly hope that events will prove me wrong, I fear that the software industrial revolution will take as long, or even longer. Figure 1 represents where we are today by juxtaposing two hotly contested imperatives as to how to escape the software crisis.[2,3]

The first viewpoint is the intangibility imperative, which is most common in computer-science circles. Its advocates view software as an solitary, mental, abstract activity akin to mathematics. For example, James Fetzer[3] attributed the following expression of this viewpoint to C.A.R. Hoare: "The construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic."

But this viewpoint is hardly unique to academics. Most programmers will readily relate to this quote, by one of Apple's most creative programmers, which compares programming to novel writing, another solitary, mental, abstract activity: "Reusing other people's code would prove that I don't care about my work. I would no more reuse code than [Ernest] Hemingway would have reused other authors' paragraphs."

The opposing viewpoint is the tangibility imperative, which is most common in software-engineering circles. Its advocates counter that although software is like mathematics or novel writing in many ways, the intangibility imperative works best for problems that do not exceed the abilities or the longevity of an individual. Intangibles are notoriously hard to produce by committee. Furthermore, the intangibility imperative concentrates power in the hands of those with the abstract reasoning skills to comprehend an intangible product: the producers. The consumers are left powerless, unable to contribute the financial and legal resources that are needed to drive deep-seated cultural and technological changes.

This drive to empower the consumer by making software as accessible and immediate as everyday tangible objects underlies the recent enthusiasm for direct-manipulation (iconic) user interfaces, browsers, personal workstations, and other techniques for making software more tangible, less abstract, and more approachable by nonprogrammers.

Hybrid approach. The recent enthusiasm for object-oriented technologies is particularly interesting in that both camps embrace it, but for opposite reasons — tangibility advocates for encapsulation and dynamic binding; intangibility advocates for inheritance and static binding. This apparent convergence but actual divergence is responsible for much of the confusion as to what the term means.

This article does not propose that either imperative should eliminate the other. Software really is an abstract/concrete swamp, a hybrid for which dual perspectives can be useful as first-order approximations to a complex reality. Rather, it proposes a paradigm shift to a new frame of reference in which both imperatives coexist, each predominating at different times and places.

I've emphasized the tangibility imperative throughout this article because of the need to begin building and using commercially robust repositories of trusted, stable components whose properties can be understood and tabulated in standard catalogs, like the handbooks of other mature engineering domains.

The intangibility imperative will always retain its current role as the primary tool of solitary, mental creativity, and will acquire even greater utility for predicting how trusted, quantifiable components, the raw materials of a mature software engineering discipline, will behave when assembled and placed under load.

Object-oriented technologies

Objects are turning up all over! There are object-oriented databases, drawing programs, telephone switching systems, user interfaces, and 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 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 we call software? No one is confused when adjectives like "small" or "fast" mean entirely different things in nuclear physics, gardening, and geology. But in our murky world of intangible abstractions, it is all too easy to lose your 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 visually intuitive, nonsequential technologies like Fabrik[4] and Metaphor Computer System's Metaphor 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 because we invent them all from first principles, everything we encounter there is unique and unfamiliar, composed of components that have never been seen before and will never be seen again and that obey laws that don't generalize to future encounters. Software is a place where dreams are planted and nightmares harvested, where terrible demons compete with magical panaceas, a world of werewolves and silver bullets.

Software is a place where dreams are planted and nightmares harvested, where terrible demons compete with magical panaceas, a world of werewolves and silver bullets.

As long all we can know for certain is the code we ourselves wrote during the last week or so, mystical belief will reign over quantifiable 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 an objective, not a technology for achieving it. It means wielding all the tools we can muster, from well-proven antiques like Cobol to missing ones like specification languages, to enable our consumers by letting them reason about our products via the commonsense skills we all use to understand tangible objects.

But because reverting to this broader meaning might confuse 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 into an organizational enterprise like manufacturing. This means letting consumers at every level of an organization solve their own software problems just as home owners solve plumbing problems: by assembling their own solutions from a robust commercial market in off-the-shelf subcomponents, which are in turn supplied by multiple lower level echelons of producers.

Clearly, software products are different from tangible products like plumbing supplies, and the differences are not small. However, there is a compelling similarity: Except for small programs that a solitary programmer builds for personal use, both programming and plumbing are organizational activities. Both are engaged in by people like those who raised the pyramids, flew to the Moon, repair their plumbing, and build computer hardware — ordinary people with the common sense to organize as producers and consumers of each other's products instead of reinventing everything from first principles.

Value rigidity

Robert Pirsig has related a powerful analogy that well describes the basic problem we software developers face today: the need to think about our values and reevaluate them, rather than rigidly hold on to them even when ultimately fatal. Pirsig wrote,[5] "The most striking example of value rigidity I can think of is the old South Indian monkey trap, which depends on value rigidity for its effectiveness. [The trap is simply a coconut shell with a hole in it, baited with food.] The monkey reaches in and is suddenly trapped — by nothing more than his own value rigidity. He can't revalue the food. He cannot see that freedom without food is more valuable than capture with it. The villagers are coming to get him and take him away. They're coming closer ... closer ... now!

"There is a fact this monkey should know: If he opens his hand he's free. But how is he going to discover this fact? By removing the value rigidity that rates food above freedom. How is he going to do that? Well, he should somehow try to slow down deliberately and go over ground

```
Set* uniqueWords;
Word* currentWord;
uniqueWords = [Set new];
while (getWord(buf) != EOF) {
  currentWord = [Word str:buf];
  [uniqueWords add:current-
Word];
}
printf("unique words = %d\n",
     [uniqueWords size]);
```

Figure 2. This program computes the number of unique words in a document. It turns tokens produced by the getWord subroutine into instances of Word (currentWord) and adds these to an instance of Set (uniqueWords), relying on the set to discard duplicates. As written here, the application is strictly type-checked at compile time. A more flexible solution would have been to declare currentWord to be of type id rather than Word* to delay binding until runtime.

that he has been over before and see if things he thought were important really were important, and well, stop yanking and just stare at the coconut for a while. Before long he should get a nibble from a little fact wondering if he is interested in it."

The software industrial revolution, like all revolutions, is as much cultural as technological. It involves not only tools but values: deeply held beliefs about the nature of software, our role in its construction and use, and ultimately our ideals of good versus bad. Revolutions happen so slowly — and often displace one group by another — because of value rigidity, the inability to relax the pursuit of an older good to gain a newer one. Examples of potential value-rigidity traps abound:

• Programs must be provably correct, as opposed to compliant to specification within a stated tolerance.

• We should focus on solutions that will bear fruit quickly, within a manager's 12-month planning horizon.

• Seamless panaceas are better than kits of diverse tools.

• Software is a closed universe in which all potential interactions between the parts of that universe can be declared when these parts are created by their compiler.

Consider the last two of these in more detail. The closed-universe model surfaces as the belief that compile-time type checking is universally "better." The preference for panaceas instead of tools surfaces as the belief that early and late binding are mutually exclusive panaceas, that the language designer should choose one at the expense of the other rather than providing many binding technologies that the user can choose from according to the job at hand.

To see how these value systems keep us trapped in the software crisis, consider the program in Figure 2. This program was written according to the closed-universe model: Everything is declared in advance so the compiler can prevent a supposedly common kind of error through strong type checking.

But in return, this programmer has given up the opportunity to escape the software crisis by building a market in reusable software components, such as Set. To see why, shift your focus from the assembly to the parts that make it up — from the application to the Set class. For Set's add: method to be consistent with the type declaration in the application (Word* currentWord), the compiler must insist that the formal argument of Set's add: method, aMember, be declared as Word*, ByteArray*, or AbstractArray*, depending on which declares the messages that add: will send to aMember. The closed-universe model forces Set's designer to anticipate what consumers will use Set for — at compile time — when Set is produced by the compiler, rather than at runtime, when it is used by installing it into its reuse environment.

But what about consumers whose set members are not implemented as a subclass of AbstractArray? Why not sets of Wrenches or sets of Unicorns? Why not sets of Sets? A commercial Set must be reusable for members of any class, regardless of how its consumer chose to implement them — regardless of the members' inheritance hierarchy.

But this involves an open-universe model in which Set's members are checked not when Set is compiled, but when it is used — when it is drawn from a library of compiled code and first encounters its members at runtime. Set must be written so its members may be chosen after Set has been compiled, packaged in a library, and delivered in binary form through the market in software components.

In Objective-C, the syntax for doing this is to replace the offending type declaration, Word*, with the anonymous type name, id. This instructs the compiler that the add: method is prepared to deal with instances of any class, excluding only types other than id. Because type checking and binding are now deferred until runtime, Set can be compiled once and for all and distributed in binary form for reuse as a commercial software component. By relaxing our frantic grasp on the compile-time type-checking bait, we've taken a step toward freedom from the software crisis.

The point of this example is not that runtime type checking and anonymous types are better or worse than the converse. Although the example used dynamic binding to attach members to sets, it used static binding to build the getWord() subroutine. The example shows that seamless panaceas — languages that do not offer multiple modularity/binding technologies — prevent these markets by failing to provide modularity/binding technologies suitable for users at different levels of the producer/consumer hierarchy.

Strongly coupled languages like Ada and C++ are deficient insofar as they do not also support loosely coupled modularity/binding technologies like screwing, bolting, welding, soldering, pinning, riveting, and mortising. And loosely coupled environments like Smalltalk are deficient insofar as they fail to support tightly coupled modularity/binding technologies like forging and casting. Hybrid environments like Objective-C and CLOS, and analogous ones that could be based on Cobol, Ada, Pascal, or C++, support multiple modularity/binding technologies as tools to be picked up or laid aside according to the job at hand.

Software architecture

It is easy to see how interchangeable parts could help in manufacturing. But manufacturing involves replicating a standard product, while programming does not. Programming is not an assembly-line business but a build-to-order one, more akin to plumbing than gun manufacturing.

But the principles of standardization and interchangeability pioneered for standard products apply directly to build-to-order industries like plumbing. They enabled the markets of today where all manner of specialized problems can be solved by binding standardized components into new and larger assemblies.

Mature industries like plumbing are less complex than ours, not because software is intrinsically more complicated, but because they — and not we — have solved their complexity, nonconformity, and changeability problems by using a producer/consumer hierarchy to distribute these problems across time and organizational space. The plumbing supply market lets plumbers solve only the complexities of a single level of the producer/consumer hierarchy without having to think about lower levels, for example, by reinventing pipes, faucets, thermostats, and water pumps from first principles.

Kuhn has pointed out that the crucial test of a new paradigm is whether it reveals a simpler structure for what was previously chaotic. Certainly, the process-centric software universe is chaotic today. The process-centric paradigm requires that only one process be "right," so each new contender must slug it out for that coveted title, "standard" — rapid prototyping versus Mil-Std-2167, object-oriented versus structured, Ada versus Smalltalk, C++ versus Objective-C. Different levels of the producer/ consumer hierarchy are not permitted to use specialized tools for their specialized tasks, skills, and interests, but must fit themselves to the latest panacea.

But by focusing on the nature of the *products* of these languages and methodologies, rather than on them as processes significant unto themselves, a simpler pattern emerges that is reminiscent of the distinct integration levels of hardware engineering (shown in Figure 3). The hardware community's monumental achievements are largely due to the division of labor made possible by the loosely coupled modularity/binding technologies shown in this figure.

Card-level pluggability lets users plug off-the-shelf cards to build custom hardware solutions without having to understand soldering irons and silicon chips. (Pluggability is the ability to bind a component into a new environment dynamically, when the component is used, rather than statically, when it is produced.) 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 the consumer needn't know or care how components from a lower level were implemented — just how to use them to solve the problem at hand.

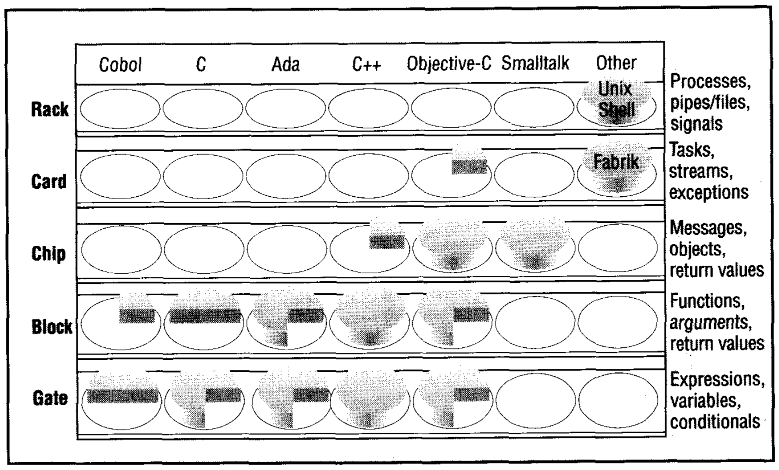

Cobol
C
Ada
C++
Objective-C
Smalltalk
Other
Rack
Card
Chip
Block
Gate
Unix Shell
Fabrik
Processes, pipes/files, signals
Tasks, streams, exceptions
Messages, objects, return values
Functions, arguments, return values
Expressions, variables, conditionals

Figure 3. Hardware engineering's levels of integration are a good model for software engineering. "Object-oriented" means different things at different levels of integration. The pies show the extent to which several popular languages support work at each level.

Each integration level in this figure already exists in software, scattered among competing languages. Asking "What is object-oriented?" is like asking for a context-independent meaning of "small." Just as it is proper that "small" means entirely different things for atoms, spark plugs, automobiles, and traffic jams, it is proper that "object" mean different things at each integration level.

Commercial example

My company, Stepstone, was founded seven years ago to pursue the ideas in this article on a commercial scale. Its mission is providing pluggable chip-level software components to C system builders who realize that building large systems (rack-level objects) with only the modularity/binding technologies that C provides is equivalent to wafer-scale integration, something that hardware engineering can barely accomplish to this day.

However, C did not support pluggability, so we built an enabling technology: a C preprocessor to support a loosely coupled chip-level modularity/binding technology within C's gate- and block-level technologies.

The goal of this extended language, Objective-C, was not to be yet another language nor, as originally envisioned, to repair C's long-standing deficiencies (subsequent versions have since addressed many of these deficiencies by supporting ANSI improvements, even for older C compilers that don't provide them). It was to provide enabling technology for building the Objective-C system-building environment and a multilevel market in software components.

Stepstone's experience amounts to an experimental study of the software-components market strategy in action.

The good news is that, with substantial libraries now in the field and others on the way, the chip-level software-components market concept has been tried and proven sound for diverse applications in banking, insurance, factory automation, battlefield management, CAD/CAM, CASE, and others. The component libraries have proven to be flexible enough, and easy enough to learn and use, that they have been used to build highly graphical applications in all of these domains, and sufficiently portable that it has been possible to support them on most hardware and software platforms during this era of rapid platform evolution.

The bad news is that this experiment has shown that it is exceedingly difficult, even with state-of-the-art technologies, to design and build components that are both useful and genuinely reusable, to document them so 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 existing interface, and to market them to a culture whose

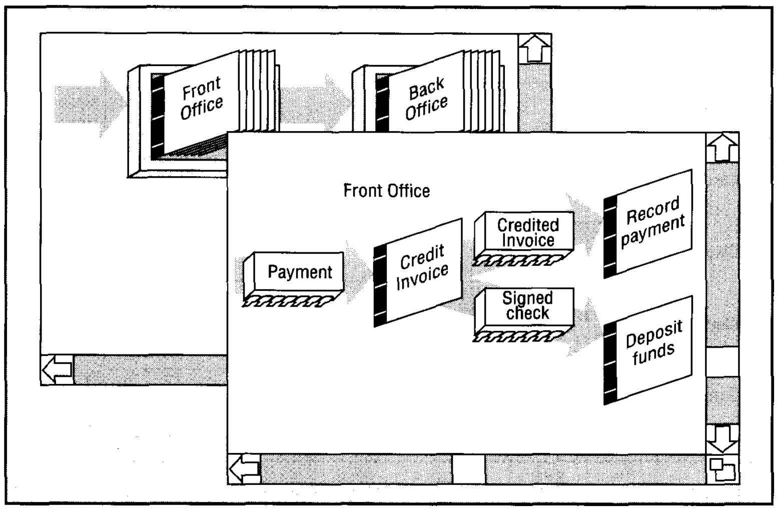


Figure 4. Example of how a nonprogrammer might construct a banking application by assembling software cards, or dataflow modules.

value system, like the Williamsburg gunsmith, encourages building everything from first principles.

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 like browsers, Objective-C is still thought of as yet another programming 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.

This experiment has also shown 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. At the gate and block levels of Figure 3, "object-oriented" means encapsulation and sometimes inheritance, but the dynamism of everyday objects has been intentionally relinquished in favor of machine-oriented virtues like computational efficiency and static binding.

At the intermediate (chip) level, the open-universe model of everyday experience is introduced. At this level, all possible interactions between parts and the whole do not have to be known and declared in advance, when the universe is created by the compiler.

The next step

But on the scale of any significant system, gate-, block-, and even chip-level objects are extremely small units of granularity — grains of sand where bricks are needed. Furthermore, since chip-level objects are no less procedural than conventional expressions and subroutines, they are just as alien to nonprogrammers. 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.

Fabrik[4] pioneered a path that I hope the object-oriented world will soon notice and follow. Because Fabrik was written in Smalltalk, it consists internally of chip-level objects. But externally, it projects a higher level kind of object, a card-level object, to the user. These higher level objects communicate, not synchronously through procedural invocation, but asynchronously by sending chip-level objects through communication-channel objects such as Streams. They amount to a new kind of object that encapsulates a copy of the machine's thread of control on a software card, along with the chip-level objects used to build that card. Software cards are objects of the sort that programmers call lightweight processes; objects that operate as coroutines of one another, not subroutines.

Because software cards can operate concurrently, they admit a tangible user interface (as Figure 4 shows) that is uniquely intuitive for nonprogrammers and, for this reason alone, more fundamentally object-oriented than the procedural, single-threaded 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 procedural invocation, they don't support inheritance, and their user interface is iconic, not textual.

By adding these, and probably other, architectural levels, each level can cater to the needs, skills, and interests of a particular constituency of the software-components market. The programmer shortage can be solved as the telephone-operator shortage was solved: by making every computer user a programmer.

Product-centered paradigm

Once Congress mandated interchangeable parts for government-purchased equipment, technologies for building them followed posthaste. The first phase focused on water-powered tools like Blanchard's pattern lathe. This lathe represents the mainstream approach of the software community until now: the search for processes that can transform progressively higher level descriptions of the problem, like the gun-stock pattern that guides this lathe, into concrete products that are correct by construction.

Today, this lathe stands in the American History Museum in Washington, D.C.— but its stands there to point out that the crucial innovation that made interchangeable parts possible was not only the elimination of cut-to-fit craftsmanship by tools like this lathe. The crucial discovery was made, not by Blanchard, but by John Hall, who realized that implementation tools were insufficient unless supplemented by specification tools capable of determining whether parts complied to specification within tolerance. The museum reinforces this point by displaying a box of hardened steel inspection gauges in the same exhibit with Blanchard's lathe.

This discovery has not yet occurred in software. Although it is easy to find articles with specification in their title, they usually mean implementation, as in "automatically generating code from specifications." Although such higher level implementation tools are clearly worthwhile, it is misleading to call them specification languages, because this obscures the absence of true specification technologies in software — the rulers, protractors, calipers, micrometers, and gauges of manufacturing. Programmers continue to rely exclusively on implementation technologies, be they ultrahigh-level programming languages — tools of certainty like the Blanchard lathe — or lower level languages — tools of risk like the rasps, files, and spokeshaves that it displaced.

Confusing implementation and specification. The confusion of implementation and specification is particularly prominent in that most fashionable of object-oriented features: inheritance. As used in object-oriented-language circles, inheritance is the Blanchard lathe of software — a powerful and important tool for creating new classes from existing ones, but not nearly as useful for specifying static properties like how they fit into their environment, and useless for describing dynamic properties like what these classes do.

Rather than laboriously building each new class by hand, inheritance copies functionality from a network of existing classes to create a new class that is, until the programmer begins overriding or adding methods, correct by construction. Such hierarchies show how a class's internals were constructed. They say nothing, or worse, mislead, about the class's specification — the static and dynamic properties that the class offers its consumers.

For example, Figure 5 shows the implementation hierarchy for a Semaphore class that inherits four existing classes. Compare this implementation hierarchy with the following facts:

• Semaphores are a kind of Queue only from the arcane viewpoint of their author. This hierarchy resulted from a speed optimization of no interest to consumers, who should view them as scheduling primitives with only wait and signal methods. How will the producer tell the consumer to avoid some, many, or all of those irrelevant and dangerous methods being inherited from all four superclasses? And if such static issues are handled statically, where will dynamic ones be handled? — the vital question of what Semaphore does.

• This hierarchy says explicitly that OrderedCollection is similar to Set and dissimilar to Queue. However, exactly the opposite is true. Queue is functionally identical to OrderedCollection. I carefully hand-crafted Queue to have each of OrderedCollection's methods, each with precisely the same semantics, to show that encapsulation lets a class's internals be revised without affecting its externals. But how will the consumer discover that OrderedCollection and Queue provide the same functionality — that they are competing implementations of precisely the same specification? And how will any commercially significant differences, like time/space trade-offs, be expressed independently of these similarities?

Parallel tools. Manufacturing handles this issue by providing two separate classes of tools: implementation tools for the producer's side of the interface and specification tools for the consumer's side. Shouldn't we do likewise? Shouldn't conceptual aids like inheritance be used on both sides, but separately, just as Blanchard's lathe and Hall's inspection gauges deal with different views of the same object's interface? Shouldn't Semaphore's consumer interface be expressed in an explicit specification hierarchy as in Figure 6, independently of the producer's implementation hierarchy?

Just as a measuring stick is not a higher level saw, a specification tool is not a higher level implementation tool. Specification is not the implementation tool's job. Separating the two would eliminate performance as a constraint on the specification tool, letting knowledge representation shells that already support rich conceptual relationships be used as a basis for a specification-language compiler. The two tools can be deployed as equal partners, both central to the development process, as in Figure 7.

The primitives of the specification language are ordinary test procedures — predicates with a single argument that identifies the putative implementation to be tested. A test procedure exercises its argument to determine whether it behaves within its specification's tolerence. For example, a putative duck is an acceptable duck upon passing the isADuck gauge. The specification compiler builds this gauge by assembling walksLikeADuck and quacksLikeADuck test procedures from the library. The compiler is simply an off-the-shelf knowledge-representation tool for invoking the appropriate test procedures stored in the test-procedure library.

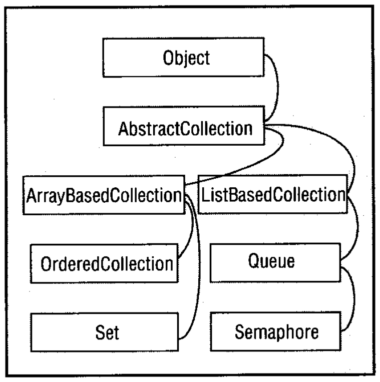


Figure 5. Implementation hierarchy showing how new classes are implemented from existing ones. However, it provides a misleading picture of what the customer is interested in: the specification of what the classes do.

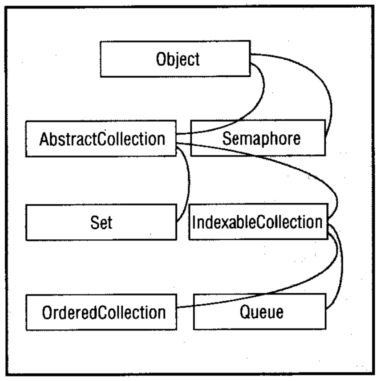


Figure 6. Specification hierarchy for the example in Figure 5. This specification hierarchy uses the names of test procedures, not classes, that verify (gauge) whether an implementation compiles to a specification within a tolerance embodied in each test.

How it works. The test-procedure libraries play the role in software as elsewhere — defining the shared vocabulary that makes producer/consumer dialogue possible. For example, in "I need a pound of roofing nails," "pound" is defined by a test procedure involving a scale, and "nail" by a test procedure involving shape recognition by the natural senses. Test procedures are particularly crucial in software because of the natural senses' inability to contribute to the specification of other-

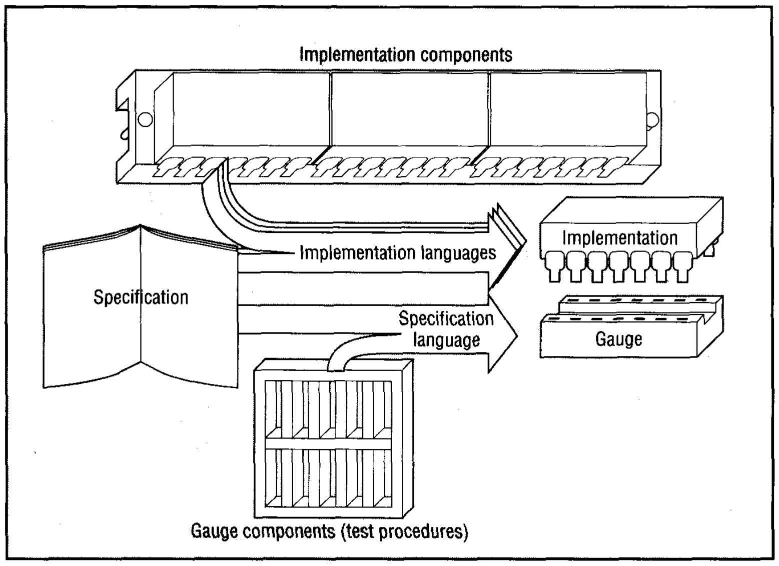


Figure 7. A development process in which specification is given the same emphasis as implementation.

wise intangible software products like Stack or Set. Making software tangible and observable, rather than intangible and speculative, is the first step to making software engineering and computer science a reality.

Test procedures collect operational, or indirect, measurements of what we'd really like to know, the product's quality as perceived by the customer. They monitor the consumer's interface, rather than our traditional focus on the producer's interface (by counting lines of code, cyclomatic complexity, Halstead metrics, and the like). This knowledge of how product quality varies over time can then be fed back to improve the process through statistical quality-control techniques, as described by W. Edwards Deming[6], that play such a key role today in manufacturing.

Implications. The novelty of this approach is threefold:

- applying inheritance concepts not only to implementation, but to specification and testing, thus making the specification explicit,
- preserving test procedures for reuse across different implementations, versions, or ports through an inheritance hierarchy, and
- distributing the specifications and test procedures between producers and consumers to define a common vocabulary that both parties can use for agreeing on software semantics.

The implications could be immense, once we adjust to the cultural changes that this implies: a shift in power away from those who produce the code to those who consume it — from those who control the implementations to those who control the specifications. Three implications are:

- Specification/testing languages could lead to less reliance on source code, new ways of documenting code for reuse, and fundamentally new ideas for classifying large libraries of code so it can be located readily in reference manuals, component catalogs, and browsers.
- Specification/testing languages could free us from our preoccupation with standardized processes (programming languages) and our neglect of standardized products (software components). Producers would be freed to use whatever language is best for each task, knowing that the consumer will compile the specification to determine whether the result is as specified.
- Specification/testing languages can provide rigor to open-universe situations when compile-time type checking is not viable. For example, in the set example described earlier, the implementation-oriented declaration AbstractArray* was too restrictive because sets should work for members that are not subclasses of AbstractArray. However, the anonymous type id is unnecessarily permissive because sets do impose a protocol requirement that you'd like to check before runtime. But because specification/testing tools can induce static meanings (isADuck) from dynamic behavior (quacksLikeADuck), why not feed this back to the language as implementation-independent type declarations? This amounts to a new notion of type that encompasses both the static and dynamic properties, rather than the static implementation-oriented meaning of today.

At Stepstone, implementing software components has never been a big problem, but making them tangible to consumers has been. The marketing department experiences this in explaining the value of a component to potential customers. Customers experience it when they try to find useful components in libraries that are organized by inheritance hierarchies and not by specification hierarchies. And the development team experiences it when changing a released component in any fashion, such as when porting it to a new machine, repairing a fault, or extending it with new functionality.

Without tools to express the old specification independently from the new and then determine if the old specification is intact while independently testing the new one, development quickly slows to a crawl. All available resources become consumed in quality assurance.

Of course, intangible software components are quite different from the tangible components of gunsmithing and plumbing, and the differences go even beyond the abstract/concrete distinction of Figure 1. The most fundamental differences include

- complexity, nonconformity, and mutability,
- intangibility (invisibility),
- single-threadedness, and
- ease of duplication.

This list originated in a list Fred Brooks[7] provided to distinguish "the inescapable essence of software, as opposed to mere accidents of how we produce software today." I added two properties that he neglected to mention: single-threadedness and ease of duplication. However, I did not do this to reinforce his implication that the software crisis is an inescapable consequence of software's essence. I did it to argue that all the items on this list are only obstacles that can, and will, be overcome:

- A robust software-components market addresses the complexity, nonconformity, and mutability obstacle by providing an alternative to implementing everything from first principles.

• Object-oriented programming technologies at each of the many levels of Figure 3 address the intangibility obstacle, particularly if supported by a specification technology as in Figure 5.

• Ultrahigh-level (card- and rack-level) object-oriented languages also address the intangibility obstacle by providing nonprocedural objects that nonprogrammers can manipulate via direct-manipulation user interfaces.

• Card- and rack-level languages address the single-threadedness obstacle by relaxing the relentlessly procedural, single-control-thread restrictions of the gate-, block-, and chip-level languages of today.

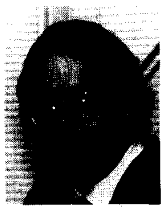
The problem, of course, is the absence of easy technological solutions to the ease-of-duplication obstacle, which stands ready to undercut any assault we might mount on the others by inhibiting the growth of a commercially robust software-components market. Such markets will be weak as long as it is difficult for component producers to guarantee income commensurate to their investments by controlling the ability to replicate copies.

But as long as the pressure for change increases as the software crisis grows ever more severe, solutions to even this last obstacle will be sought — and they will be found, either by us or our competitors. Ideally, the solutions will be technical. Perhaps they will be legal, economic, and policy sanctions imposed by our consumers. Perhaps they will be the kind of sheer determination that the free-enterprise system is famous for inspiring.

I only wish that I were as confident that the changes will come quickly or that we, the current software-development community, will be the ones who make it happen. Or will we stay busy at our terminals, filing away at software like gunsmiths at iron bars, and leave it to our consumers to find a solution that leaves us sitting there? ❖

References

1. T. Kuhn, *The Structure of Scientific Revolutions,* Univ. of Chicago Press, Chicago, 1962
2. E. Dijkstra, "The Cruelty of Really Teaching Computer Science," *Comm. ACM,* Dec. 1989, pp. 1,398-1,404.
3. J.H. Fetzer, "Program Verification: The Very Idea," *Comm. ACM,* Sept. 1988, pp. 1,048-1,063.
4. D. Ingalls et al., "Fabrik: A Visual Programming Environment," *Proc. Object-Oriented Programming, Systems, Languages, and Applications Conf.,* ACM, New York, 1988.
5. R.M. Pirsig; *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values,* Quill William Morrow, New York, 1974.
6. W.E. Deming, *Out of the Crisis,* Center for Advanced Engineering Study, Massachusetts Inst. of Technology, Cambridge, Mass., 1989.
7. F.P. Brooks, "No Silver Bullet: Essence vs. Accidents of Software Engineering," *Computer,* April 1987, pp. 10-19.



Brad J. Cox is a cofounder and chief technical officer of Stepstone Corp. and the originator of the Objective-C system-building environment and many of its libraries.

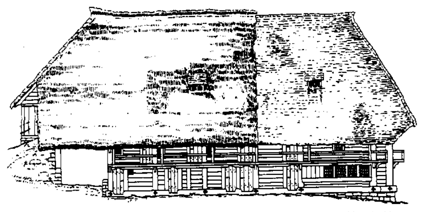
Cox received his PhD from the University of Chicago and carried out postdoctoral studies at the National Institutes of Health and at the Woods Hole (Mass.) Marine Biological Laboratories. He is a member of ACM and IEEE.

Address questions about this article to the author at Stepstone Corp., 75 Glen Rd., Sandy Hook, CT 06482; Internet cox@stepstone.com.

2i Industrial Informatics

The Fine Art of Construction

Innovation Meets Quality



One of many Black Forest Farmhouses near Freiburg, hundreds of years old. A highly functional architecture, skilled craftsmanship and selected construction materials are proofs for a qualified engineering know how generated in the heart of Europe. It's there nearby Switzerland where German quality also meets French innovation.

Our company 2i is located in the capital of this area and we provide Software Technology of true German-Swiss quality: MOCKA®- *a Modula-2 Programming Environment for serious and demanding programmers:*

- *Rich and easy-to-use tool set including compiler, editor, debugger, library, language interfaces etc.*
- *Top compiler performance on SUN, VAX, SONY and others*
- *High quality of generated code in performance and absence of bugs*
- *Module libraries for reusability*
- *Conforming to evolving standards*
- *Easy to learn - Only a small step from PASCAL or C*
- *Safe investment when moving via Modula-2 in small and cost saving steps towards next generation VHLLs like Ada.*

Please write, fax or phone to
2i Industrial Informatics
SYSTEMATIC Software Technology Group
Haierweg 20e
D-7800 Freiburg i.Br., West-Germany
Tel.: ++49- 761- 4 22 57
FAX: ++49-761-47 43 12
Partners in the U.S. and in Japan welcome for distribution.

MOCKA *is a registered trademark of the GMD Research Group, Karlsruhe.*

Reader Service Number 7