Editor: Bertrand Meyer, Eiffelsoft, 270 Storke Rd., Ste. 7, Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

The Next Software Breakthrough

Bertrand Meyer, Eiffelsoft

Software development evolves less quickly than the industry's own hype would suggest: The basic issues surrounding the design and construction of software have remained unchanged throughout its history. Nevertheless, it is possible to identify five breakthroughs that have occurred since the time software development was recognized as a discipline in its own right:

- Structured programming, in the early '70s.
- The rise of the C culture (including make, pipes, shells, scripting languages), in the late '70s.
- The PC revolution, in the mid '80s.
- Object technology, in the early '90s.
- The Internet upheaval, happening now.

These approximate times do not indicate when the ideas were invented (object technology, for example, dates back to Simula in 1967) but rather when they started to affect a large segment of the industry. Also, the list does not imply a personal judgment that these events were intellectually the most important (although I do think they were all beneficial); they are simply the ones that had the most effect on the industry. Using another criterion would have resulted in a different list (Lisp, for example, is not a breakthrough in the sense used here but was an intellectual milestone). Also note that the third and fifth items are not strictly software advances: As Niklaus Wirth remarked many years ago, our field remains driven in large part by progress in hardware.

So what will be the sixth breakthrough? To predict what might come next, it is useful to consider two breakthroughs that did *not* happen.

WHAT DIDN'T BREAK THROUGH

Software reuse did not happen. Although some reusable software elements have achieved a remarkable degree of success, especially in the Windows world, we have failed so far to realize the vision of refounding the software field as a components-based industry. The common explanation for the failure of reuse is that it requires a better management infrastructure. I disagree with this conclusion, because I think the major issues are in fact technical (see "The Reusability Challenge," Feb. 1996, pp. 76-78).

Adoption of formal methods did not happen. Here too we have seen some isolated successes (not to mention incessant discussions in the pages of *Computer* and *IEEE Software*), but the effect on the industry as a whole has been little more than a scratch on the surface. To explain the failure of formal methods, many point to the typical programmer's lack of mathematical training and to the constraints of industrial development.

These explanations don't seem quite right to me; they are too narrow. I think they miss the basic issue: Reuse can only succeed with some injection of formal methods. And the only chance for formal methods to succeed on a large scale is if they are applied to the development of reusable components.

BUILDING A NEW FOUNDATION

Let me start with the second proposition.

Formal methods (such as Z, VDM, Larch, and B) apply mathematical techniques to specify, document, and whenever possible validate software. In spite of many advances in making these techniques easier to apply, they still require extra effort and some mathematical sophistication. In other words, they add to development cost at a time when our industry is more focused on productivity (reducing costs) than quality (getting better software in the long term).

This explains why the use of formal methods has largely been confined to expensive mission-critical systems—and in fact to only a subset of such projects.

In the context of developing reusable components, however, things change completely! Now the extra effort becomes economically justifiable. If you are building a software component of which you hope to sell tens or hundreds of thousands, then all the mathematical apparatus that looked too expensive on a one-shot development start making a lot of sense. If formal techniques let you catch just one bug before the component reaches your customers, they will more than pay for themselves.

Reuse can only succeed with formal methods. And formal methods can only succeed if used to develop reusable components.

Which brings us to the potential for widespread reuse. In my opinion it is foolish to think that new approaches being proposed for reuse—such as CORBA and Java Beans—can succeed without some degree of formalism. How can you hope to release thousands of components without systematically including what earlier columns here have introduced as the basic form of component specification: the contract?

A contract indicates what each component expects from its clients (the precondition), what abstract properties it ensures on its results (the postcondition), and what global conditions it maintains (invariants). The ominous counterexample is the $500 million failure of the Ariane-5 rocket launcher, caused by the reuse of a component that was not equipped with the proper specification (J.-M. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane," Jan. 1997, pp. 129-130).

In the absence of a systematic approach to reuse based on formal methods, we risk many more Ariane-like failures. Moreover, reuse will not take off, as all the propaganda in the world cannot make up for precise, clear specifications. The usual analogy from the hardware field is clear enough here: What hardware engineer would reuse an electronic component without an impeccable specification of its inputs, its outputs, and its operating conditions?

This is the negative view. But the same idea can also be rephrased positively: if we are ready to make the necessary effort to specify and validate our components formally—an effort that is, as noted, economically justified because of the size of the potential market—then the vision of a component-based software industry *can* become a reality.

The components should be made available in major OO languages (such as C++, Java, Smalltalk, and Eiffel), with IDL interfaces for CORBA integration, and perhaps versions in non-OO languages (such as C). All these versions should be derived—as automatically or at least as systematically as possible—from a base version developed in a language supporting formal techniques. I would submit Eiffel as the candidate to play that role, because it combines OO mechanisms and built-in support for design by contract. Others might prefer to start from a formal specification language such as B or Z.

No other idea could do remotely as much for the software industry as the movement toward using a set of formally specified, fundamental software components.

Is this move to a set of formally specified fundamental components desirable? Yes. It is the best thing that could happen to the industry today. I know of no other idea that could do even remotely as much to address software's most crying deficiencies. And nothing else would improve just about everything—productivity, reliability, ease of adaptation—by a quantum leap.

Will it happen? I don't know. We, the actors in the software game (developers, managers, and users) are the ones who will decide, through our actions more than our words, whether we really want to carry out the next breakthrough in software. ❖