Every Little Bit Counts: Toward More Reliable Software

Bertrand Meyer, ISE

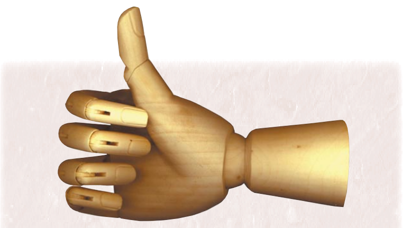
So far, we've had it easy. The public has been remarkably tolerant of our collective inability to produce high-quality software.

True, by and large, software works much of the time, more or less. If society hadn't come to rely so fundamentally on our profession, Y2K, for one, wouldn't be such a big deal. But the overall service that we render is not good enough by any measure. A year ago, Ted Lewis produced some pretty scary statistics about the huge waste of resources that poor quality causes ("Joe Sixpack, Larry Lemming, and Ralph Nader," *Computer*, July 1998, pp. 107-109).

The public's tolerance will not last forever because people are increasingly resenting the bugginess of much of the software we produce. Consider this extract from Walter F. Mossberg's recent column in the *Wall Street Journal* ("Personal Technology," 30 Sept. 1999, MarketPlace, p. 1):

> Please don't bother to write to me with suggestions to fix the [numerous Windows-related problems discussed in the article]. The whole point is that owners of computers shouldn't have to get involved in making them work as promised. They should just work, all of the time.

Editor: Bertrand Meyer, Interactive Software Engineering, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com



Imagine software reviews that tell us how good the software really is without just counting the bells and the whistles.

Not so long ago, technology dazzled editors of mass circulation magazines; now those editors are beginning to demand quality. Even the professional press is starting to worry; consider this extract from a recent Nicholas Petreley column in *InfoWorld* ("Silence Is Deadly…," 16 Aug. 1999, p. 114):

> Software publishers aren't interested in writing solid, bug-free code because they are convinced that features sell, not quality... . Computer journalists should focus less on features and more on reliability when reviewing software. More importantly, we should go out of our way to rip out the fingernails and rearrange the face of any vendor that delivers programs with security holes and bugs.

That would be big news indeed; imagine software reviews that tell us how good the software really is without just counting the bells and the whistles.

One of the most critical components of software quality is reliability. Efforts to improve reliability are not new. In fact, there are many different approaches.

REMAINING BROAD-MINDED

The group of approaches I've outlined here is a little unconventional because of its breadth. One of the characteristics of the software engineering community is that it's sometimes split into separate communities, each of which suggests that you "just do this" and everything will work fine. Of course each community defines "this" differently.

For example, you have

- the management school, which holds that all that really matters are better management approaches;
- the formal specification school, which suggests we won't achieve anything unless we specify everything mathematically—and then we won't need testing at all;
- the testing school, which views formal specifications as an academic pastime and believes that the only meaningful solution is to devise systematic testing strategies;
- the metrics school, which focuses on assessing everything quantitatively;
- the open source crowd, which believes that only by extensive public scrutiny can we successfully develop reliable software.

Each of these schools holds a piece of the truth, but none of them holds the whole truth.

Any "just do this" approach is wrong; the problem is far too complicated for easy solutions. Although I have contributed a few suggestions myself in the form of methodological principles, language support, and software tools, I do not think that any single solution can carry the day. The best position is believing that every little bit helps; we must keep an open mind and avoid ruling out good ideas.

What follows is a list of good ideas. I will certainly have forgotten some, but I welcome reader comments that point to such omissions.

PREVENTION AND CURE

Separate from their classification as management and technology, reliability techniques fall into two categories:

- a priori techniques, which strive to build software right;
- a posteriori techniques, which attempt to right the wrongs.

Formal specification is an example of a priori; testing, of a posteriori.

Much of the practice today is in a posteriori techniques. We build software that's not very good and, through brute force, debug it into correctness. I certainly won't suggest that, given current techniques, we should test our software less. But by shifting some of the balance to a priori efforts, we may go a long way toward correcting some of the most serious problems. We need both cure and prevention, but an ounce of prevention saves a lot of cure.

MANAGEMENT APPROACHES

The management school reminds us that good engineering practice means, among other things, well-defined management policies.

Process-based

The Capability Maturity Model and, to a lesser extent, the ISO-9000-based approaches have had positive effects in some segments of the industry, mostly among large companies. Forcing organizations to understand, document, and control their software process—and make it reproducible—is a definite benefit.

Such approaches have been criticized as being focused too much on form and not enough on substance: It is good to count bugs and to track delays, but it would be better to eliminate them. This doesn't mean the ideas are bad, but it does mean that these approaches have to be combined with more technology-oriented solutions. The Jet Propulsion Laboratory's ISO-9000 certification ("we have a document for everything") wasn't enough to avoid the loss of the Mars Climate Orbiter, caused by a piece of software that used an English-unit value while the rest used metric.

Closed and open

A large part of the industry attempts to follow Microsoft—"the market leader"—but doing so doesn't guarantee quality. In fact, some would say it guarantees the reverse, which is an exaggeration. Microsoft's software model does ensure that successful products benefit from a critical mass.

We build software that's not very good and, through brute force, debug it into correctness.

At the other end of the spectrum you have the open-source and free-software enthusiasts who believe in the power of public scrutiny. While open source and free software are attractive ideas, to my knowledge no one has as yet provided formal evidence of the superiority of software produced that way, although informal examples are not hard to find. Ken Thompson's recent disparaging comments about the quality of Linux code ("Unix and Beyond: An Interview with Ken Thompson," *Computer*, May 1999, pp. 58-64) are a little sobering.

Quantitative

Metrics are also frequently cited. It is clear that we need more quantitative approaches to assessing and predicting what we do. Project metrics (time, people, and money) are just as necessary as product metrics (bugs, size, and complexity).

Education

Education is critical. Software engineers too often do not know some of the basic techniques of modern computer science. I always find it striking that only a minority of professionals know, for example, the Hoare approach to semantics, which is perhaps the closest approach we have to some of the basic scientific laws of other fields (such as Ohm's laws). We need better initial training and, perhaps even more importantly, retraining and continued education for working professionals.

The Dilbert approach

Some management-style approaches are unconventional. Kent Beck's extreme programming, for example, is in part a Dilbertian revolt against management-imposed organizational fiats à la CMM. Extreme programming is perhaps shocking to some—and not everything in it is valuable—but there are certainly lessons to be learned by everyone.

TECHNICAL APPROACHES

Management isn't everything; we need better technology. Since the possibilities for technological improvements are diverse, the technical approaches are promising.

Formal methods

It's impossible to ignore formal methods. True, they have a bad reputation in some circles as being too heavy and difficult. That reputation, however, is not entirely justified. Formal methods have achieved a number of successes. They are the only game in town when it comes to guaranteeing a regulatory authority that you have produced a correct engineering design.

Just as importantly, learning to apply formal methods makes you a much better developer even for projects in which you don't work in a completely formal way. Those who think of formal methods as an academic curiosity with no future may be in for a few surprises. As I noted in an earlier column ("The Next Software Breakthrough," July 1997, pp. 113-114), formal methods hold a particular promise in connection with reuse: Reusable components need strong warranties, and formal-methods costs can be justified economically by the economies of scale permitted by reuse.

Design by contract

A more moderate version of formal methods, closely combined with the principles of object technology, is design by contract, which several installments of this column have described and advocated. I believe it's one of the most potent ideas for improving the state of software technology.

Testing

Approaches to testing are becoming more systematic. We all know that testing cannot come even remotely close to exhaustiveness. But this doesn't mean we can't be systematic and effective in our testing strategies. Recent work on testing shows how to breathe new life into classical techniques—such as mutation testing—in connection with the newer ideas of object technology and contracts.

Modern languages and techniques

Modern languages definitely help. True static typing catches many potentially serious errors before they have had the time to strike. Automatic memory management avoids vicious bugs. Avoidance of dangerous features such as pointer arithmetic frees us from many potential disasters.

Object technology has brought us tremendous improvements. What's perhaps critical here is the dramatic simplification that well-applied OO software construction brings to software architectures. Simplicity is key to quality and especially to reliability: You can't get it right if it's complex. These benefits of object technology, however, assume that it's applied systematically, almost dogmatically. Incomplete or half-hearted approaches, especially if they don't fully apply principles of data abstraction and information hiding, are not much better than pre-OO techniques.

Component-based development

Component-based development (CBD) holds of course a great promise, which I've described in previous columns. Here too, the techniques must be applied properly. In particular, we badly need more expressive Interface Definition Languages for both CORBA and COM to support the expression of semantic constraints.

Serious component-based development has the potential of reengineering the industry. Whether we will be able to realize that promise depends on how seriously we take the software engineering principles—without which there can't be any serious CBD.

None of the ideas outlined here will suffice to provide the breakthrough that our field requires today. But if we take all of them seriously and succeed in combining them, we may be able to realize major advances.

It will take a lot of prevention and a lot of cure. ❖

Be a part of the HOTTEST new contest for computer engineering students!

The search is on for teams of undergraduate students from around the world to compete in the first annual IEEE Computer Society International Design Competition.

- **Use state-of-the art components to solve real-world problems!**
- **Compete for cash prizes of up to $15,000!**
- **Apply what you've learned!**
- **Bring fame and glory to your school!**

PRIZES

First place	$15,000
Second place	$10,000
Third place	$6,000
Fourth place	$3,000
Fifth place	$2,000
Honorable mention	$1,000

First, second, and third place teams also each receive a financial aid fund for their schools.

Important Dates

Applications due
1 December 1999

Projects due
4 May 2000

Top 10 projects selected
2 June 2000

CSIDC World Finals in Washington, DC
25-27 June 2000

For more information or to apply online, see computer.org/csidc/