Jeffrey Voas

Software Quality's Eight Greatest Myths

EDITOR: Jeffrey Voas • Reliable Software Technologies • jmvoas@rstcorp.com

For years now, I have been fortunate to be able to sample many of the software quality conferences and workshops held in the US and in Europe. Numerous IEEE and non-IEEE organizations deserve commendation for providing these excellent venues of learning and technical interchange.

However, I've started to feel that the software quality community has become too satisfied with the "state of the practice." Today's conferences offer few speakers with new ideas, and the enthusiasm that once surrounded our community seems absent. Those days when people would proudly hand out a business card with a title such as "software safety evangelist" (a real example) seem distant.

If those days are gone, the question becomes "why?" After all, none of the major challenges of creating quality software or assuring quality have been conquered. We face the same problems today that we did a decade ago, but now the urgency for solutions is greater.

Our present dilemma can be best explained by walking through the past decade's major software quality trends and their associated myths. By myths, I refer to the claims used to support the numerous software quality "silver bullets" that flourished and faded in the past decade. Although I could add lesser items to my list, I consider the following eight as the main culprits. (Because these fads are hard to compare fairly and quantitatively, they appear in no particular order.)

1. Process Improvement and Maturity

The myth associated with this trend states that measuring an organization's process maturity is equivalent to measuring the organization's software quality. This implies that simply building a more mature process will also produce mature (that is, higher-quality) software.

Process improvement and rating your professional maturity are laudable. However, it's fallible to assume that a software development organization that receives a high rating will produce software as good as the organization.[1] Unfortunately, this myth is endemic in the software and IT industries, and decades will pass before the myth is erased.

2. Formal Methods

A special case of the first myth, this myth states that formal methods are the "process improvement" answer to any and all security, reliability, and safety problems. Formal methods are simply rigorous development processes for mathematically demonstrating that software retains certain desirable properties. Formal methods aim to make everything precise to eliminate ambiguities, inconsistencies, and other logically incorrect behaviors that might exist in current system plans or definitions.

Although there is nothing wrong (and a lot right) with applying formal methods, their limitations have been well publicized[2] and their adoption highly limited. The key problems are that they are hard to implement, expensive, and not foolproof.

3. Languages and OOD

This myth indicates that by changing the language or design paradigm, problems that could not be resolved using existing languages or design strategies will go away. Les Hatton summarizes this problem nicely: he says that we have relied too heavily on the "language of the day" to solve problems that we could not solve using yesterday's language,

0740-7459/99/$10.00 © 1999 IEEE

and because our problems were never language-related, we naturally failed.[3]

Today's systems are creeping past all reasonable complexity thresholds. They are being implemented through complicated languages (that have so many features that few people understand the features fully or correctly). For example, the current craze—object-oriented languages—provides threads, polymorphism, inheritance, encapsulation, information hiding, and so on. These features cause serious problems when misused. You could argue that these complicated languages are making it harder to build quality software than if we used older languages that were less feature-rich.

Modern design paradigms that advocate abstraction (for example, information hiding and encapsulation) also make system-level testing more difficult to perform efficiently and adequately. Making systems harder to test will never engender higher-quality systems. Testing is already hard enough!

4. Metrics and Measurement

This myth says that numerical information about the development processes and code reveals whether the software is good or not. Before we can debunk this notion, we must be precise as to what it means for code to be "good."

For most people, good code computes the desired function in the desired manner (for example, correctly and with real-time constraints). Good is not a measure of how the code is structured or looks; good is an assertion that the semantics of the function that the code computes is the code's intended function.

Because structural metrics do not measure semantics, they cannot say whether the code is good (using this interpretation)—neither can process metrics. Unfortunately, when the field of metrics was young, some suggested that metrics could measure semantics, and when it was shown they could not, metrics received its still-tarnished reputation.

Another problem that has shadowed its reputation is that the collected metrics are often not fed back into the development process to improve the process. Interestingly, code metrics are probably better at assessing the quality of development processes than they are at assessing the code quality. Feed the results of measurement back into the organization and improve your processes!

Also, it is vital to recognize that metrics are indirect measures of immeasurable properties. For example, you cannot measure a program's testability and maintainability. But you can measure the number of lines of code. Because a program with one line of code will be more testable and maintainable than a one-million line program, the "lines of code" metric is one way to estimate how testable and maintainable code will be. The fundamental problem is that people still try to use metrics as absolute predictors (for example, if the cyclomatic complexity is greater than 10, go wild!). Generic claims cannot always be true for all systems. Instead, here's a good rule of thumb: *metrics give guidance; they are not absolute recipes for how to achieve quality*.

Metrics give guidance; they are not absolute recipes for how to achieve quality.

5. Software Standards

The fifth myth states that by following published standards, you can toss common sense on software development out the window—in other words, following a recipe (standard) blindly. (Admittedly, if you work in a regulated industry, you are forced to blindly follow the rules and regulations of that industry's regulator).

Software-engineering standards have grown explosively during the past 20 years. (Most of these standards have been process-oriented—a pitfall discussed in the first myth.) A fundamental challenge for organizations (for example, the ISO, IEEE, and IEC) that promulgate standards is to provide information on the value added by following their standards. For example, if you follow standard *A*, it will cost *B* in resources, and you will receive *C* benefits for having done so. If each standard carried with it a simple (average case) analysis such as this, standards would be easier to compare, contrast, and adopt.

In short, I have numerous concerns about standards, a few of which I list here:

♦ They lack timeliness (they are usually approved and published well after they were relevant);

♦ Some view them as impediments to competition instead of advocates for quality;

- They have unquantified value-added benefits;
- They do not specify how they are to be satisfied; and
- They have an unknown relationship to established "best practice" or related standards.

I am not advocating that standards should not be used, but one size does not fit all; you should consider your situation's specifics before opting for a standard.

6. Testing

This myth states that testing can get a project out of any bind. Just toss the code over the fence, and the testers will clean up all problems.

This is clearly foolish. Capers Jones' data says that the probability of this occurring is around 15%. That is, if a project is in serious trouble and the developers wait until the testing phase to address the problems, the project will be very difficult to salvage.

7. Computer-Aided Software Engineering

The next myth, probably my favorite, states that programming a specification through a schematic or pictorial language will produce higher-reliability code. CASE was the rage back in the early 1990s; it argued for using computers to generate text (code) from pictures. The thinking was that CASE could improve software quality because people make fewer errors when drawing pictures. Once a picture existed, a computer could take the picture and automatically generate code.

The intuition here is reasonable. But incorrect pictures will be translated into incorrect code. So the notion that pictorial representations of systems will result in higher-reliability code is suspect. Garbage in—garbage out.

8. Total Quality Management

The final myth says that if we meditate on quality long and hard enough, quality will sink into the product.

TQM is the perfect example of that myth in action. TQM is a manufacturing religion that argues that if you "eat, sleep, and dream" quality, then quality is more likely to permeate a product. And in manufacturing, this works.

Software is not manufacturing, however. It is an inventive process. Software is a creative expression using logic. The notion that a "quality zealotry" from the manufacturing industry would apply to software development was mistaken.

Stick to the Facts

I believe that the failure of these ideas as standalone silver bullets has made practitioners cynical about new ideas in software quality. The concern, then, is that future breakthroughs could be too quickly dismissed. Furthermore, if the community decides that the many problems related to software quality are unsolvable, research dollars and the next generation of bright researchers will not be available.

It is true that when the aforementioned eight ideas are taken in moderation and combination, they provide ways to produce good software. My hat is off to the thousands of scientists, researchers, graduate students, and practitioners that have labored for years to create these seminal ideas.

However, I must echo a warning: our research community must be more careful to not oversell its ideas to practitioners before the supporting evidence is in hand. That supporting evidence must fairly consider both the costs and limitations of adopting new technologies.

We all have a role to play here. Practitioners need to write articles and give talks at forums informing researchers of the real problems. And researchers must validate their claims on real systems (not toy systems) before marketing their ideas. If the research and practitioner communities unite, it can be a partnership where the sum far outweighs the parts. ❖

References

1. J. Voas, "Can Clean Pipes Produce Dirty Water?" *IEEE Software*, July/Aug. 1997, pp. 93–95.
2. S.L. Pfleeger and L. Hatton, "Investigating the Influence of Formal Methods," *Computer*, Vol. 30, No. 2, Feb. 1997, pp. 33–43.
3. L. Hatton, "Does OO Sync with How We Think?" *IEEE Software*, May/June 1998, pp. 46–54.

Jeffrey Voas is cofounder and chief scientist of Reliable Software Technologies. He received a PhD in computer science from the College of William and Mary. Contact him at jmvoas@rstcorp.com.