

This Decade's Eight Greatest Myths About Software Quality

J. Voas (jmvoas@rstcorp.com)

Reliable Software Technologies, Sterling VA USA

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 should be commended for providing excellent venues for learning and technical interchange that is need in the software quality community.

From several recent meetings, however, a general feeling has started to come over me that the software quality community has become too satisfied with the state-of-the-practice. Attendance at today's conferences provides few speakers with new ideas. Talks are usually re-castings of old ideas. The enthusiasm that once surrounded our community seems absent. Those days when people would proudly hand out a business card with a title like “software safety evangelist” (no joke—I really got one) seem distant.

If it is true, the question becomes “why?” After all, none of the major challenges of creating quality software or assuring quality have been conquered. The problems we face today are identical to those of a decade ago. Worse, the consequences of those problems are greater, thus increasing the urgency for solutions.

Reasons for concern abound. The Internet has shrunk the World. We face a future in which embedded processors may someday live in everything worth more than a few dollars. Microwaves may someday have IP addresses. If this is the world to come, can we afford to be satisfied with where we are today?

My position is that the research community is no longer providing solutions with the same vigor of years past. I am not saying that the practitioner community is not attempting to produce better software. I believe, however, that the research sector of our community has abandoned the search for revolutionary advances to the state-of-the-art in building and assuring software quality.

My reasoning for why this may have happened can best be explained by walking through the major milestones in software quality from the past decade. The past decade ushered in many proclaimed “software quality silver bullets.” As we all know, none panned out, leaving the overall software community cynical and quite tolerant to lesser quality software. And because none panned out, I will refer to the claims that were used to argue for these ideas as “myths.” So let's review eight key milestones and the myths associated with each.¹

1. Process Improvement/Maturity The first myth states that measuring an organization's process maturity is equivalent to measuring the organization's software quality.

¹While lesser milestones could be added to lengthen the list, I consider these the main culprits. Note also that these are in no particular ordering since they are hard to compare fairly and quantitatively.

This implies that by simply building a more mature process, the side-benefit will be mature (*i.e.*, higher quality) software.

I acknowledge that process improvement and rating one's professional maturity is laudable. The problem stems from assuming that if a software development organization receives a high rating, software produced by that organization will be as good as the organization supposedly is [3]. Unfortunately, this myth is endemic in the software and IT industries and it will be decades before the myth is erased.

- 2. Formal Methods** The formal methods myth is 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. The argument for formal methods is to make everything precise and formal in order to eliminate ambiguities, inconsistencies, and other logically incorrect behaviors that might exist in the current plans or definitions for the system.

In case you are wondering, formal methods are simply rigorous development processes for mathematically demonstrating that software retains certain desirable properties. While 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** The third myth states 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 [1] 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, and because our problems were never language related, we naturally failed.

Recognize that today's systems are creeping past all reasonable thresholds for complexity. They are being implemented via complicated languages (that have so many features that few people understand the features fully or correctly). For example, the current craze, object-oriented languages, provide threads, polymorphism, inheritance, encapsulation, information hiding, etc. These features cause serious problems when misused. In fact, it is fair to argue that these complicated languages are making it harder to build quality software than if we used older languages that were less feature rich.

And not only are the languages making it harder to produce quality systems. Modern design paradigms that advocate abstraction (*e.g.*, information hiding and encapsulation) make system-level testing more difficult to perform in an efficient and adequate manner. Making systems harder to test will never engender higher quality systems. Testing is already hard enough!

- 4. Metrics and Measurement** The fourth myth states 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 is code that computes the desired function and does so in the desired manner (*e.g.*, 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 function that the code was intended to compute.

Because structural metrics do not measure semantics, they cannot say whether the code is good (using this interpretation). Nor can process metrics. Unfortunately when the field of metrics was young, there were those that suggested that metrics could, and when it was shown they could not, metrics received the tarnished reputation that it still has today.

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

Also it vital to recognize that metrics are indirect measures of unmeasurable properties. For example, the testability and maintainability of a program cannot be measured. But the number of lines of code can be measured. Since 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 (*e.g.*, if the cyclomatic complexity is greater than 10, go wild!). Because generic claims such as this cannot always be true for all systems, measurement is viewed skeptically. Instead, a good rule of thumb is: *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, common sense about how to develop software can be tossed out the window. Instead, just follow a recipe (standard) blindly. (Admittedly, if you work in a regulated industry you are forced to blindly follow the rules and regulations of of the regulator of that industry).

The growth in software engineering standards has been explosive during the past 20 years. (Most of these standards have been process-oriented; this pitfall was discussed in the first myth.) A fundamental challenge for organizations (*e.g.*, ISO, IEEE, IEC, etc.) that promulgate standards is to provide information on the value-added by following their standards. For example, if standard *A* is followed, it will cost *B* in resources and you will receive *C* benefits for having done so. If each standard in existence 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 with standards, a few of which are their: (1) lack of timeliness (they are usually approved and published well after the point in time when they were relevant), (2) perceived ulterior motives as being impediments to competition as opposed to advocates for quality, (3) unquantified value-added, (4)

vagueness in how to satisfy, and (5) unknown relationship to established “best practice” or related standards. I am not advocating that standards should not be used, but I am arguing that one size does not fit all and thus you should consider the specifics of your situation before opting for a standard.

- 6. Testing** The sixth 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 Jone’s data says that the probability of this occurring is around 15%. That is to say that if a project is in serious trouble and the developers wait until the testing phase to address the problems, it is very unlikely that the project will be salvaged.

- 7. Computer Aided Software Engineering (CASE)** The seventh myth, and probably my favorite, states that if people are allowed a way to program a specification via a schematic/pictorial language, higher reliability code will result. Computer Aided Software Engineering (CASE) was the rage back in the early 1990s; it argued for using computers to generate text (code) from pictures. The thinking behind CASE was that software quality could be improved if people programmed in pictures (since 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 picture 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 (TQM)** And finally, the eighth myth states 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 be permeate into 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.

In summary, I believe that the failure of these ideas as “stand-alone” silver bullets has made practitioners cynical about new ideas in software quality. The concern then, is that future breakthroughs could be too quickly dismissed. Further, 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 was 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 over sell its ideas to practitioners before the supporting evidence is “in hand.” That supporting evidence must fairly consider the costs of adopting new technologies as well as the limitations.

In closing, we all have a role to play here. Practitioners need to write articles and speak 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] L. HATTON. Does OO Sync With How We Think? *IEEE Software*, 15(3), May 1998.
- [2] S. L. PFLEEGER AND L. HATTON. Investigation the Influence of Formal Methods. *IEEE Computer*, 30(2), February 1997.
- [3] J. VOAS. Can Clean Pipes Produce Dirty Water? *IEEE Software*, 14(4):93–95, July 1997.