Assuring Software Quality Assurance

Jeffrey Voas, Cigital

The term quality assurance, or QA, has a variety of interpretations. The most common is that it ensures that developers, testers, or independent auditors have performed some form of scrutiny on a system to validate that it will work as required. Software quality assurance is similar but applies to the code or noncode artifacts.

Software quality assurance

We can trace SQA’s roots back to the 1960s, when IBM used the term in the context of final product testing. SQA also has deep roots in the US Department of Defense, which created a family of military specification standards required of all software vendors seeking DoD contracts (the most famous of which is probably MIL-STD 2167A).

However, not all people buy into the belief that SQA is needed or is scientific—unfortunately, some consider it a stepchild of the software life-cycle process. Reasons for this are many, but one of the predominant criticisms against SQA is that it often occurs late in the life cycle, thus becoming a last-ditch attempt to bolt quality on at the end of development. Also, SQA’s return on investment is often calculable, convincing some that it is a waste of resources.

There has also been much debate about when to perform SQA, how much to apply, and how to measure its effectiveness. Even so, in the history of our profession, we’ve probably never needed SQA more than we do now.

Tolerance to defective software is much lower today than it was in the early 1990s—the blue screen of death was annoying to desktop users then, but it is virtually intolerable to enterprises now.

Defining quality

Quality has many interpretations in the IT community. First of all, is software that is reliable but not secure still high-quality code (and vice versa), or must the code be both reliable and secure to be labeled quality software?
Furthermore, do other desirable characteristics such as fault tolerance, performance, and availability play any role in what is meant by quality, and if so, how?

Second, it is important to differentiate what quality means in regulated versus nonregulated industries. Regulated industries (such as gaming, avionics, medical, transportation, and so forth) have prescribed methods that developers must apply, and regulatory approval requires independent certification that the developers used the required standards, development and testing methods, and tools. In some cases, regulatory agencies such as the US Federal Aviation Administration require the same level of assurance concerning the tools used during development as they do for the avionics code used for flight. However, in nonregulated industries, the degree of SQA performed is almost certainly market or client driven. If clients do not demand better quality, why perform more than minimal SQA?

Third, hundreds of standards exist that say they promote higher-quality software. What most of these standards desperately lack is something other than anecdotal evidence of their benefit and a credible body of independently accepted knowledge concerning their return on investment. After all, we know how to develop high-quality safety-critical software—but not cheaply or quickly.

Fourth, although there are taxonomies of software engineering standards and some comparisons, there is no clear way to compose standards. For example, if we build and test component A according to standard 1, and build and test component B according to standard 2, can we convincingly argue that the composition (of the components) inherits some minimal level of development and testing processes? That is, what is the union of standards 1 and 2? (Note that this has nothing to do with the quality of the composed products; that is a different composability problem.)

Fifth, we live in a world where assuring a fixed level of quality for a fixed software component might soon be impractical owing to the speed at which the world is changing. Product QA has traditionally assumed some bounds on the software’s environment and use. It is increasingly hard to swallow those assumptions. This then begs the question: What can we do to increase software’s quality-centric adaptability and intelligence? I’m talking about software that can increase or decrease its quality on the basis of how the software believes that its environment or mission has changed.

Finally, the value that any QA process or model adds is only as good as the process or model’s quality—in other words, the QA’s quality is key. This is often overlooked and is a key reason why many QA programs fail, thus giving QA a “black eye.”

This focus section of IEEE Software is a nice blend of known best practices (that need to be revisited from time to time) with new ideas that require additional validation and exploration. The articles discuss effective software process management, best practices, accelerated stress tests, software measurement programs, and automated QA for document understanding systems. I hope that each of you can find at least one “take home” message in each article. I welcome your feedback.

About the Author

Jeffrey Voas is a cofounder and chief scientist of Cigital. His research interests include composition strategies for COTS software, software product certification and warranties, and software quality measurement. He received his PhD in computer science from the College of William & Mary. He is a senior member of the IEEE, President of the IEEE Reliability Society, and an associate editor in chief on IEEE Software. Contact him at voas@cigital.com.