Software Quality Unpeeled

Dr. Jeffrey Voas
SAIC

The expression software quality *has many interpretations and meanings. In this article, I do not attempt to select any one in particular, but instead help the reader see the underlying considerations that underscore software quality. Software quality is a lot more than standards, metrics, models, testing, etc. This article digs into the mystique behind this elusive area.*

The term software quality has been one of the most overused, misused, and overloaded terms in software engineering. It is a generic term that suggests quality software but lacks general consensus on meaning. Attempts have been made to define it. The Institute for Electronics and Electrical Engineers (IEEE) Standard 729 defines it as:

> … totality of features of a software product that bears on its ability to satisfy given needs and … composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer. [1]

However, this attempt and others are few, and not precise. In fact the second edition of the Encyclopedia of Software Engineering [2] does not have it listed as an entry; the encyclopedia skips straight from "Software Productivity Consortium" to "software reading." And worse, books with software quality in the title never give a definition to it in their pages [3, 4].

If you review the past 20 years or so, you will find an abundance of other terms that have been employed as pseudo-synonyms for software quality. Examples include process improvement, software testing, quality management, the International Organization for Standardization 9001, software metrics, software reliability, quality modeling, configuration management, Capability Maturity Model® Integration, benchmarking, etc. In doing so, the term *software quality* has wound up representing a family of processes and ideas more than it represents *good enough* software. In short, software quality has become a culture and community more than a technical goal [5].

In this article, I will avoid the quicksand associated with trying to come up with a one-size-fits-all definition. Instead, I will expose how software quality is composed of various layers and, by peeling off different layers, it allows us to have a rational discussion between a typical software supplier and end user such that an agreement can be reached as to whether or not the software is good enough.

Certification

We will begin dissecting software quality by first looking at the multiple viewpoints behind the term *certification*. This will provide us with a look into our first layer.

"In some instances phantom users more heavily determine whether the software is fit for purpose than the traditional inputs. In short, it is environment that gives fit for purpose context."

The term is often used to refer to certifying people skills. For example, the American Society for Quality (ASQ) has a host of certifications that individuals can attain in order to demonstrate competence in certain fields, e.g., they can become an ASQ Certified Software Quality Engineer. An individual can also become certified in specific commercial software packages, e.g., a Microsoft Certified Software Engineer.

For the purposes here, I employ a different perspective that comes from three schools of thought. The first school deals with certifying that a certain set of development, testing, or other processes applied during the pre-release phases of the life-cycle were satisfied. In doing so, you certify that the *processes were followed and completed*. (Demonstrating that they were applied correctly is a trickier issue.) In the second school, you certify that the developed software *meets the functional requirements*; this can be accomplished via various types of testing or other analyses. For the third school, you can certify that the software itself is *fit for purpose*. This third school will be the most useful, and throughout this article, it will be considered software to be *good enough* if it is fit for purpose.

In this article, the term *purpose* suggests that two things are present: (1) executable software; and (2) an operating environment. An *environment* is a complex entity: It involves the set of inputs that the software will receive during execution, along with the probability that the events will occur [6]. This is referred to as the *operational profile* [6]. But it also involves the hardware that the software operates on, the operating system, available memory, disk space, drivers, and other background processes that are potentially competing for hardware resources, etc. These other factors are as much a part of the environment as are the traditional inputs; they have been termed *invisible* or *phantom* users.

In some instances phantom users more heavily determine whether the software is fit for purpose than the traditional inputs. In short, it is environment that gives fit for purpose context. By more completely defining and thus bounding the environment to include phantom users, we gain an advantage in that we can reduce the set of assumptions needed to predict whether the software is good enough. Understanding the distinction between traditional inputs and phantom users is one ingredient needed to argue that *fit for purpose* has been achieved.

Further, note that rarely will there be only one environment that software, and in particular general purpose software, will encounter during operation. That offers a key insight as to why general purpose software is not certified by independent laboratories; such laboratories could not be

Scenario	Meets Requirements	Satisfies Development Processes	Fit for Purpose
1	No	No	No
2	No	No	**Yes**
3	No	Yes	No
4	No	Yes	**Yes**
5	Yes	No	No
6	Yes	No	**Yes**
7	Yes	Yes	No
8	Yes	Yes	**Yes**

Table 1: *Views on Certification*

omnipotent and could not know all of the potential target environments [7].

By revisiting the three schools of thought on certification, we discover eight ways to visualize software quality (See Table 1). Let us look at a couple.

In Table 1, scenario 2 represents a system that did not meet the requirements and was not developed according to the specified development procedures, but miraculously, the end result was software that was usable in the field. While this seems implausible, it is possible. Scenario 7 is the opposite: a system that met the requirements and was developed according to specified development procedures but resulted in unusable software. Scenario 7 may seem like heresy to many in the community of software quality practitioners. It is not; it simply dispels the myth that requirements elicitation is far from a perfect science and that simply following common sense *do's* and *dont's* (as spelled out in a development process plan) guarantees good enough software [8].

Note that only four of these scenarios yield good enough software: 2, 4, 6, and 8. The other four provide a product that is not usable for its target environment and that brings us back to the discussion of why scoping the target environment as precisely as possible is an important piece of what software quality means.

In summary, *fit for purpose* is the nearest of the three certification schools of thought of the IEEE definitions for software quality. However, we cannot only rely on knowing the environment and expect to be justified in proclaiming we have achieved software quality. Let us explore other considerations.

Three High-Level Attributes of Fit for Purpose

Most readers would probably be comfortable with labeling software as being of good quality if the software could ensure that (1) it produces accurate and reliable output, (2) it produces the needed output in a timely manner, and (3) it produces the output in a secure and private manner. These three criteria simply state that you get the right results at the right time at the correct level of security. These are the next three considerations that cannot be ignored and must incorporate into what software quality means.

While each of these is intuitive, none is precise enough. The family of attributes referred to as the ilities is a good starting point to help increase that precision [9]. This family includes behavioral characteristics such as reliability, performance, safety, security, availability, fault-tolerance, etc. (These attributes are also sometimes termed non-functional requirements.) Other family members such as dependability, survivability, sustainability, testability, interoperability, and scalability each require some degree of one or more of the first six attributes. So, for example, to have a dependable system, some level of reliable and fault-tolerant behavior is necessary. To have a survivable system, some amount of fault tolerance and availability is required, and so on. However, to simplify this discussion towards our goal of understanding the term software quality, we will focus only on the first six: reliability, performance, safety, security, availability, and fault-tolerance.

"Ility" Oxymorons

Table 2 illustrates combinations of these six *ilities*. If we were to flesh this table out as we did in Table 1 we would have 64 rows; however, here we only show 14 combinations for brevity. (For the cells left empty, we are not considering the degree to which that attribute contributes to the quality of the software's behavior.)

Let us look at a few of these categories and determine what they represent:

- **Category 1.** Suggests that the software is reliable, has good performance, does not trigger unsafe events to occur (e.g., in a transportation control system), has appropriate levels of security built in, has good availability and thus does not suffer from frequent failures resulting in downtime, and is resilient to internal failures (i.e., fault tolerant). Is all of this possible in a single software system?
- **Category 2.** Suggests that the software offers reliable behavior, but suffers from the likelihood of producing outputs that send the system that the software feeds inputs into an unsafe mode. This would represent a safety-critical system where hazardous fail-

Table 2: *Ility Combinations*

Category	Reliability	Performance	Safety	Security	Availability	Fault-tolerance
1	Yes	Yes	Yes	Yes	Yes	Yes
2	Yes		No			
3	No		Yes			
4	No			Yes		
5	No				Yes	
6		No			Yes	
7		Yes	No			
8			Yes	No		
9			No	Yes		
10	Yes			No		
11	Yes			No	Yes	
12	Yes	No		Yes		
13		Yes		Yes		
14	No	No	No	No	No	No

ures are unacceptable; hazardous failures are categorized differently for such systems than failures that do not facilitate possible disastrous loss-of-life or loss-of-property consequences. (Note that software by itself is never unsafe; however software is often referred to as unsafe if it produces outputs to a system that put the system into an unsafe mode. Safety is a system property, not a software property.) A classic example of a reliable product that is unsafe is placing a functioning toaster into a bathtub of water with the cord still connected; the toaster is reliable, but it is not safe to go near.

- **Category 3.** Suggests that the software behaves so unreliably when executed that it cannot put the system into an unsafe mode. An example here would be that the software gets hung up in a loop and the safety functionality is never invoked.
- **Category 8.** Suggests safe but not secure software behavior. This is quite realistic for a safety-critical system with no security concerns. Note that the interesting aspect of this category is how safety and security are defined. Many people use these terms interchangeably, which is incorrect.
- **Category 11.** Suggests that the software behaves reliably and has good availability, but lacks adequate security precautions. Many systems suffer from this problem.
- **Category 12.** Suggests that the software behaves reliably, is extremely slow, but has adequate security. It makes one wonder if the system is so slow that it is effectively unusable, and thus secure, since it would take too long to break in.
- **Category 13.** Suggests high levels of security and high levels of performance. In certain situations that is plausible, however typically security kills performance and vice versa.
- **Category 14.** This is the easiest combination to achieve. Anyone can build a useless system.

The main point here is that the aforementioned high-level attributes (1) produce accurate and reliable output, (2) produce the needed output in a timely manner, and (3) produce the output in a secure and private manner are actually composed of the lower-level *ilities*. Another important point not to overlook is the fact that some of the *ilities* are not compatible with one another. An example of this can easily be found using fault tolerance and testability. A final important point is that some combinations of the *ilities* are simply counterintuitive, such as a system that is safe but unreliable.

One last thing to note: It is vital to get solid definitions for the *ilities* and to know which ones are quantifiable. For example, reliability and performance are quantifiable; security and safety are not. This makes it far easier to make statements such as *we have very high reliability but an unknown level of security.*

The Shall Nots

There is yet another layer in the notion of fit for use that deals with *negative* functional requirements. Think of a negative requirement as "the software shall *not* do X," as opposed to a functional requirement stating that "the software *shall* do X."

"For certain types of systems, particularly safety-critical systems, enumerating negative requirements is a necessity. And for software requiring security capabilities, security rules and policies are its equivalent to negative requirements."

Negative requirements are far more difficult to elicit than regular requirements. Why? Because humans are not programmed to anticipate and enumerate all of the bad circumstances that can pop up and that we need protection against; we are instead programmed to think about the *good* things we want the software to do.

For certain types of systems, particularly safety-critical systems, enumerating negative requirements is a necessity. And for software requiring security capabilities, security rules and policies are its equivalent to negative requirements. For example, a negative security requirement could be that the software shall never open access to a particular channel unless it can be guaranteed that the information passing through the channel is moving between trusted entities. The difficulty in defining *shall not's* for security is that we cannot imagine all of the different forms of malicious attacks that are being invented on-the-fly and if we cannot imagine those attacks, we likely will not prevent them.

Before leaving the topic of negative functional requirements, it is worth mentioning an interesting relationship between them and the environment. So far, we have only mentioned traditional inputs and phantom users as players in the environment. Traditional inputs are those that the software expects to receive during operation. But there are two other types of inputs worth mentioning: *malicious illegal* and *non-malicious illegal*. A malicious illegal input is one that someone deliberately feeds into the software to attack a system, and a non-malicious illegal input is simply an input that the system designers do not want the software to accept but has no malicious intent. In both cases, filtering on either type of input can be useful to ensure that certain inputs do not become a part of the environment and in doing so ensure that negative functional requirements are enforced.

Time

The next layer in our quest for software quality is *time*. Software has fixed longevity; it can be expanded, as we learned from Y2K, but not indefinitely.

One of the easiest ways to explain why time fits here is to look at the situation where a software package operates correctly on Monday but does not operate correctly on Tuesday. Further, the software package was not modified between these days. (This is the classic problem that quickly carves down the number of freshman computer science majors.) Why has this problem occurred?

It all goes back to the importance of environment in the understanding of software quality. Earlier we defined the environment as inputs with probabilities of selection, hardware configurations, access to memory, operating systems, attached databases, and whether other background processes were overindulging in resources, etc.

But what is not mentioned was *calendar time*. Environment is also a function of time. As time moves forward, other pieces of the environment change. And so while all effort and expense can be levied toward what we perceive is evidence supporting the claim that we have good enough software, we need to recognize that even if we do have good enough software, it may be only for a short window of time. Thus, software quality is *time-dependent*, a bitter pill to swallow.

Cost

We cannot end this article without mentioning *cost*. The costs associated with software quality are exasperated by the un-family like behaviors of various *ilities*. Not only are there technical trade-offs discovered when trying to increase the degree to which one *ilitiy* exists only to find that another is automatically decreased, but there is the financial trade-off quagmire concerning how to allocate financial resources between distinct *ilities*. If you overspend on one, there may not be enough funds for another. And, as if the technical considerations are not hard enough when trying to define software quality, the financial considerations come aboard, making the problem worse.

Conclusion

In this article, a set of layers for what software quality means has been unpeeled. I have argued that a more useful perspective for what software quality represents starts from the notion of the software being *fit for purpose*, which requires:

1. Understanding the relationship between the *functional requirements* and the *environment*.
2. Understanding the three high-level attributes of software quality: the software: (a) produces *accurate and reliable* output, (b) produces the needed output in a *timely* manner, and (c) produces the output in a *secure and private* manner.
3. Understanding that the i*lities* afford the potential to have varying degrees of the high-level attributes.
4. Understanding that the *shall-not* functional requirements are often of equal importance to the functional requirements.
5. Understanding that there is a *temporal* component to software quality; software quality is not static or stagnant,
6. Understanding that the *ilities* offer technical incompatibilities as well as financial incompatibilities.
7. Understanding that the environment contains many more parameters such as the phantom users than is typically considered.

Thus, software quality, when viewed with these different considerations, becomes a far more interesting topic, and one that will continue to perplex us for decades to come.◆

References

1. IEEE. Standard Glossary of Software Engineering Terminology IEEE. American National Standards Institute/IEEE Standard 729-1983: 1983.
2. Marciniak, J., ed. Encyclopedia of Software Engineering. Second Edition. Wiley Inter-Science: 2002.
3. Wieczorek, Martin, and Dirk Meyerhoff, editors. Software Quality: State of the Art in Management, Testing, and Tools. Springer. New York: 2001.
4. Gao, J.Z., H.S. Jacob Tsao, and Y. Wu. Testing and Quality Assurance for Component-Based Software. Artech House. Norwood, MA: 2003.
5. Whittacker, J., and J. Voas "50 Years of Software: Key Principles for Quality." IEEE IT Professional. 4(6): 28-35, Nov. 2002.
6. Musa, John D., Anthony Iannino, and Kazuhiiro Okumoto. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, NY, 1987.
7. Voas, J. "Software Certification Laboratories?" CrossTalk Apr. 1998.
8. Voas, J. "Can Clean Pipes Produce Dirty Water?" IEEE Software July 1997.
9. Voas, J. "Software's Secret Sauce: The 'Ilities." IEEE Software Nov. 2004.

About the Author

Jeffrey Voas, Ph.D., is currently director of systems assurance at Science Applications International Corporation (SAIC). He was the president of the IEEE Reliability Society from 2003-2005, and currently serves on the Board of Governors of the IEEE Computer Society. Voas has published numerous articles over the past 20 years, and is best known for his work in software testing, reliability, and metrics. He has a doctorate in computer science from the College of William & Mary.

SAIC
200 12th ST South
STE 1500
Arlington, VA 22202
Phone: (703) 414-3842
Fax: (703) 414-8250
E-mail: j.voas@ieee.org