Definition of Software Quality – What is Software Quality?

Software quality is an abstract concept. Its presence can be difficult to define, but its absence can be easy to see instantly. Thus, in the quest for improving software quality, we must first understand the software quality definition. Wikipedia describes software quality as follows:

"In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance). It is often described as the 'fitness for purpose' of a piece of software."

There are many variations to the definition of software quality, but if you examine the definition above, "fitness for purpose" questions whether or not the software fulfils its purpose, or "Does it do what it's supposed to do?" Those are the characteristics that we see as end users. Quality of design and quality of conformance to that design are related to internal aspects of the software, some of which we may see, like the user interface's navigation, placement of controls and so on. Others, we would not normally see, like code architecture, code quality and code security.

Quality in Software Engineering

In broader terms, the software quality definition of "fitness for purpose" refers to the satisfaction of requirements. But what are requirements? Requirements, also called user stories in today's Agile terms, can be categorized as functional and non-functional. Functional requirements refer to specific functions that the software should be able to perform. For example, the ability to print on an HP Inkjet 2330 printer is a functional requirement. However, just because the software has a certain function or a user can complete a task using the software, does not mean that the software is of good quality. There are probably many instances where you've used software and it did what it was supposed to do, such as find you a flight or make a hotel reservation, but you thought it was poor quality. This is because of "how" the function was implemented. The dissatisfaction with "how" represents the non-functional requirements not being met.

For this purpose the International Organization for Standardization (ISO) developed ISO 25010 [1] as a model for specifying non-functional requirements. The model shown below illustrates the categorization of non-functional requirements.

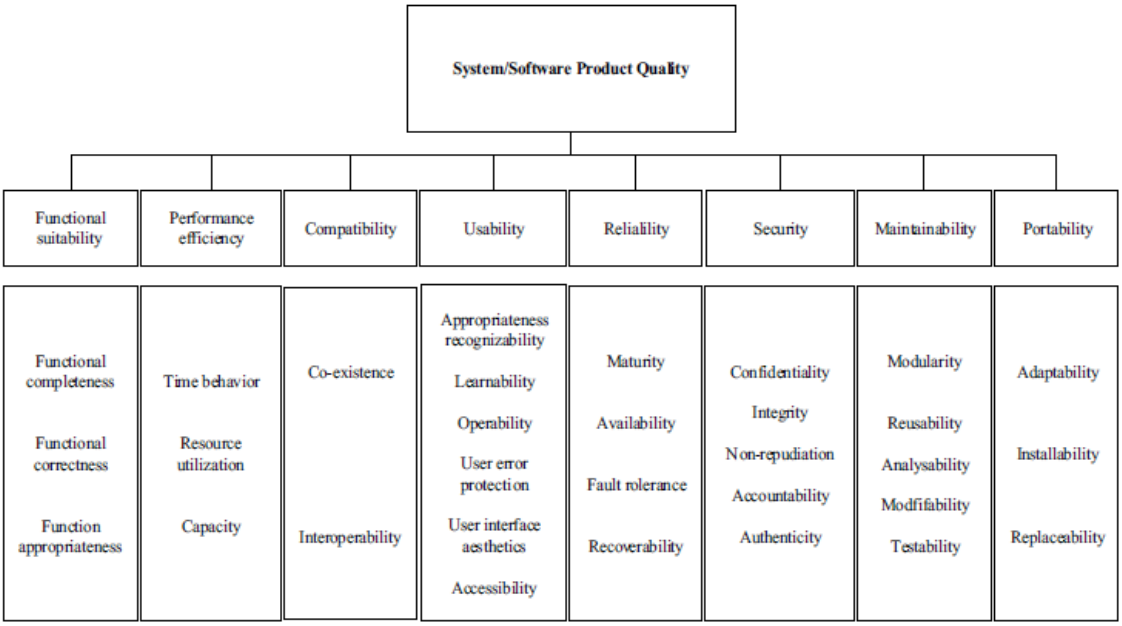


Figure 1- [1] ISO/IEC CD 25010 Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Quality model and guide, 2011.

At first glance, you may think that the left most characteristic, Functional Suitability, is equivalent to a functional requirement, but it's not. Sub-characteristics functional completeness, functional correctness and functional appropriateness apply to functions that have been implemented and are characteristics of those functions. For instance, functional completeness is defined as the degree to which the set of functions covers all the specified tasks and user objectives. So, "Print from HP Inkjet 2330 Printer" could have been implemented from a functional requirement point of view. But how was it implemented? Was it complete for all options? Did it have double-sided printing? If not, then it might not be good quality from a functional completeness point of view.

Why Non-Functional Quality Components Are Important

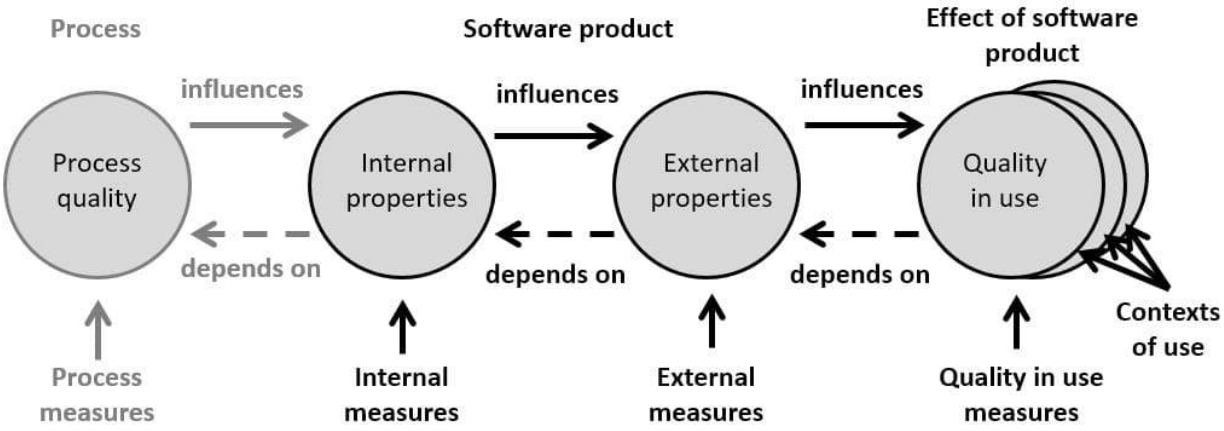
Satisfying non-functional requirements such as performance, ease of use and learnability first requires specifying and defining. Only then can they be satisfied, and satisfying them can be even more difficult than satisfying functional requirements. So what does this mean for you? Let's examine the following non-functional characteristics using the same printing example:

- Functional Suitability (functional appropriateness) – Does the function facilitate the completion of the user's task(s) and objectives? If the user doesn't want to print on that printer or wants to print a PDF but isn't given those options, then maybe not.
- Performance Efficiency (time behaviour) – Does the printer function respond within three seconds?

- Compatibility (interoperability) – Can the user print over a variety of networks and printers and on computers with different operating systems (Windows and Mac)?
- Usability (learnability) – Can the user figure out how to print or will it take a rocket scientist?
- Reliability (recoverability) – When the printer is unplugged in the middle of printing a task, is the user notified?
- Security (non-repudiation) – Is there a record that the printer printed the file successfully?
- Maintainability (testability) – Can test criteria be specified for the print function?
- Portability (adaptability) – Can the software automatically adapt to new printer models, or an update in printer driver software? Can the print function provide shortcuts for highly sophisticated users?

Now that we have an understanding of non-functional requirements, let's examine the quality lifecycle in the diagram below. Looking at the three circles below, internal quality represents quality that you wouldn't see and is measured by internal properties such as code quality. External quality represents what we have discussed above in the non-functional quality model and is typically measured by actual execution of the code and examination of software behaviour.

Software Quality Lifecycle



Here is yet another model: quality lifecycle. This model may be closer to how you perceive quality. That is, quality from an end user viewpoint when they are actually using the software in real life and not in a lab. That's what "contexts of use" means. For example, software quality testing can happen on a test server and have perfect test results, but users in their environment may have different results. They may not be able to find a button or control as easily as a tester would, or maybe they want to print directly from a place in the application that you never thought of.

Also important to note in Figure 2 is the use of the arrows and dotted lines. You'll notice that there is a relationship between internal quality, external quality and quality in use. Namely, internal quality has an influence on, but not a direct correlation with, external quality and that external quality depends on internal quality. Let's think about this. This means that you can have great code quality

(internal quality) and still have poor external quality (software behaviour). This makes sense in reverse too. The software might work okay, but the internal quality could be terrible.

Using the same application of arrows and dotted lines as in the quality lifecycle diagram (Figure 2), you can see that product quality depends on process quality and process quality influences product quality. The reason for using the magic word “influences” is that even if you have fantastic, repeatable and improving software quality testing processes, it doesn’t mean that the processes are geared toward solving the right problem or even building the right product. So, while you may have excellent processes, you might still have mediocre products. Conversely, you could have great product quality, but poor processes. Consider the history of CMMI, developed by the Software Engineering Institute at Carnegie Melon, with funding from the Department of Defence. The DOD were tired of getting poor quality product. Their hypothesis was that better processes (and hence requiring CMMI certification) would result in better product. However, this didn’t always happen.

Why Setting Quality Requirements Isn’t Easy

Born over a decade later, Agile attempted to solve this problem, focusing on the known fact that customers change their minds not on purpose, but because their understanding of what they want can sometimes only change if they see it in workable form.

Many say that conformance to requirements is the primary definition of quality. But as you saw from this discussion, specifying “requirements” is not an easy task. Everyone has different interpretations of text and drawings based on their context of understanding. That’s why one of the primary tenets of Agile is to deal with requirements that are changing and are sometimes incomplete or unspecified until seen. A simple “that’s what I wanted!” or “that’s not exactly what I meant” in response to a demonstration of working software beats pages and pages of documentation.

In summary, defining what software quality means to you and your software, and then developing a means to measure and evaluate, can help you improve. However, if you don’t know what you are measuring, you can’t evaluate, much less improve!