Quality Definitions of Qualities

Spencer Smith
McMaster University, Canada
smiths@mcmaster.ca

Jacques Carette
McMaster University, Canada
carette@mcmaster.ca

Olu Owojaiye
McMaster University, Canada

Peter Michalski
McMaster University, Canada

Ao Dong
McMaster University, Canada

Abstract

...

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Author: Please fill in \keywords macro

Contents

1 Introduction **2**

2 Qualities of Software Products, Artifacts and Processes **2**
2.1 Installability . . . 2
2.2 Correctness . . . 2
2.3 Verifiability . . . 2
2.4 Validatability . . . 3
2.5 Reliability . . . 3
2.6 Robustness . . . 3
2.7 Performance . . . 3
2.8 Usability . . . 3
2.9 Maintainability . . . 4
2.10 Reusability . . . 4
2.11 Portability . . . 4
2.12 Understandability . . . 4
2.13 Interoperability . . . 5
2.14 Visibility/Transparency . . . 5
2.15 Reproducibility . . . 5
2.16 Productivity . . . 5
2.17 Sustainability . . . 6

3 Desirable Qualities of Good Specifications **7**
3.1 Completeness . . . 7
3.2 Consistency . . . 8
3.3 Modifiability . . . 8
3.4 Traceability . . . 8
3.5 Unambiguity . . . 9

3.6 Verifiability 9
3.7 Abstract 9

1 Introduction

Purpose and scope of the document. [Needs to be filled in. Should reference the overal research proposal, and the "state of the practice" exercise in particular. —SS]

The presentation is divided into two main sections: i) qualities that apply to software products, software artifacts and software development processes, and ii) qualities that are considered important for good specifications. The specification could be a specification of requirements, design or a test plan.

2 Qualities of Software Products, Artifacts and Processes

To assess the current state of software development, and to understand how future changes impact software development, we need a clear definition of what we mean by quality. The concept of quality is decomposed into a set of separate components that together make up "quality". Unfortunately, these are called *qualities*. These are associated to the software product, the software artifacts (documentation, test cases, etc) and to the software development process itself, and combinations thereof.

Our analysis is centred around a set of software qualities. Quality is not considered as a single measure, but a collection of different qualities, often called "ilities." These qualities highlight the desirable nonfunctional properties for software artifacts, which include both documentation and code. Some qualities, such as visibility and productivity, apply to the process used for developing the software. The following list of qualities is based on Ghezzi et al. (2003). To the list from Ghezzi et al. (2003), we have added three qualities important for SC: installability, reproducibility and sustainability.

2.1 Installability

A measure of the ease of installation.
Installability is related to the effort required to install a software in a specified environment. In other words, its capability to be installed in a specified environment.

2.2 Correctness

Software is correct if it matches its specification. There is no direct tool or method for measuring correctness. One way of building confidence in correctness is by reviewing to ensure that each requirement stated is one that the stakeholders and experts desire. By maintaining traceability, consistency and unambiguity, we can reduce the occurrence of errors and make the goal of reviewing for correctness easier.
The extent to which a software's specification, design and implementation is free from errors is its correctness. The quality of a software's operation is dependent on the degree of correctness Berander et al. (2005). Correctness and reliability are said to have dependencies, such that if a system exhibits a high degree of correctness then it tends to be reliable.

2.3 Verifiability

Verifiability involves "solving the equations right" (Roache, 1998, p. 23); it benefits from rational documentation that systematically shows, with explicit traceability, how the governing

equations are transformed into code.

2.4 Validatability

Validatability means "solving the right equations" (Roache, 1998, p. 23). Validatability is improved by a rational process via clear documentation of the theory and assumptions, along with an explicit statement of the systematic steps required for experimental validation.

2.5 Reliability

Reliability is a critical quality for scientific software, since the results of computations are meaningless, if they are not dependable. Reliability is closely tied to verifiability, since the key quality to verify is reliability, while the act of verification itself improves reliability.
Reliability models can be used to predict reliability of a software product. For example measuring Mean Time to Fail (MTTF) can be a good measure of reliability Berander et al. (2005)

2.6 Robustness

The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions IEEE Std 610.12-1990. The quality can be further informally refined as the ability of a software to keep an acceptable behavior, expressed in terms of robustness requirements, in spite of exceptional or unforeseen execution conditions (such as the unavailability of system resources, communication failures, invalid or stressful inputs, etc.) Fernandez et al. (2005).

2.7 Performance

The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage IEEE Std 610.12-1990. Performance considerations can make certification challenging, since QA becomes more difficult for more complex code. However, as Roache (Roache, 1998, p. 355) points out, using simpler algorithms and reducing the number of options in general purpose code, is not always a practical option.

2.8 Usability

ISO defines usability as

> The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.

Nielsen and (separately) Schneidermann have defined usability as part of usefulness and is composed of

- Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the design?
- Efficiency: Once users have learned the design, how quickly can they perform tasks?
- Memorability: When users return to the design after a period of not using it, how easily can they re-establish proficiency?
- Errors: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- Satisfaction: How pleasant is it to use the design?

In that context, it makes sense to separate *usefulness* into *usability* (purely an interface concern) and *utility* (in the economics sense of the word).

There are two ISO standards covering this, namely ISO/TR 16982:202 and ISO 9241.

The Interaction Design Foundation https://www.interaction-design.org/literature/topics/usability further lists the following desirable outcomes:

1. It should be easy for the user to become familiar with and competent in using the user interface during the first contact with the website. For example, if a travel agent's website is a well-designed one, the user should be able to move through the sequence of actions to book a ticket quickly.
2. It should be easy for users to achieve their objective through using the website. If a user has the goal of booking a flight, a good design will guide him/her through the easiest process to purchase that ticket.
3. It should be easy to recall the user interface and how to use it on subsequent visits. So, a good design on the travel agent's site means the user should learn from the first time and book a second ticket just as easily.

One core reference, for definitions and metrics, is Bevan Bevan (1995).

2.9 Maintainability

The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment IEEE Std 610.12-1990. ISO/IEC 25010 refers to maintainability as the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers ISO/IEC 25010:2011. Maintainability is necessary in scientific software, since change, through iteration, experimentation and exploration, is inevitable. Models of physical phenomena and numerical techniques necessarily evolve over time Carver et al. (2007); Segal and Morris (2008). Proper documentation, designed with change in mind, can greatly assist with change management. QA activities need to take the need for creativity into account, while not smothering it (Roache, 1998, p. 352).

2.10 Reusability

The degree to which a software module or other work product can be used in more than one software system IEEE Std 610.12-1990. Reusability provides support for the quality of reliability, since reliability is improved by reusing trusted components Dubois (2005). (Care must still be taken with reusing trusted components, since blind reuse in a new context can lead to errors, as dramatically shown in the Ariane 5 disaster (Oliveira and Stewart, 2006, p. 37–38).) The odds of reuse are improved when it is considered right from the start.

2.11 Portability

The ease with which a system or component can be transferred from one hardware or software environment to another IEEE Std 610.12-1990. An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment Mooney (1990).

2.12 Understandability

Understandability is artifact-dependent. What it means for a user-interface (graphical or otherwise) to be understandable is wildly different than what it means for the code, and

even the user documentation.

The literature here is thin and scattered. More work will need to be done to find something useful.

Interestingly, the business literature seems to have taken more care to define this. Here we encounter

> Understandability is the concept that X should be presented so that a reader can easily comprehend it.

At least this brings in the idea that the *reader* is actively involved, and indirectly that the reader's knowledge may be relevant, as well as the "clarity of exposition" of X.

Section 11.2 of Adams et al. (2015) does have a full definition.

2.13 Interoperability

> Interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged IEEE (1991). The degree to which two or more systems, products or components can exchange information and use the information that has been exchanged ISO/IEC 25010:2011. The capability to communicate, execute programs, and transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units ISO/IEC/IEEE24765:2010. Interoperability is a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, in either implementation or access, without any restrictions AFUL.

2.14 Visibility/Transparency

[I found little contents related to visibility or transparency. The following definition is the only one I could find, but may be irrelevant to this project. —AD]

> 1. the degree to which a transaction can access object state concurrently with other transactions. 2. the specification, for a property, of "who can see it?" ISO/IEC/IEEE24765:2010

2.15 Reproducibility

Reproducibility is a required component of the scientific method Davison (2012). Although QA has, "a bad name among creative scientists and engineers" (Roache, 1998, p. 352), the community need to recognize that participating in QA management also improves reproducibility. Reproducibility, like QA, benefits from a consistent and repeatable computing environment, version control and separating code from configuration/parameters Davison (2012).

2.16 Productivity

> The best definition of the productivity of a process is
>
> $$Productivity = \frac{Outputs\ produced\ by\ the\ process}{Inputs\ consumed\ by\ the\ process}$$
>
> Defining inputs. For the software process, providing a meaningful definition of inputs is a nontrivial but generally workable problem. Inputs to the software process generally

comprise labor, computers, supplies, and other support facilities and equipment. Defining outputs. The big problem in defining software productivity is defining outputs. Here we find a paradox. Most sources say that defining delivered source instructions (DSI) or lines of code as the output of the software process is totally inadequate, and they argue that there are a number of deficiencies in using DSI. However, most organizations doing practical productivity measurement still use DSI as their primary metric Boehm (1987).

Productivity is the amount of output (what is produced) per unit of input used.If we can measure the size of the software product and the effort required to develop the product, we have:

$$productivity = size/effort \tag{1}$$

Equation (1) assumes that size is the output of the software production process and effort is the input to the process. This can be contrasted with the viewpoint of software cost models where we use size as an independent variable (i.e., an input) to predict effort which is treated as an output. Equation (1) is simple to operationalize if we have a single dominant size measure, for example, product size measured in lines of code Kitchenham and Mendes (2004).

2.17 Sustainability

One of the original definitions of sustainability (for systems, not software specific), and still often quoted, is:

The ability to meet the needs of the present without compromising the ability of future generations to meet their own needs Brundtland (1987).

This is the definition used by International Institute for Sustainable Development (2019).
To make it more useful, this definition is often split into three dimensions: social, economic and environmental. [cite UN paper [9] in Penzenstadler and Femmer (2013) —SS] To this list Penzenstadler and Henning (2013) have added technical sustainability Penzenstadler and Femmer (2013). Where technical sustainability for software is defined as:

Technical sustainability has the central objective of long-time usage of systems and their adequate evolution with changing surrounding conditions and respective requirements Penzenstadler and Femmer (2013).

The fourth dimension of technical sustainability is also added by Wolfram et al. (2017). Technical sustainability is the focus on the thesis by Hygerth (2016).

Sustainable development is a mindset (principles) and an accompanying set of practices that enable a team to achieve and maintain an optimal development pace indefinitely Tate (2005).

Parnas discusses as software aging Parnas (1994).
SCS specific definitions:

The concept of sustainability is based on three pillars: the ecological, the economical and the social. This means that for a software to be sustainable, we must take all of

its effects – direct and indirect – on the environment, the economy and the society into account. In addition, the entire life cycle of a software has to be considered: from planning and conception to programming, distribution, installation, usage and disposal Heine (2017).

The capacity of the software to endure. In other words, sustainability means that the software will continue to be available in the future, on new platforms, meeting new needs Katz (2016).

Definition from Neil Chue Hong:

Sustainable software is software which is: – Easy to evolve and maintain – Fulfils its intent over time – Survives uncertainty – Supports relevant concerns (Political, Economic, Social, Technical, Legal, Environmental) Katz (2016).

Paper critical of a lack of a definition Venters et al. (2014).
Sounds like definition of maintainability.
Find paper that combines nonfunctional qualities into sustainability.
Sustainability depends on the software artifacts AND the software team AND the development process.

3 Desirable Qualities of Good Specifications

To achieve the qualities listed in Section 2, the documentation should achieve the qualities listed in this section. All but the final quality listed (abstraction), are adapted from the IEEE recommended practise for producing good software requirements IEEE (1998). Abstraction means only revealing relevant details, which in a requirements document means stating what is to be achieved, but remaining silent on how it is to be achieved. Abstraction is an important software development principle for dealing with complexity (Ghezzi et al., 2003, p. 40). Correctness was in the above list, so it is not repeated here. Smith and Koothoor (2016) present further details on the qualities of documentation for SCS.

3.1 Completeness

Documentation is said to be complete when all the requirements of the software are detailed. That is, each goal, functionality, attribute, design constraint, value, data, model, symbol, term (with its unit of measurement if applicable), abbreviation, acronym, assumption and performance requirement of the software is defined. The software's response to all classes of inputs, both valid and invalid and for both desired and undesired events, also needs to be specified.

A specification is complete to the extent that all of its parts are present and each part is fully developed. A software specification must exhibit several properties to assure its completeness:

- No TBDs. TBDs are places in the specification where decisions have been postponed by writing "To be Determined" or "TBD."
- No nonexistent references. These are references in the specification to functions, inputs, or outputs (including databases) not defined in the specification.
- No missing specification items. These are items that should be present as part of the standard format of the specification, but are not present.

- No missing functions. These are functions that should be part of the software product but are not called for in the specification.
- No missing products. These are products that should be part of the delivered software but are not called for in the specification. Boehm (1984).

3.2 Consistency

Documentation is said to be consistent when no subset of individual statements are in conflict with each other. That is, a specification of an item made at one place in the document should not contradict the specification of the same item at another location.

A specification is consistent to the extent that its provisions do not conflict with each other or with governing specifications and objectives. Specifications require consistency in several ways.

- Internal consistency. Items within the specification do not conflict with each other.
- External consistency. Items in the specification do not conflict with external specifications or entities.
- Traceability. Items in the specification have clear antecedents in earlier specifications or statements of system objectives Boehm (1984).

Consistency requires that no two or more requirements in a specification contradict each other. It is also often regarded as the case where words and terms have the same meaning throughout the requirements specifications (consistent use of terminology). These two views of consistency imply that mutually exclusive statements and clashes in terminology should be avoided Zowghi and Gervasi (2003). Consistency: 1. the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component 2. software attributes that provide uniform design and implementation techniques and notations ISO/IEC/IEEE24765:2010

3.3 Modifiability

Here we do seem to have a simple, if somewhat uninformative, definition:

▶ **Definition 1.** Modifiability is the degree of ease at which changes can be made to a system, and the flexibility with which the system adapts to such changes.

IEEE Standard 610 seems to speak about this. (which is superceded?)

3.4 Traceability

Here the Wikipedia page `https://en.wikipedia.org/wiki/Traceability` is actually rather informative, especially as it also lists how this concept is used in other domains. A generic definition that is still quite useful is

▶ **Definition 2.** The capability (and implementation) of keeping track of a given set or type of information to a given degree, or the ability to chronologically interrelate uniquely identifiable entities in a way that is verifiable.

By specializing the above to software artifacts, "interrelate" to "why is this here" (for forward tracing from requirements), this does indeed give what is meant in SE.

Various standards (DO178C, ISO 26262, and IEC61508) explicitly mention it.

24765-2017 - ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary has a full definition, namely

1. the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another;
2. the identification and documentation of derivation paths (upward) and allocation or flowdown paths (downward) of work products in the work product hierarchy;
3. the degree to which each element in a software development product establishes its reason for existing; and discernible association among two or more logical entities, such as requirements, system elements, verifications, or tasks.

3.5 Unambiguity

Documentation is said to be unambiguous only when every requirement's specification has a unique interpretation. The documentation should be unambiguous to all audiences, including developers, users and reviewers.

3.6 Verifiability

Every requirement in the documentation must be the one fulfilled by the implemented software. Therefore all the requirements should be clear, unambiguous and testable, so that a person or a machine can verify whether the software product meets the requirements.

3.7 Abstract

Documented requirements are said to be abstract if they state what the software must do and the properties it must possess, but do not speak about how these are to be achieved. For example, a requirement can specify that an Ordinary Differential Equation (ODE) must be solved, but it should not mention that Euler's method should be used to solve the ODE. How to accomplish the requirement is a design decision, which is documented during the design phase.

References

Kevin MacG Adams et al. *Nonfunctional requirements in systems analysis and design*, volume 28. Springer, 2015.

AFUL. Definition of interoperability. http://interoperability-definition.info/en/.

Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, et al. Software quality attributes and trade-offs. *Blekinge Institute of Technology*, 2005.

Nigel Bevan. Measuring usability as quality of use. *Software Quality Journal*, 4(2):115–130, 1995.

B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, Jan 1984. doi: 10.1109/MS.1984.233702.

Barry W. Boehm. Improving software productivity. *Computer*, pages 43–47, 1987.

G. H. Brundtland. *Our Common Future*. Oxford University Press, 1987. URL https://EconPapers.repec.org/RePEc:oxp:obooks:9780192820808.

Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.77.

A. P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, July-Aug 2012.

P. F. Dubois. Maintaining correctness in scientific programs. *Computing in Science & Engineering*, 7(3):80–85, May-June 2005. doi: 10.1109/MCSE.2005.54.

Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In *IFIP International Conference on Testing of Communicating Systems*, pages 333–348. Springer, 2005.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Robert Heine. What is sustainable software? https://www.energypedia-consult.com/en/blog/robert-heine/what-sustainable-software, July 2017.

Henrik Hygerth. Sustainable software engineering : An investigation into the technical sustainability dimension. Master's thesis, KTH, Sustainability and Industrial Dynamics, 2016.

IEEE. Ieee standard computer dictionary: A compilation of ieee standard computer glossaries. *IEEE Std 610*, pages 1–217, Jan 1991. doi: 10.1109/IEEESTD.1991.106963.

IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, October 1998. doi: 10.1109/IEEESTD.1998.88286.

IEEE Std 610.12-1990. Ieee standard glossary of software engineering terminology. Standard, IEEE, 1991.

IISD International Institute for Sustainable Development. Sustainable development. https://www.iisd.org/topic/sustainable-development, October 2019.

ISO/IEC 25010:2011. Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models. Standard, International Organization for Standardization, Mar 2011.

ISO/IEC/IEEE24765:2010. Systems and software engineering - vocabulary. Standard, International Organization for Standardization, Dec 2010.

Daniel Katz. Defining software sustainability. https://danielskatzblog.wordpress.com/2016/09/13/defining-software-sustainability/, September 2016.

B. Kitchenham and E. Mendes. Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering*, 30(12):1023–1035, Dec 2004. doi: 10.1109/TSE.2004.104.

James D. Mooney. Strategies for supporting application portability. *Computer*, 23(11):59–70, 1990.

Suely Oliveira and David E. Stewart. *Writing Scientific Software: A Guide to Good Style*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521858968.

D. L. Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287, May 1994. doi: 10.1109/ICSE.1994.296790.

Birgit Penzenstadler and Henning Femmer. A generic model for sustainability with process- and product-specific instances. In *Proceedings of the 2013 Workshop on Green in/by Software Engineering*, GIBSE '13, pages 3–8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1866-2. doi: 10.1145/2451605.2451609. URL http://doi.acm.org/10.1145/2451605.2451609.

Patrick J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

Judith Segal and Chris Morris. Developing scientific software. *IEEE Software*, 25(4):18–20, July/August 2008.

W. Spencer Smith and Nirmitha Koothoor. A document-driven method for certifying scientific computing software for use in nuclear safety analysis. *Nuclear Engineering and Technology*, 48(2):404–418, April 2016. ISSN 1738-5733. doi: http://dx.doi.org/10.1016/j.net.2015.11.008. URL http://www.sciencedirect.com/science/article/pii/S1738573315002582.

Kevin Tate. *Sustainable Software Development: An Agile Perspective*. Addison-Wesley Professional, 2005. ISBN 0321286081.

CC Venters, C Jay, LMS Lau, MK Griffiths, V Holmes, RR Ward, J Austin, CE Dibsdale, and J Xu. Software sustainability: The modern tower of babel, 2014. URL http://eprints.whiterose.ac.uk/84941/. (c) 2014, The Author(s). This is an author produced version of a paper published in CEUR Workshop Proceedings. Uploaded in accordance with the publisher's self-archiving policy.

Nina Wolfram, Patricia Lago, and Francesco Osborne. Sustainability in software engineering. In *2017 Sustainable Internet and ICT for Sustainability, SustainIT 2017, Funchal, Portugal, December 6-7, 2017*, pages 55–61, 2017. doi: 10.23919/SustainIT.2017.8379798. URL https://doi.org/10.23919/SustainIT.2017.8379798.

Didar Zowghi and Vincenzo Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993 – 1009, 2003. ISSN 0950-5849. doi: https://doi.org/10.1016/S0950-5849(03)00100-9. URL http://www.sciencedirect.com/science/article/pii/S0950584903001009. Eighth International Workshop on Requirements Engineering: Foundation for Software Quality.