Chapter 10
Software Quality and Quality Management

All engineering disciplines have notions of product quality. Along with these notions come mechanisms and best practices ensuring that, for a given product, each item of the product has a specified quality. Furthermore, we are used to thinking that the most critical of these quality metrics are absolute. If the product fails to meet these absolute quality metrics, the customer might have legal claims on the producer. Such quality breaches are therefore expected to be relatively rare in most engineering disciplines.

Our expectations of the quality of software are different. For software running on standard consumer equipment, we expect a stream of security updates. These updates fix security holes that clearly must have been in the product that was bought prior to the installation of the update. This state of affairs is not the result of the neglect of the software engineering community or the result of a lack of investment in quality assurance by the manufacturers. Instead, it is the result of inherent challenges in the concept of software development. The support of quality through firm notions and mathematical rigour existing in many other engineering fields has been shown to be hard to replicate in a scalable manner in software engineering (see Chap. 9).

Still, the quality assurance of software has received a massive amount of attention. In this chapter, we provide an overview of the most important developments, aiming to see if methods have been developed that can help address our question.

10.1 What is Software Quality Management?

Early in the development of computers, it became apparent that the instructions that guide the operation of machines could be stored as data. This naturally pointed towards the formation of a new notion, *software*, that encompassed the executable files in a computer system. The flexibility of separation between the physical machine itself and its operating instructions was shown to be immensely powerful and software soon developed into an engineering discipline in its own right.

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_10

Soon, however, it became evident that the engineering of software presented its own very specific challenges. Unintentional errors in the code appeared to be hard to avoid and, when created, equally hard to find and correct. Throughout the decades, a great deal of effort has been invested into finding ways to ensure the quality of developed software. These investments have provided insights into some key properties that make software quality assurance different from what we find in other engineering disciplines [14].

Complexity One way of measuring the complexity of a product is to count the number of distinct operating modes in which it is supposed to function. A piece of software may be required to work on top of different platforms, work together with different sets of other software running alongside, and be guided by a set of parameters and settings that can be freely chosen by the user. The combinatoric explosion of possible configuration settings for a software package easily runs into the millions. Traditional engineering artefacts rarely see them run into just the thousands.

Visibility Software is invisible. This means that the opportunity to find mistakes without observing their consequences is limited to very few people. In most physical products, many mistakes are impossible simply because they would be obvious to the eye of anyone looking at them.

Manufacturing The production phase is very different in software engineering, since it simply consists of making some files available, either through download or through the standardized process of producing storage media with the files on them. There is no production planning phase in which mistakes can be discovered and the manufacturing is carried out without anyone seeing the product.

Assuring the quality of software is therefore different from assuring the quality of most other products. Because of the general invisibility of software and the fact that its quality is not influenced by the process of manufacturing copies of it, software quality assurance is aimed at the production and maintenance phases. The main elements of software quality assurance are the development process, the quality model, and the implemented quality management scheme. These three notions are described in the upcoming sections.

10.2 Software Development Process

The initial *waterfall* model for software development was described in the early 1970s. Many variations of the model exist but they generally consisted of a series of stages, with the following stages in common: a requirements phase, a specification phase, an implementation phase, a testing phase, and a maintenance phase. Although the model has been ridiculed and used as an example to avoid, there is a general consensus that the tasks described in these phases remain important in any software development effort. Most of the models we have seen since, such as those

in iterative, incremental, and spiral approaches, relate to the tasks identified in the original waterfall method.

One particular trend that has been influential in recent years is the inclusion of insights on human psychology and interaction patterns into the software development process. This has led to the development of so-called *agile* approaches [12], among which *Scrum* [25] is particularly popular and successful.

10.3 Software Quality Models

The goal of any software development process is to promote the quality of the software under development. The notion of *software quality* is, however, highly nontrivial and it has therefore been subject to significant research and standardization efforts. These efforts have a common starting point in the efforts of McCall et al. [8, 22] and Boehm et al. [6], where quality aspects are ordered into a hierarchy. Figure 10.1 depicts the hierarchy as defined by Boehm.

Later efforts refined and updated the underlying ideas of McCall and Boehm in several ways. In particular, the quality attributes highlighted in these early models have been argued to be heavily based on the designer's point of view. There are, however, other stakeholders, such as the user/customer and the company that develops the software as part of a business model. Depending on the role, the importance given each attribute may differ [4]. Security as a quality metric has also grown considerably

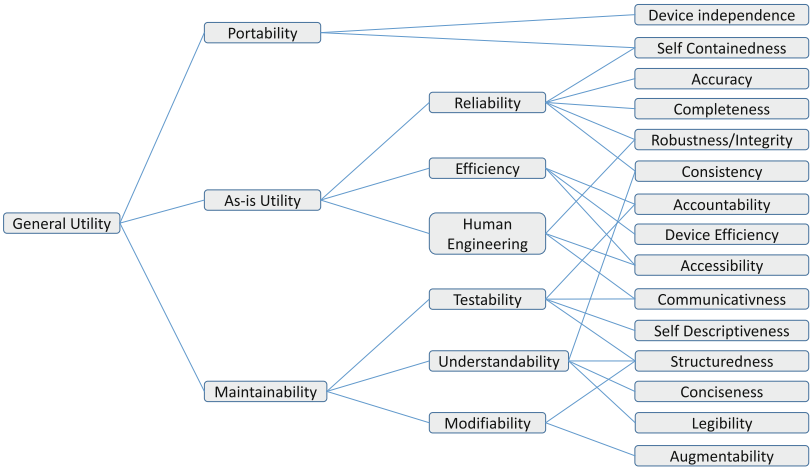

Fig. 10.1 The hierarchy of software quality characteristics of Boehm et al. [6]. The edges in the hierarchy denote necessary conditions. For example, testability, understandability, and modifiability are necessary conditions for maintainability

in importance over the decades. It is thus included in all modern quality models and has been the subject of significant academic scrutiny [15].

10.4 Software Quality Management

Quality management has a long history in production and development. Most of the concepts developed have been shown to be transferrable to software development, although with some adaptations. Alan Gillies [16] lists four principal aspects of quality management for software development:

Development procedures These include the tools used by the development team, in testing procedures, and in training staff.

Quality control This includes planning, progress meetings, documentation control, code walkthroughs, and so forth.

Quality improvement Organized staff activity aiming at establishing a quality culture amongst the staff.

Quality assurance Monitoring activity to ensure that all aspects of the quality system are carried out correctly.

There exist many different manifestations of these aspects in organizations, standards, and the academic literature. What they all have in common is that they specify the quality of work processes. This will, of course, have implications on the quality of the finished product, but hard requirements of the product itself are to be found elsewhere. We therefore move on to the metrics of software quality.

10.5 Software Quality Metrics

Whereas a quality model as described in Sect. 10.3 gives us a structure for the quality concept, the model has to be supported by metrics to be useful in a quality assurance setting. Unfortunately, there is a divide between the qualitative notions in a model and our ability to accurately capture these notions in numbers. It is a sad fact that the quality of software cannot be measured in terms of absolute and unambiguous scales [16].

Depending on the quality notions under scrutiny, the definitions and collection of metrics differ significantly in complexity. Whereas portability is a quality characteristic that can largely be captured by the frequency and number of platform-dependent constructs in the code, the notion of user friendliness is typically captured with metrics involving questionnaires and correctness/robustness can be captured by logging the number of post-release bugs corrected.

The main interest of this book concerns the metrics that relate to security. Since security has been included as a separate quality characteristic in most quality models, the world has witnessed many attempts to capture security in a quantitative way. Still,

as pointed out by Jansen [21], we have a long way to go and it is not likely that all aspects of the problem are resolvable. One reason for this is that our understanding of security does not degrade gracefully with the number of security flaws in the code. With one flaw, all security could be lost and the second flaw thus adds very little to the situation. Although this is not true in all scenarios, it is clearly true in ours. If a system vendor has included a hidden backdoor in the code, the number of backdoors included in total is of little significance.

10.6 Standards

The notion of quality management is intrinsically elusive, since it has different meanings for different people, in different roles, in different situations, and at different times. This issue presents problems in any industry that needs to be able to communicate with some degree of accuracy and efficiency how quality is ensured in an organization or for a product. A series of standards are being developed to address these problems and some of the ones used in the development of software are listed below [15].

ISO 9001 This is a general standard applicable to any organization in any line of business. It defines a set of operation processes and proposes designing, documenting, implementing, monitoring, and continuously improving these operation processes [17].

ISO/IEC 9126 This standard contains a quality model for software and a set of metrics to support the model. It was first issued in 1991 [20].

ISO/IEC 25010 In 2011, this standard replaced ISO 9126. It contains a modernized set of quality attributes and, in particular, attributes related to security were given more attention [18].

ISO/IEC 15504 This standard, which is sometimes referred to as SPICE, covers a broad set of processes involved in software acquisition, development, operation, supply, maintenance, and support.

CMM/CMMI The term *CMM* stands for Capability Maturity Model and specifies software development processes. It is structured such that a development organization can be defined as belonging to one of a set of *maturity levels*, depending on the processes it has in place. The term *CMMI* stands for Capability Maturity Model Integration, which superseded CMM [10].

ISO 27001 This model was specifically designed for the certification of information security processes and is part of a series of standards related to information security [19]. Like ISO/IEC 25010, ISO/IEC 15504, and CMM/CMMI, ISO 27001 is process centric. This means that the models are based on the underlying assumption that if the process is correct, the outcome will be satisfactory [16].

The use of such standards and their certification arrangements is likely to have had positive effects on the quality of software. It is also reasonable to assume that the standards that focus on security aspects, such as the ISO-27000 series and Common

Criteria, have contributed to the development of secure software and to the safe operation of information and communications technology installations. Still, it is important to note that the contribution of these standards lies in the structured use of methods that have been described and developed elsewhere. Therefore, if our problem – namely, verifying that an equipment vendor is indeed performing malicious acts against us – cannot be solved by existing technical approaches, then structuring these approaches into a standardized framework is not likely to help. Neither is the certification of an organization or a piece of software according to the same standards. In our quest for a solution, we must therefore look at the strengths of our technical methods. Only when they are strong enough can we benefit from making them into a standard.

10.7 Common Criteria (ISO/IEC 15408)

The *Common Criteria* [1] is the result of a joint effort by many nations. They comprise a standard that requires our particular attention for two reasons. First, like ISO 27001, it is aimed specifically at security. Second, unlike ISO 27001, it contains requirements for products, not only processes.

The key concepts in this standard are the *protection profiles* by which a user specifies security requirements. A vendor claiming to have a product meeting these requirements can have the product tested for compliance in independent accredited testing laboratories. The rigour of the testing can be adapted to the criticality of the component in question and, for this purpose, seven Evaluation Assurance Levels (EALs) have been defined. The lowest of these, EAL1, requires only functional testing for correct operation, whereas EAL7 mandates full formal verification of the source code:

- EAL1: Functionally tested.
- EAL2: Structurally tested.
- EAL3: Methodically tested and checked.
- EAL4: Methodically designed, tested, and reviewed.
- EAL5: Semiformally designed and tested.
- EAL6: Design semiformally verified and tested.
- EAL7: Design formally verified and tested.

The wide adoption of Common Criteria has been a major step towards more secure information technology systems. Still, nothing in this effort addresses cases in which the company developing the system injects unwanted functionality. The highest assurance level, EAL7, is, at the time of writing, considered prohibitively costly for most development projects for reasons discussed in Chap. 9. Furthermore, evaluation focuses primarily on assessing documentation, which ultimately has to be provided by the company that is not to be trusted, and secondarily on functional testing, which we concluded above can be easily fooled. So, at the end of the day,

we have to draw the same conclusion for Common Criteria as we did for the other standards. Structuring existing methods into a framework will not solve our problem unless there is a solution in the methods themselves. In the following sections, we cover the basic methods that make up the assurance levels of Common Criteria.

10.8 Software Testing

Testing remains the most frequently used technique for the validation of the functional quality of a piece of software. It has been claimed that, in a typical development project, more than 50% of the total cost is expended in testing and this figure has remained more or less stable since the early days of programming [24]. The importance of testing in the development of stable and trustworthy software has led to a large body of research in the field. Still, we have not been able to turn testing into an exact science in the sense that there exist test methodologies with the guaranteed ability to find all faults. There are several reasons for this.

First, for most software of any complexity, the combination of stimuli that the software should be able to handle is subject to a combinatorial explosion that prevents the tester from exhausting all possible test scenarios. This means that an important part of the testing effort is to choose the set of tests the software should undergo [2]. The goal of this test set could be to cover all branches of the code or – for some definition of input equivalence – all the equivalence classes of input parameters.

Most serious testing situations involve such a large set of test cases that they will have to be carried out automatically. This brings us to the second problem of testing, namely, that of automatically recognizing incorrect behaviour. In the field of software testing, this is often referred to as the *test oracle problem* [3]. In complex applications, this problem is frequently a major obstacle in the testing process.

Some software systems are actually deemed untestable. These are systems where exact test oracles cannot be defined or where the system interacts with the physical environment in such complex ways that it is impossible to adequately cover all the test cases. For these cases, building a model of the system and then performing tests on the model instead of the software itself has been proposed [7]. This concept is quite similar to model checking, which is discussed in Sect. 10.9.

Although software testing is and remains the main vehicle for the validation of software functionality, the state of the art falls short of helping us when facing a potentially dishonest equipment vendor. Triggering unwanted software can be accomplished by arbitrarily complex sets and sequences of stimuli, meaning that the likelihood of being detected in a test case can be made arbitrarily small. Furthermore, model testing is not likely to help us, since the malicious functionality will clearly not be communicated in a way that will let it become part of the model. For testing to be of help to us, we will need a scientific breakthrough that is not on the horizon at the time of this writing.

10.9 Verification Through Formal Methods

One of the methods referred to in the Common Criteria is code verification. This consists of writing an accurate specification of how the software should behave and performing a rigorous mathematical proof that the piece of software actually conforms to the specification. The correctness of this proof can in turn be checked automatically by a relatively simple program. In Chap. 9, we discussed formal methods in some detail and concluded that both hope and despair are associated with this approach.

The hope is related to the fact that the approach itself carries the promise of being watertight. If there is a formal and verifiable proof that there are no malicious actions in the software code, then a mathematical certainty has been established. The despair is due to two facts. First, it can be argued that formal methods only move the complexity from understanding the software to understanding and proofreading the specifications. The second – and this is the more serious one – full formal specification and verification are considered prohibitively costly for larger systems [11]. This is further supported by the fact that very few pieces of software have been reported to have undergone full formal verification as specified in the highest assurance level, EAL7, of Common Criteria.

One particular development that seems to address both the complexity of specification and the cost of formal verification is model checking [9]. This consists of building an abstract model of the software that is sufficiently small so that its state space can be made the subject of an exhaustive search. Although this is a promising approach, our problem pops up again in the question of how to build the model of the software. This task is intrinsically difficult and has to be carried out by human experts [5]. The likelihood of an intentional backdoor in the code showing up in the model is therefore quite similar to the likelihood of it being detected by reviewing the code itself. Model checking is therefore not a silver bullet to our problem, so we move on to seeing if code review can help.

10.10 Code Review

Code reviews have been performed in security engineering for a long time. The goal of code reviews in security is to look for known classes of vulnerabilities that typically result from programming mistakes. These reviews can be performed as part of the development process or take place entirely after the system has been developed. There are several types of reviews, depending on which code is available. Down et al. [11] mention the following alternatives, varying in the information available to the tester:

- Source only.
- Binary only.
- Both source and binary.

- Binary with debugging information.
- Black box, where only the running application is available.

As argued in Chap. 4, a review of the source code alone gives a dishonest equipment vendor full opportunity to include malicious code through the compiler or other development tools. In the case of the black box, we are, in principle, left with testing, a situation we discussed in Sect. 10.8. This leaves us with the testing situations in which the binaries are available.

Code reviews of binaries have many the same strengths and weaknesses as formal methods. On one hand, they hold the promise that, when carried out flawlessly and on a complete binary representation of a piece of software, all hidden malicious code can be found. Unfortunately, there are two major shortcomings: one is that finding all hidden malicious code is generally impossible, as shown in Chap. 5. Even finding some instances will be extremely challenging if the malicious code has been obfuscated, as described in Chap. 7. The second shortcoming is one of scale. Down et al. [11] estimate that an experienced code reviewer can analyse between 100 and 1,000 lines of code per day. Morrison et al. [23] estimate that a full analysis of the Windows code base should take between 35 and 350 person–years. To make things worse, these figures are based on the assumption that it is the source code that is to be analysed, not the binaries; the real costs are therefore likely to be higher.

One way to handle this problem is to perform a code review on only parts of the software product. Which parts are to be selected for scrutiny could be chosen by experts or by a model that helps choose the parts that are more likely to contain malicious code. Such *defect prediction models* have been used for some time in software engineering to look for unintended faults and a range of similar models have been proposed for vulnerability prediction. The value of the vulnerability prediction models is currently under debate [23]. We do not need to take a stand in that discussion. Rather, we observe that the models try to identify modules that are more likely to contain unintended vulnerabilities. If the vulnerabilities are intended, however, and we can assume that they are hidden with some intelligence, then a model is not likely to be able to identify the hiding places. Even if such a model existed, it would have to be kept secret to remain effective.

It is thus a sad fact that rigorous code reviews and vulnerability detection models also leave ample opportunity for a dishonest equipment producer to include malicious code in its products.

10.11 Discussion

Fuggetta and Di Nitto [13] took a historical view of what was perceived as the most important trends in software process research in 2000 and compared this to what was arguably most important in 2014. One of the key observations is that security has gained significantly in importance and, consequently, the community should ‘identify and assess the major issues, propose ways to monitor and manage threats, and

assess the impact that these problems can have on software development activities.' Although the statement was intended for scenarios with unintended security holes and where the perpetrator is a third party, it does shed light on the extent to which software development processes and quality assurance are promising places to look for solutions to the problem of untrusted equipment vendors and system integrators. If these tools and methods are inadequate to help the developers themselves find unintended security holes, they are probably be a far cry from helping people outside of the development team to find intentional and obfuscated security holes left there by a dishonest product developer.

The findings in this chapter confirm the above observation. We have covered the concepts of development processes, quality models, and quality management and have found that they relate too indirectly to the code that runs on the machine to qualify as adequate entry points in the verification of absence from deliberately inserted malware. The field of software quality metrics in some aspects directly relates to the source code but, unfortunately, it is doubtful that all aspects of security are measurable. In addition, counting the number of bugs found has still not given us bug-free software; thus, it is not likely that counting security flaws will provide any guidance in finding deliberately inserted malware.

Methods that go deep into the code running on the machine include testing, code reviews, and formal methods. In the present state of the art, formal methods do not scale to the program sizes we require, code review is error prone and too time-consuming to be watertight, and testing has limited value because of the problem of hitting the right test vector that triggers the malicious behaviour. Even when triggered, recognizing the malicious behaviour may be highly nontrivial. It is clear that these methods still leave ample space for dishonest equipment providers to insert unwanted functionality without any real danger of being exposed.

Although these approaches still fall far short of solving our problem, they remain a viable starting point for research on the topic. In particular, we hope that, at some time in the future, the combination of formal methods, careful binary code review, and testing will increase to a significant level the risk of being exposed if you include malware in your products. However, we have a long way to go and a great deal of research must be done before that point is reached.

References

1. http://www.commoncriteriaportal.org
2. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
3. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
4. Berander, P., Damm, L.O., Eriksson, J., Gorschek, T., Henningsson, K., Jönsson, P., Kågström, S., Milicic, D., Mårtensson, F., Rönkkö, K., et al.: Software quality attributes and trade-offs. Blekinge Institute of Technology (2005)

5. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer Science and Business Media, Berlin (2013)
6. Boehm, B.W., Brown, J.R., Kaspar, H.: Characteristics of Software Quality. North-Holland, Amsterdam (1978)
7. Briand, L., Nejati, S., Sabetzadeh, M., Bianculli, D.: Testing the untestable: model testing of complex software-intensive systems. In: Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), ACM (2016)
8. Cavano, J.P., McCall, J.A.: A framework for the measurement of software quality. In: ACM SIGMETRICS Performance Evaluation Review, vol. 7, pp. 133–139. ACM (1978)
9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
10. CMMI Product Team: CMMI for development, version 1.2 (2006)
11. Down, M., McDonald, J., Schuh, J.: The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson Education, Boston (2006)
12. Fowler, M., Highsmith, J.: The agile manifesto, Softw. Dev. **9**(8), 28–35 (2001)
13. Fuggetta, A., Di Nitto, E.: Software process. In: Proceedings of the on Future of Software Engineering, pp. 1–12. ACM (2014)
14. Galin, D.: Software Quality Assurance: From Theory to Implementation. Pearson education, Boston (2004)
15. García-Mireles, G.A., Moraga, M.Á., García, F., Piattini, M.: Approaches to promote product quality within software process improvement initiatives: a mapping study. J. Syst. Softw. **103**, 150–166 (2015)
16. Gillies, A.: Software Quality: Theory and Management (2011). https://www.lulu.com
17. ISO 9001:2015 quality management systems - requirements
18. ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models
19. ISO/IEC 27001:2013 information technology – security techniques – information security management systems – requirements
20. ISO/IEC 9126-1:2001 software engineering–product quality–part 1: quality model
21. Jansen, W.: Directions in Security Metrics Research (2009)
22. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in software quality. volume I. Concepts and definitions of software quality. Technical report, DTIC Document (1977)
23. Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. ACM–Association for Computing Machinery (2015)
24. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, New York (2011)
25. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice-Hall, Englewood Cliffs (2001)