Reliable Software: Industry Best Practices

Paul D. Martin, Ph.D.

Harbor Experts

I. INTRODUCTION: ALGORITHMS, IMPLEMENTATIONS, AND ERROR SOURCES

1. To begin, it is of fundamental importance to understand what industry best practices entail. To do so we must first understand why software must be developed in accordance with any practices at all, and why merely testing its output is insufficient to rule software reliable. We must therefore first differentiate between an algorithm and its implementation. An algorithm is an ordered sequence of steps for performing a particular task. Algorithms can be analyzed, their properties can be studied, and their theoretical effectiveness can be measured. To be used in a software product, however, an algorithm must be implemented by a software developer.

2. While algorithms are a high-level mathematical concept, the implementation of these constructs falls into the realm of software development. Just as a structural engineer may design and review bridge plans, it comes down to the actual construction workers to implement these plans correctly without any mistakes. As with other types of engineering, sometimes mistakes are made in the process of implementing an algorithm. These mistakes can be obvious on their face—the algorithm produces the wrong output every time it is run; attempting to run the implementation of the algorithm causes the code to crash outright; the code doesn't run at all; et cetera. However, such errors are easy to detect and correct. Most errors in practice are therefore much more subtle. The algorithm mostly works and produces reasonable output in easy to observe cases, but in less common cases or under certain conditions the program produces incorrect output.

3. Such errors are so common in computer science that entire systems of knowledge have been developed around training developers to produce fewer errors through use of specific software development "best" practices and around methodology for testing, detecting and fixing errors. Professional organizations in the field of computer science such as the Institute of Electrical

and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM), which (despite the historical origins of their names) are two of the leading professional organizations in the field of software development, publish extensively on these topics.

II. FOUNDATIONAL TESTING STANDARDS AND DEFINITIONS

4. For example, the IEEE has published a widely referenced guide to software development best practices called the Software Engineering Body of Knowledge (SWEBOK)[1]. As it describes, "In recent years, the view of software testing has matured into a constructive one. Testing is no longer seen as an activity that starts only after the coding phase is complete with the limited purpose of detecting failures. Software testing is, or should be, pervasive throughout the entire development and maintenance lifecycle." [SWEBOK at 4-1]. The SWEBOK contains a detailed description of testing best practices. "Indeed, planning for software testing should start with the early stages of the software requirements process, and test plans and procedures should be systematically and continuously developed—and possibly refined--as software development proceeds. These test planning and test designing activities provide useful input for software designers and help to highlight potential weaknesses, such as design oversights/contradictions, or omissions/ ambiguities in the documentation. … It is perhaps obvious but worth recognizing that software can still contain faults, even after completion of an extensive testing activity. Software failures experienced after delivery are addressed by corrective maintenance." [SWEBOK at 4-2]. Thus, it is important to note that the IEEE recognizes the fact that despite a robust testing environment, software implementations can still contain bugs. Indeed, the SWEBOK and other similar guides such as the ISO Software Testing Standard 29119 describes a large set of possible

[1] https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3

testing methodologies and topics that all, at their heart, are designed to help finding programming errors in the implementation of software code.

5. Software testing "consists of the *dynamic* verification that a program provides *expected* behaviors on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain" [SWEBOK 4-1]. Essentially, testing allows us to identify when the software is not performing how it should, where this error occurs, and how it can be fixed. As the SWEBOK states, "One of the aims of testing is to detect as many failures as possible." [SWEBOK at 4-3]. As mentioned above, good testing allows software designers to build robust and safe applications. Without testing, one risks a faulty product that can adversely consumer safety, whether digital or physical.

III. TYPES OF TESTING

6. To use another analogy, how and why we test software is like how and why one would test a car. It would not be enough to say that the car overall runs; consumers, builders, and engineers want to ensure that the car would run with minimal maintenance, each component works as it should, and that the car has performance guarantees (i.e., it can move from place A to place B). Without proper testing of the car, consumer safety can be jeopardized. For example, Nissan issued a recall of cars from 2002-2006 due to a faulty airbag sensor.[2] Thus, testing not only helps developers identify faults in the software, but also encourages consumer safety.

7. Software testing is categorized into levels of testing that range in order the base level of individual components (i.e., unit testing), the interaction between components (i.e., integration testing), and the highest level of the entire software application (i.e., system testing) [SWEBOK at 4-1]. Each level of testing provides varying information on the state and

[2] https://www.nhtsa.gov/press-releases/do-not-drive-warning-nissan-infiniti-takata#:~:text=Nissan%20has%20issued%20a%20%E2%80%9CDo,defective%20air%20bag%20is%20replaced.

performance of the application. In the context of cars, unit testing would refer to testing each component. For example, testing whether the battery has power or if the lock button locks the car. Each of these are individual features within the car that one would test and failing any of these would point to exactly where the problem exists. In the software context, unit testing would refer to testing a function within the software application. For example, consider a function that determines if a number is odd or even. Unit testing this function would ensure that the function returns true for 2 and false for 1.

8. The next level of testing, integration testing, gauges how the components interact with each other. In the context of cars, the brake pedal should interact with the cruise control system, and if a driver steps on the brake pedal, cruise control should disengage. Integration testing would ensure that this interaction happens when expected. A failed integration test indicates that although the components may work individually, the system comprising the whole of the components does not properly function when the components are combined. In software, an integration test could be used to test whether a database and an application communicate with each other.

9. System testing aims to test the state of the complete system and contains several types of testing, such as: performance, stress, security, usability, and regression testing. Each type of system testing describes the state of the behavior of the system. Performance testing evaluates that the system performs as expected. For example, when testing a car, a test engineer would make sure that a car does not breakdown driving above 90 miles per hour or after being driven 1,000 miles. This is all within expected performance. Stress testing attains at what level the system breaks down. In the context of cars, a car should be able to run hundreds of hours in a row, and in software, we would expect a software system to be able to handle thousands if not millions of simultaneous

users. Security testing ascertains to what extent the system is secure. Security testing a car would mean that a car is not opened by a key for a different car. Usability testing determines the extent each user can use the system. For example, a usability test in a car could determine if a user can turn on the emergency lights. Each of these tests play an integral part of the engineering and development process and are common in industries other than software. Failure to perform these tests results in catastrophic failures that is often not determined until after the deployment of the system when users have been affected. For example, Tesla recalled the Cybertruck to fix faulty windshield wipers resulting in financial losses.[3]

IV. INTEGRATING TESTING THROUGHOUT THE SOFTWARE DEVELOPMENT LIFECYCLE

10. The importance of software testing resulted in the development of industry standards by bodies, such as the IEEE SWEBOK, and the publication of well-known industry publications such as *Code Complete*[4] and *The Pragmatic Programmer*,[5] that emphasize the importance of software testing and provide foundational expectations of such. "Software testing is, or should be, pervasive throughout the entire development and maintenance life cycle" [SWEBOK 4-1]. This idea is further reiterated by *The Pragmatic Programmer*, "we need to build testability into the software from the very beginning" [pg. 189]. Testing practices are not an afterthought to build around but a fundamental part of the development cycle. As Code Complete describes, "developer testing should probably take between 8-25% of the project time" [*Code Complete* pg. 502]. To lean on the analogy of car testing, car manufacturers do not build safety tests for airbags after the car is designed and manufactured. The design reflects the need to pass

[3] https://www.cnbc.com/2024/06/25/tesla-recalls-cybertruck-to-fix-faulty-windshield-wipers-loose-trim.html
[4] https://en.wikipedia.org/wiki/Code_Complete
[5] https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/

these tests. “Writing test cases first forces you to think at least a little bit about the requirements and design before writing code, which tends to produce better code” [*Code Complete* pg. 504]. For example, in software, one would not design a stress test to make sure that a social network can handle millions of simultaneous users after the development process. The social network should be designed with the test in mind.

11. Testing is a continuous process throughout the software development cycle. “Testing plans and procedures should be systematically and continuously developed” [SWEBOK 4-2]. As they are integrated with the software development cycle, “testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process” [SWEBOK 4-13]. Good documentation “is an integral part of the formalization of the test process” [SWEBOK 4-13]. These test documents include “the test plan, test design specification, test log, and test incident report” [SWEBOK 4-13]. Further, a “repository of test materials should be under the control of software configuration management” [SWEBOK 4-13]. Thus, industry standards state that a full testing plan managed by a team should be in place. Expected results for each test should be included, and “execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results” [SWEBOK 4-15]. After performing the tests, "the results of testing should be evaluated to determine whether or not the testing has been successful” [SWEBOK 4-15]. If a failed test happens, “an analysis and debugging effort is needed to isolate, identify, and describe it” [SWEBOK 4-15]. Further, “when test results are particularly important, a formal review board may be convened to evaluate them” [SWEBOK 4-15].

12. When industry standards are not met, the consequences can be drastic—harm to consumers, monetary losses, and reputation loss. For example, Baseus recalled magnetic power

banks due to overheating of the lithium-ion battery leading to fires.[6] This issue could have been found by performing extensive stress testing prior to sale. The IEEE states that "thorough testing is essential to ensure software systems function correctly, are secure, meet stakeholders' needs, and ultimately provide value to end users."[7] Software testing alleviates many issues that are found. In yet more examples, Starbucks was forced to temporarily close 60% of their stores due to a problem in their payment software, leading to lost sales[8]; a Bloomberg terminal bug crashed the stock market and forced the UK to delay bond selling[9]; a software error in a Boeing airplane caused a plane crash that resulted in over 200 deaths[10]; a race condition and improper isolation in the design of the Therac-25 radiation therapy caused the irradiation and death of six cancer patients.[11] Each of these problems were directly correlated with a lack of sufficient software testing and caused catastrophic losses. By meeting software testing industry standards, companies can mitigate problems that users would face. As stated in *The Pragmatic Programmer*, "All software you write will be tested – if not by you and your team, then by eventual users."

[6] "Baseus has received 171 reports of incidents, including 132 reports of bulging or swelling batteries and 39 reports of fires, resulting in 13 burn injuries and about $20,000 in property damage." [https://www.cpsc.gov/Recalls/2024/Baseus-Magnetic-Wireless-Charging-Power-Banks-Recalled-Due-to-Fire-Hazard-Imported-by-Shenzhen-Baseus-Technology]

[7] https://www.computer.org/resources/importance-of-software-testing

[8] https://www.geekwire.com/2015/starbucks-lost-millions-in-sales-because-of-a-system-refresh-computer-problem/

[9] https://www.theguardian.com/business/2015/apr/17/uk-halts-bond-sale-bloomberg-terminals-crash-worldwide

[10] https://www.barrons.com/news/inquiry-into-2019-ethiopian-air-crash-confirms-software-failure-01671821708

[11] https://ethicsunwrapped.utexas.edu/case-study/therac-25

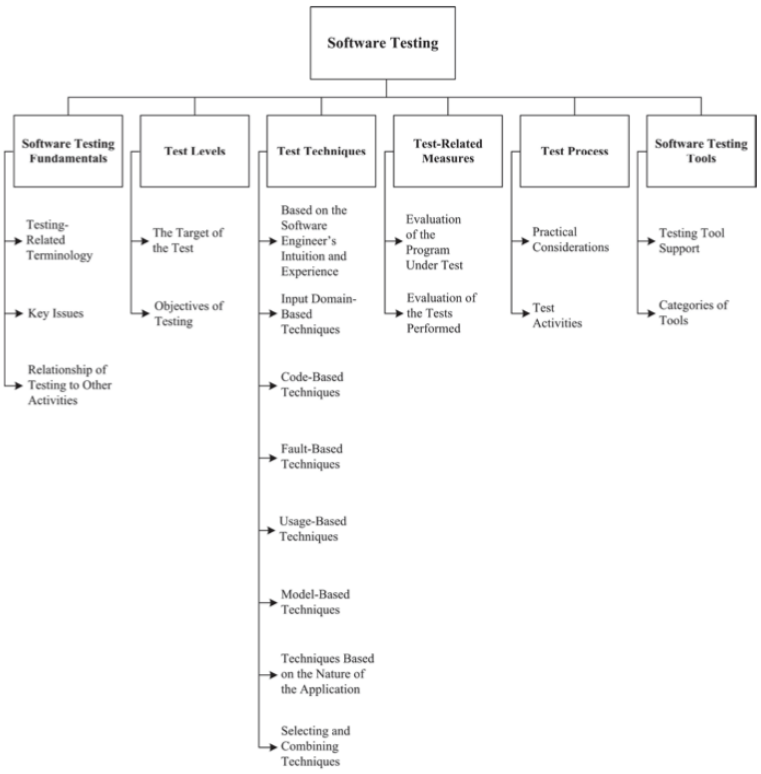

Figure 4.1. Breakdown of Topics for the Software Testing KA

[SWEBOK Figure 4.1]

V. COVERAGE METRICS

13. In my opinion, and according to software engineering standards, each function should have its own unit testing, and any interaction between functions or services should have its own integration testing. Indeed, this is what I teach in my courses at Johns Hopkins when describing how to help find and mitigate security vulnerabilities in code. Furthermore, Control flow-based coverage criteria is a common measure for determining the adequacy of the testing plan. "The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage has been achieved" [SWEBOK 4-9]. *Code Complete* further emphasizes this, "a better coverage standard is to meet what's called

'100% branch coverage,' with every predicate term being tested for at least one true and one false value." [pg. 504].

VI. ROLE OF MANUAL SOURCE CODE REVIEW

14. In addition to testing, another way to identify likely sources of errors in a software project is to perform a manual review of the source code. Source code reviews are extremely common in legal cases. Indeed, I have personally performed or supervised more than 48 source code reviews in cases ranging from breach of contract to faulty software due to improper software development practices to patent infringement.

15. There are many computer science experts who routinely review large codebases for legal cases. Indeed, source code review on codebases with hundreds of thousands, millions, or even billions of lines is a common part of many legal cases with substantial technical aspects and I have personally performed and supervised many such reviews. These reviews have gleaned substantial meaningful information including allowing me to determine the functioning of protocols and algorithms as well as to understand whether code was written in a way that adhered to reasonable industry standards.

16. Although source code bases can be large, indeed I have reviewed codebases with more than one million files, it is neither necessary nor possible to review each file individually. First, it is important to note that in large codebases many files are typically either autogenerated, boilerplate, or part of third-party dependencies that do not need to be separately reviewed. Further, it is generally the case that a source code review of a large codebase is looking to understand how specific functionality is implemented. As such, much of the code that is produced in a codebase is not directly relevant to the functionality being reviewed—this key functionality can often represent a tiny subset of the source code that was produced in a case. Even in the case that the

amount of source code to review is large, there exist many popular tools to help a manual reviewer improve the efficiency of source code review. For instance, an incomplete list includes searching tools such as grep,[12] which is commonly used to search for specific symbols such as variable and function names in the source code. Similarly, integrated development environments and programmer text editors such as Visual Studio Code[13] are commonly used to visualize and navigate codebases with which the reviewer is unfamiliar. It allows a reviewer to quickly jump from the use of a symbol to its definition, even when the definition and the use are in different source code files. It also makes tracing and cross-referencing function calls efficient. In fact, I have co-authored a whitepaper on source code reviews in which I described a variety of tools that are typically useful in a code review.[14]

17. Source code reviews can achieve several goals. In some cases, they can be used to understand how a particular algorithm or workflow works in a commercial product. Such code reviews are common in intellectual property litigation. Other source code reviews are designed to discover and catalog code quality issues that can lead to errors. This is common in code reviews related to breach of contract but can also be useful for code reviews about software reliability. Source code review can also be used to determine how and whether source code has been developed and tested in a robust fashion that is compliant with industry best practices.

[12] https://www.gnu.org/software/grep/
[13] https://code.visualstudio.com/
[14] https://harborexperts.com/Harbor_Experts_Source_Code_Review.pdf