Taxonomy of Common Software Testing Terminology: Framework for Key Software Engineering Testing Concepts

Robert F. Roggio
broggio@unf.edu

Jamie S. Gordon
jamie.s.gordon@unf.edu

School of Computing
University of North Florida
Jacksonville, FL 32224, United States

James R. Comer
j.comer@tcu.edu
Computer Science Department
Texas Christian University
Fort Worth, TX 76129, United States

Abstract

Most accredited computing programs have at least a single course addressing a software development process. These courses typically include a discussion of fundamental concepts and terminology that includes software testing. While many key concepts are in common use, terms describing testing are often misunderstood, misused, and misguided. The purpose of this paper is to provide a framework for commonly used and misused terminology central to software testing, and also to demonstrate their application in three common classes of testing: static and dynamic testing, black box and white box testing, and verification, validation, and acceptance testing.

Keywords: software testing, static and dynamic testing, black box and white box testing.

1. SOFTWARE TESTING

Background

The term, software testing, often evokes conflicting understandings of what is meant. What is being tested, what is a test, who performs the tests, and what is a "tester"? Additionally, what is the difference between a program having a fault, or error, or failure, or defect, and what are the various kinds of tests and what are their similarities and differences? The authors of this paper feel that a basic understanding of these principals is essential in order to provide a framework of terminology when software engineers – or, for that matter, any stakeholder, discusses the subject. Is it possible to talk about an essential activity, such as testing, such that all participants have a

consistent understanding of the meaning? Sadly, rarely is this the case, as evidenced by Naik and Tripathy, Galin, and others. (Niak & Tripathy, 2008) (Galin, 2004) (Juran, 2000) It seems as if one must define context before positive conversation may ensue. Thus, the effort to develop a common base of understanding appears to have merit.
Interestingly, the importance of a paper on essential concepts arose during development of another paper that sought to address differences between traditional testing procedures and object-oriented testing procedures. While discussing the subject of testing, the authors noted different understandings, and perceptions, of many commonly used terms. Humbling as it was, this was the reality that prompted the development of the current paper.

Definition of Software Testing
Software testing is a verification process for the assessment of software quality and a process for achieving that quality (Naik & Tripathy, 2013). Interestingly, software testing is used to support the interests of all stakeholders of an application. In particular, software testing is essential for:

- end-users to determine whether developed or otherwise maintained software meets specifications,
- developers to ensure that the code successfully implements a credible design,
- designers to ensure that their solution is one that meets specifications,
- and, to testers, to ensure that products to be delivered do indeed meet the client's needs.

Moreover, stakeholders include:

- customer service representatives who are often charged with responding to clients who 'call' to communicate a malfunction,
- and, to administration and finance individuals who may bill clients for software provided.

The list is endless and all have a vested interest in what is called - 'testing.'

Given this backdrop, it should be clear that different levels of testing need to be done by various stakeholders at different times (during or subsequent to development). To do so requires that procedures be designed to uncover issues - all with various views of outcomes. Thus, in order to frame this paper, the authors have limited the treatment of testing to those stakeholders whose main concern is the design, implementation (programming), and end user testing.

Please also note that while the categories are indeed different in many respects and hold different meanings for different stakeholders, there is considerable overlap. The specific workplace for software development will no doubt have its own vocabulary in addressing the world of software testing. To begin, it is important to establish a basic set of definitions.

2. TESTING TERMINOLOBY

Terms
Four useful and related terms, are frequently encountered when dealing with events that occur when software fails to perform as expected (Niak & Tripathy, 2008). References to these terms: ***failure***, ***error***, ***fault***, and ***defect*** are common in the industry; yet, unfortunately, although their means are related, they have different interpretations among practitioners. As an overview:

- A ***failure*** is defined as a behavior exhibited by a system that does not match what has been described in specifications.
- An ***error*** is an incorrect system state which could lead to a failure.
- A ***fault*** is the cause of an error. In general a fault leads to an error which leads to a failure, although not strictly so (Naik & Tripathy, 2013).
- A ***defect***, also according to Niak & Tripathy, refers to a design issue that leads to faults, although this is not as strict a definition (Niak & Tripathy, 2008).

Similar to Niak and Tripathy's terminology framework may be found in Galin. (Galin, 2004) His approach is very similar to that of Niak and Tripathy. Stressing that as practitioners we are mainly interested in software failures that disrupt or interrupt the use of software, he asserts that we must examine the relationship between software faults and failures. (Galin, 2004)

Galin begins with the simplest term, **software error** and offers that this can be a simple grammatical error in a line of code or a logical error in carrying out one or more of the client's requirements. But, once stated, Galin continues to point out that not all software errors become **software faults**. A software error may indeed cause improper functioning of the software in general or in a specific application but in other

instances, the error may not cause a problem in the software as a whole; sometimes "part of these cases … the fault may be corrected or "neutralized" by subsequent code lines." (Galin, 2004)

Galin goes on to assert that we are interested in the relationship between software faults and software failures. Recognizing that not all software faults end up as software failures, he points out that a **software failure** occurs only when it is "activated." Thus in many executions of a piece of software, the software fault is never discovered because specific software executions do not activate the software fault. Of course, then, in these instances, no software failure is discovered.

Galin captures his approach to software errors, faults, and failures nicely in Figure 1.

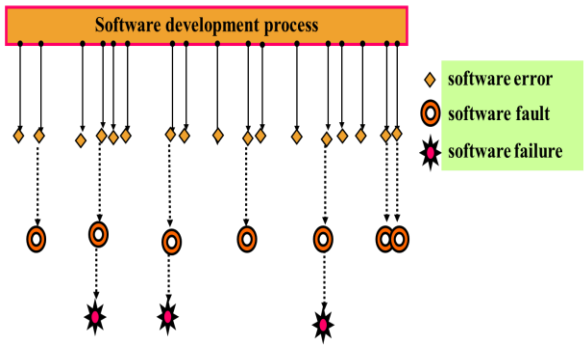

Figure 1 Software Errors, Software Faults and Software Failures (Galin, 2004)

Still others have different 'takes' on these terms. Walia and Carver state that an **error** is a mistake in the human thought process while trying to understand given information, solve problems or use methods and tools. (Wallia & Carver, 2012) **Software faults** are defined by IEEE as "an incorrect step, process, or data definition in computer programs." Favaro and her colleagues state that a **software failure** is "the inability of code to perform its required function within specified performance requirements." (Favaro, et al, 2013)

Down to Earth Examples

Let's consider a very simple example to illustrate these differences. Consider a specification that requires a very basic computation such as distance = rate * time. This is simple enough. This is a basic formula given in physical science 101. Algebraically, solving for rate would be defined as rate = distance / time. Applying this relationship to an automated solution designed to compute distance as a function of rate and time, we can address the standard definitions more closely.

Defect

Starting with design, perhaps the formula is erroneously misunderstood and designed as distance = rate + time (vice distance = rate * time). Clearly, if coded incorrectly, the resulting outputs would likely produce what might appear as a reasonable result; that is, until software testing is undertaken. A software developer, tester, end user, analyst, etc. might discover that the answers are incorrect in specific test cases. The defect is in the design. The formula is incorrect. The 'solution' to the requirement is incorrectly specified and designed, and although the program may well run to, end of job, the defect is (hopefully) clear.

Stutzke integrates treatment of these terms by defining a defect as "An observation of incorrect behavior caused by a failure or detection of a fault." (Stutzke, 2005) The failure in this case is an incorrect result (discovered during testing) and is the manifestation of a fault or incorrect result; Stutzke goes on to point out that the fault is an error that could cause a program to fail or potential failure. He defines error as the amount by which the result is incorrect.

Error

The failure was the production of an incorrect system state: the producing of an incorrect value. The state of the system is now incorrect. For the distance = rate + time, the resultant state of distance is incorrect.

Stutzke cites that an error can be the simple result of a misunderstanding. He cites the fault-tolerance discipline that addresses these terms: in Fault Tolerance the discipline distinguishes between human action (a mistake), the manifestation or result of the mistake (hardware or software fault), the specific result of the fault (a failure), and the amount by which the result is incorrect (the error). Again, the defect is the observation caused by the failure (event) or detection of a fault.

Fault

The fault is the cause of the error which was a design defect leading to this fault. A fault led to

a failure, the incorrect result discovered by testing. The fault here is implementing the design defect (distance = rate + time) which manifests itself in the detection of a failure.

Failure
It is commonplace to say the fault (cause of the error) led to a failure, where the failure in our example is the behavior of the application (adding rate to time in lieu of multiplying rate by time) during run time to produce the expected result. The production of an unexpected result points out a fault.

Conclusion
While all this might at first glance appear to be unimportant, the differences between discovering errors in design as opposed to discovering failures in implementation are quite significant from a cost perspective. Thus, realizing that a software defect is a design issue vis' a vis' one associated with implementation can affect the overall development and testing processes and can negatively impact the understanding of what the engineering of software really means.

It is important for a software engineer to have a commonly accepted set of terminology for communications, which is central to modern software development practices. To successfully communicate, we need a common language. Precision in identifying root causes of software errors (design defect, implementation fault, etc.) is essential to good software development practices so that proper best practices can appropriately address the wide-ranging origins of software errors.

3. TYPES OF TESTING

Software testing can be classified into many subcategories, often depending on one's perspective and often based on terms in common use in one's working environment. According to *Software Test Engineering @ Microsoft*, a number of test categories arises from the breaking down of work items in a workplace. This paper suggests a list that includes functional testing, specification testing, security testing, regression testing, automation testing and beta testing. The paper cites that the list is intentionally incomplete and requests supplements to the list. One response included unit/API testing, acceptance testing, stress/load testing, performance benchmark testing, and release testing. Still another response included performance testing, stress testing, interoperability testing, conformance testing, static testing, and maintainability testing. (blogs.msdn.com/b/chappell/archive/2004/03/24/95718.aspx) This diversity clearly supports that there are simply many types of testing, and that types of tests appear to be centered on one's focus or interest. Given this, the authors have taken liberty to divide software testing into a few different broad categories to include static and dynamic testing, white-box and black-box testing, and verification and validation testing.

Static and Dynamic Testing

Static Testing
In general, static testing can be performed on both documentation (specification documents, design documents, etc.) and source code (pseudo-code, source programs, scripts, etc.). (Johnson, 1996) Pressman discusses static testing tools as those that embody tools used to test code, specialized testing languages, and requirements-based testing tools. (Pressman, 1997) Code-based testing tools process source code (or a program description language) as the primary input and undertakes several analyses resulting in generation of test cases. They also identify a number of poor programming practices (identifiers defined and not used; incompatibilities between definition and use of attributes and more). Specialized testing languages enable a software developer to develop detailed test specifications and describe each test case and the logistics needed for its execution. Requirements-based testing tools inspect user requirements to suggest test cases or classes of tests to exercise the requirements. All of these are accommodated without any execution of code.

Certainly careful static analysis of documentation can reveal many issues. Defects may be discovered in the specification and/or design stages as well, without any need for any actual program development and subsequent execution. For example, Structure Charts for procedural development and many UML diagrams (class diagrams, object diagrams, subsystem and package diagrams, sequence and communications diagrams) are all candidates for testing without any 'program' execution. All of these may well lead to the discovery of defects by observing how, for example, a sequence of object responsibilities (methods) are invoked in a sequence diagram used to capture the procedural flow in a scenario captured from a

use case. Such an analysis might lead to the movement of responsibilities from one object to another in the interests of good design.

Consider static analysis of requirements. Static analysis of requirements can take place by visually inspecting the specification document and test for sufficiency, necessity, feasibility, completeness, and measurability. While indeed we are reviewing specifications, tests of this nature are static and do test the specifications.

Consider static analysis in design. Consider then a simple sequence diagram that is used to show the collaboration of objects and their method calls that are 'designed' to implement a scenario in a use case. In developing the sequence diagram, it is reasonably easy to discover that responsibilities assigned to an object, that is, methods, are poorly placed. For example, good cohesive design encourages the incidence of attributes and the methods that process these attributes to be located within the same object. In developing the sequence diagram, poor design can readily show that the methods and the data are not together. This kind of static test can easily result in modifying the object design so as to improve cohesion and hence provide for a better design. Again, this is a simple static test in tracing through a scenario in its accommodation in OO design. Additional static design tests include viewing, for example, UML diagrams to determine degree of coupling, object obsolescence, candidates for dividing and conquering complex objects and more.

Traditionally, static testing often addresses programming and deals with analysis of written code through walk-throughs and/or code inspections that result in algorithm analysis, and syntax or semantic checks (Nail and Tripathy, 2008). However, no actual execution is done in this stage as 'static' testing implies. It is purely investigation of the structure of code and hypothesizing what might happen at run-time. Many compilers and integrated development environments (IDE's) are designed to greatly assist programmers with this process. An example of static testing in programming is running a static analyzer looking for unreachable code, or 'dead code' that often arises in programs that have been modified over the years. In cases where programs have been maintained over a period of many years, they may have undergone many changes. Oftentimes a programmer must surgically delve into existing code to add features or correct errors without corrupting the existing functionality. Usually the programmer is given insufficient time to do a thorough analysis and must modify the program for a redeployment within often severely imposed time constraints. The programmer must react quickly and precisely and is not afforded the time he/she might need in order to undertake a thorough analysis.
A static view of code may reveal shortcomings via visual 'smells' that suggests the need for refactoring. Code smells, in and of themselves, are not bugs and do not necessarily lead to a non-functioning program. They may, however indicate weaknesses in design and may lead to code failure in the future. Long, multi-functional classes, methods with large numbers of parameters and options and many more smells suggested by Fowler may well suggest refactoring. (Fowler, 2012)

Dynamic Testing
In contrast to static testing, dynamic testing involves execution of a design or written code (m*ost dynamic testing is done on code).*

Pressman states that "dynamic testing tools interact with an executing program checking path coverage, testing assertions about the values of specific variables, and otherwise instrumenting the execution flow of the program." (Pressman, 1997) Niak and Tripathy state that dynamic testing involves analysis of behavioral and performance of the design and code (Naik and Tripathy, 2008), while Schulmeyer and MacKenzie cite that dynamic analysis methods involve the execution of a development activity designed to "detect errors by analyzing the response of a product to sets of input data." (Schulmeyer and MacKenzie, 2000) Clearly, desired outputs and/or ranges of output must be known ahead of time. Too, testing is the most frequent dynamic analysis activity. It is interesting to note that while dynamic testing is most often associated with code execution, dynamic testing can be applied during prototyping – especially during software requirements verification and validation. While the precise outputs are likely not always known, it can sometimes be determined that the system response to an input meets system requirements.

To show how broadly the principles of dynamic testing extend, Schulmeyer includes the running of static analysis tools as part of what he calls *Implementation Verification* and the running of

dynamic analysis tools as part of *Validation*. We will concentrate on dynamic testing of code.

Dynamic Testing of Code represents a very large and encompassing set of tools for software testing. As an example of practical dynamic testing, consider the following real-world example that formed a part of dynamic testing of major programs.

Consider a program called *Percent Execute*; a program used long ago in the U.S. Air Force. Its purpose was to monitor the run-time behavior of programs as part of the testing activities before deployment of the software. The purpose of Percent Execute was simply to discover how much (literally) of a program was actually executed given an input dataset. Given specific inputs (and several different sets of inputs), just what portions of a program were / were not executed? Clearly, different input data would cause different execution paths to be executed. The methodology called for a source program to be instrumented with source code probes (discussed later) that were inserted into every program unit (method, function, paragraph, module, etc.). Afterwards, the program was re-compiled and executed with carefully designed sets of input data to determine what parts of the program were being executed. Dynamic testing clearly (and often) revealed that key parts of the executable code were not exercised. This was disturbing given that essential edits were discovered to go unexecuted but were assumed to have been. For example, edits in financial programs to ensure financial and data integrity were sometimes simply not executed for some input data. Without dynamic testing, making this determination would have been very difficult and would have involved inspecting output files record by record – a very labor-intensive process. Running the instrumented program might reveal that 30% or 40% of a program was actually executed (specific code segments executed were reported). Naturally, all segments of the program were not expected to run for all input test data sets, as the program logic accommodated. But for specific sets of inputs, key parts of the programs were expected to run. This is a great example of dynamic testing - run the program and monitor its run time behavior. Testing such programs dynamically pointed out serious defects (design issues implemented in code) causing errors (production of an incorrect state); the fault was the cause of the error (logical design resulting from poor design) and the resulting failure arose from the resultant behavior (manifestation of the fault(s) through reports generated by summary data produced by the instrumented program executed upon program completion).

(The source code 'probes' are merely integer counters in a single array. Each programming construct (function, paragraph, method, etc.) was instrumented to add 1 to a counter in the array that was associated with that construct. Upon conclusion of the program, the value of each array element represented the total number of times that construct was executed, ranging from zero to a higher number. A function was appended to the program and was executed just prior to normal program termination. This code accessed the array and displayed the numbers of times each programming construct was executed.)

Most modern IDEs offer the ability to monitor variables and their changing values during runtime. Students using Eclipse, NetBeans, or a number of other popular IDEs are familiar with these features that can track program execution allowing one to step through a program one statement at a time and observe how the values of attributes change. These are further examples of dynamic testing and support Stutzke's contention that dynamic analysis is the process of "…operating a system or component under controlled conditions to collect measurements to determine and evaluate the characteristics and performance of the system or component." (Stutzke, 2005)

Black-Box and White-Box Testing

Another grouping of test categories, not mutually exclusive from static and dynamic testing, is black-box and white-box testing. When creating test cases, various sources need to be considered such as specifications captured, perhaps, from use cases or user stories, design documents captured in structure charts or UML diagrams, and actual source code or pseudo-code, captured in a wide range of IDEs. Also, there is available documentation.

Pressman sums up the differences between black-box and white-box testing rather nicely: "Any engineering product (and most other things) can be tested in one of two ways: 1) knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational, at the same time searching for errors in each function; 2) knowing the internal

workings of a product, tests can be conducted to ensure that 'all gears mesh.', that is, that internal operation performs according to specification and all internal components have been adequately exercised. The first test approach is called *black-box testing* and the second, *white-box testing."* (Pressman, 1997)

White-box Testing
(Sometimes called structural or glass-box) testing is done through examination and knowledge of source code. White-box testing examines execution flow through algorithms via 'coverage measures' such as examination of statement coverage, path coverage and branch coverage investigations. White-box testing, in its many forms, monitors the internals of a program and tracks and determines 'how' the program executes, how much of the code is being exercised. Also considered is how many tests through an algorithm are necessary to assure a minimal or acceptable level of testing, what constitutes a minimal set of tests needed to assure a high level of reliability, how 'robust' the program must be and similar tests.

White-box testing considers many program / system execution characteristics. Consider this more closely. Recognizing that one can never assert that a program is error-free, white-box testing addresses factors such as how many edits need to be included in the code to assure an acceptable level of reliability? In particular, is the program one that deals with safety-critical applications, aircraft or weaponry instrumentation, financial systems, or health systems? How much code must be added and tested to assure acceptable levels of reliability and how much reliability is really needed? These are a few of the factors whose answers are used to determine the degree to which edits and other checks are included in both the design and implementation to achieve desired levels of reliability, robustness, and fault tolerance. These are all execution time tests and are verified during run time.

In white-box testing, there needs to be some assurance that code that must be executed is indeed being executed via tests with specific input data. In a way, it is close to but involves both static and dynamic testing. In dynamic testing, test results can point out programming anomalies or areas not executed or time spent in program components (perhaps implying that these are candidates for optimization). But white-box testing (in the coding sense) goes deeply into the internals of the program, to the code itself. The testing yields significant analyses citing statements executed or branches not taken, or execution paths not executed and similar low level information to the developer. The critical thinking is that white-box testing involves the detailed execution analysis of the program's guts; that is, statements, branches, paths, function calls, method calls, and more.
While dynamic testing is used to collect measurements and evaluate characteristics and performance of a component, and can be seen as part of validation, white-box testing, on the other hand, is at the lowest level and is needed for the developers (analyst and programmers) to consider in assuring effective dynamic testing.

Black Box Testing
In contrast to white-box testing is black-box testing, sometimes referred to as end-user testing. In black-box testing, the internals of program execution are not an issue; rather, key concerns center on the production of the correct output given specific inputs. Are the results timely—and accurate? And are all of the requirements accommodated?

In black-box testing, the program is viewed as a black box. The program must read in the inputs, process the data, and check the outputs. While this sounds simple, it is not. Certainly running the test is easy, but the design of suitable test cases may well be an onerous task as a host of carefully designed sets of tests must be generated, oftentimes including boundary testing, stress testing, regression testing, functional testing, and other related black-box testing issues. All of these tests are designed to determine if the application produces the correct outputs given a variety of inputs that exercise / test both the functional and non-functional requirements (Kulak and Guiney, 2004).

Testing requires both functionality (outputs produced given inputs) and non-functional testing (system loading, reliability, robustness, scalability, portability, maintainability, security and more. Black-box testing is often done as part of validation by end users, hence the reason for it sometimes being referred to as end-user testing.

Black-box testing is done without knowledge of the internal workings of code (Turner and Robson, 1993) Instead test cases are derived from the specifications or design or any other documentation that implies functionality. In this

way, black-box testing is only concerned with what can be generated from running the application. Defects are often discovered in black-box testing and may be traced back to design issues or perhaps implementation issues. Failures (behavioral issues; the producing of unintended results) may also be readily observed via black-box testing. In contrast, the cause(s) of the error (fault) and the producing of an incorrect system state (error) are more typically discovered via white-box testing.

Verification, Validation and Acceptance Testing

These tests reflect still another category of testing – again, not mutually exclusive of static and dynamic testing and black-box and white-box testing - using terms often in common use with different stakeholders. Verification is often combined with another software engineering concept known as validation. These are two different types of testing with different goals in mind, unlike static and dynamic testing which both seek to find faults in code and defects in design on the development side.

Stutzke sums up the differences between verification and validation. He says that verification deals with evaluation of products in a given [development] activity "to determine both correctness and consistency with respect to the products and standards provided as input to that specific activity." Verification ensures that "you have built it right." In contrast, validation confirms that the product, as provided (or as it will be provided) will fulfill its intended use. Validation ensures that "you built the right thing." (Stuzke, 2005) In more detail, consider the following elaboration of these definitions.

Verification testing is the pursuit of establishing that a particular phase of a software system has satisfied the requirements which had been decided upon before embarking on that phase (Naik and Tripathy, 2013) Thus, verification testing is typically white-box testing but may also include black-box testing. Essentially, verification is done by the developers or maintainers of software to ensure that the software meets requirements This is often the activity undertaken by software developers typically during unit testing. It follows from this that although verification testing is generally white-box testing, clearly the developer is interested in producing correct outputs given specific inputs. Specifically, the product is built right.

Validation testing is done to assure that software meets the needs of those who intend to use it (Naik and Tripathy, 2013). Validation testing is, thus, often black-box testing and is concerned with ensuring functionality. Validation testing provides the customer confidence that the software system is adequate for its intended use. Essentially successful validation testing provides assurance to the user that their expectations have been met. Customers typically undertake validation exercises to ensure the right thing was built.

While verification testing is used to eliminate defects and faults that cause error states and visible failures, validation testing shows that there are no failures. Stated equivalently, in verification: programmer runs unit tests against specifications and eliminates defects and faults causing error states and visible failures; in validation: end user runs tests to determine if specific inputs result in specific outputs. Clients / end-users run tests to ensure no failures are experienced.

One sometimes sees the term, acceptance testing and acceptance criteria. Acceptance criteria are often defined by the designers in the hopes that satisfying the criteria adequately demonstrates to the user that their needs have been met. Also acceptance testing is designed to help the end-user gain confidence in the code.

4. CONCLUSIONS

The paper has provided definitions of fault, errors, failures and defects with specific examples to provide clarity in their use. While the paper did not propose a study to verify the approaches offered by researchers in the literature review, value lies in establishing a solid basis of definition and use of these commonly misunderstood and misused key definitions both in the workplace and in the classroom. Practitioners and students must use precise definitions when referring to defects, errors, faults, and failures.

The authors have also applied these terms to three major categories of testing: static and dynamic testing, white-box and black-box testing, and verification, validation, and acceptance testing. While there are other categories of testing that are often unique to specific software development methodologies, most of these categories can easily fit within a

framework of the three testing categories provided.

5. REFERENCES

Favaro, Francesca, M., David Jackson, and Joseph Saleh, *Software Contributions to Aircraft Adverse Events: Case Studies and Analyses of Recurrent Accident Patterns and Failure Mechanisms*, Reliability Engineering & System Safety 113, May 2013.

Fowler, Martin, contributions by: Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison- Wesley, 2012, ISBN: 0-201-48567-2.

Galin, Daniel, *Software Quality Assurance,* Pearson / Addison Wesley, 2004 (ISBN 0-201-70945-7), Chapter 2, What is Software Quality?

Juran, J.M. and A. Blanton Godfrey, *Juran's Quality Control Handbook,* 5th edition, McGraw-Hill, New York ISBN-13: 978-0071165396, 2000.

Kulak, Daryl and Eamonn Guiney, *Use Cases: Requirements in Context,* Addison Wesley, 2004, ISBN: 0-321-15498-3.

Morris S. Johnson, Jr., "A Survey of Testing Techniques for Object-Oriented Systems," Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '96).

Naik, K., & Tripathy, P., *Software Testing And Quality Assurance: Theory And Practice*, John Wiley & Sons, 2008. p. 7-27.

Pressman, Roger, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1997, ISBN 0-07-052182-4.

Schulmeyer, C. Gordon and Garth R. MacKenzie, *Verification & Validation of Modern Software –Intensive Systems*, Prentice-Hall PTR, 2000 ISBN: 0-13-020584-2.

Stutzke, Richard D., Estimating Software-Intensive Systems, Pearson Education Inc., 2005 ISBN 0-201-70312-2.

Turner, C.D.; Robson, D.J., "The State-Based Testing of Object-Oriented Programming Conference on Software Maintenance, CSM-93, Proceedings, 1993 ISBN 0- 8186-4600-4.

Walia, Gursimran S., and Jeffrey C. Carver, *Using Error Abstraction and Classification to Improve Requirement Quality: Conclusions from a Family of Four Empirical Studies,* Springer Science + Business Media, LLC 2012.