Software testing

Software testing is the act of examining the artifacts and the behavior of the software under test by validation and verification. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but not necessarily limited to:

- analyzing the product requirements for completeness and correctness in various contexts like industry perspective, business perspective, feasibility and viability of implementation, usability, performance, security, infrastructure considerations, etc.
- reviewing the product architecture and the overall design of the product
- working with product developers on improvement in coding techniques, design patterns, tests that can be written as part of code based on various techniques like boundary conditions, etc.
- executing a program or application with the intent of examining behavior
- reviewing the deployment infrastructure and associated scripts & automation
- take part in production activities by using monitoring & observability techniques

Software testing can provide objective, independent information about the quality of software and risk of its failure to users or sponsors.[1]

Contents

Overview

Although software testing can determine the correctness of software under the assumption of some specific hypotheses (see the hierarchy of testing difficulty below), testing cannot identify all the failures within the software.[2] Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against test oracles — principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts,[3] comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions, but only that it does not function properly under specific conditions.[4] The scope of software testing may include the examination of code as well as the execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.[5]:41–43

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing assists in making this assessment.

Faults and failures

Software faults occur through the following process: A programmer makes an error (mistake), which results in a fault (defect, bug) in the software source code. If this fault is executed, in certain situations the system will produce wrong results, causing a failure.[6]:31

Not all faults will necessarily result in failures. For example, faults in the dead code will never result in failures. A fault that did not reveal failures may result in a failure when the environment is changed. Examples of these changes in environment include the software being run on a new computer hardware platform, alterations in source data, or interacting with different software.[7] A single fault may result in a wide range of failure symptoms.

Not all software faults are caused by coding errors. One common source of expensive defects is requirement gaps, i.e., unrecognized requirements that result in errors of omission by the program designer.[5]:426 Requirement gaps can often be non-functional requirements such as testability, scalability, maintainability, performance, and security.

Input combinations and preconditions

A fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product.[4]:17–18[8] This means that the number of faults in a software product can be very large and defects that occur infrequently are difficult to find in testing and debugging. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*) — usability, scalability, performance, compatibility, and reliability — can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Software developers can't test everything, but they can use combinatorial test design to identify the minimum number of tests needed to get the coverage they want. Combinatorial test design enables users to get greater test coverage with fewer tests. Whether they are looking for speed or test depth, they can use combinatorial test design methods to build structured variation into their test cases.[9]

Economics

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided, if better software testing was performed.[10]

Outsourcing software testing because of costs is very common, with China, the Philippines, and India being preferred destinations.[11]

Roles

Software testing can be done by dedicated software testers; until the 1980s, the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing,[12] different roles have been established, such as *test manager*, *test lead*, *test analyst*, *test designer*, *tester*, *automation developer*, and *test administrator*. Software testing can also be performed by non-dedicated software testers.[13]

History

Glenford J. Myers initially introduced the separation of debugging from testing in 1979.[14] Although his attention was on breakage testing ("A successful test case is one that detects an as-yet undiscovered error."[14]:16), it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification.

Testing approach

Static, dynamic, and passive testing

There are many approaches available in software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas executing programmed code with a given set of test cases is referred to as dynamic testing.[15][16]

Static testing is often implicit, like proofreading, plus when programming tools/text editors check source code structure or compilers (pre-compilers) check syntax and data flow as static program analysis. Dynamic testing takes place when the program itself is run. Dynamic testing may begin before the program is 100% complete in order to test particular sections of code and are applied to discrete functions or modules.[15][16] Typical techniques for these are either using stubs/drivers or execution from a debugger environment.[16]

Static testing involves verification, whereas dynamic testing also involves validation.[16]

Passive testing means verifying the system behavior without any interaction with the software product. Contrary to active testing, testers do not provide any test data but look at system logs and traces. They mine for patterns and specific behavior in order to make some kind of decisions.[17] This is related to offline runtime verification and log analysis.

Exploratory approach

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design, and test execution. Cem Kaner, who coined the term in 1984,[18]:2 defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."[18]:36

The "box" approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that the tester takes when designing test cases. A hybrid approach called grey-box testing may also be applied to software testing methodology.[19][20] With the concept of grey-box testing—which develops tests from specific design elements—gaining prominence, this "arbitrary distinction" between black- and white-box testing has faded somewhat.[21]

White-box testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) verifies the internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing, an internal perspective of the system (the source code), as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs.[19][20] This is analogous to testing nodes in a circuit, e.g., in-circuit testing (ICT).

White Box Testing Diagram

While white-box testing can be applied at the unit, integration, and system levels of the software testing process, it is usually done at the unit level.[21] It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

Techniques used in white-box testing include:[20][22]

- API testing – testing of the application using public and private APIs (application programming interfaces)
- Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
- Mutation testing methods
- Static testing methods

Code coverage tools can evaluate the completeness of a test suite that was created with any method, including black-box testing. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.[23] Code coverage as a

software metric can be reported as a percentage for:[19][23][24]

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test
- *Decision coverage*, which reports on whether both the True and the False branch of a given test has been executed

100% statement coverage ensures that all code paths or branches (in terms of control flow) are executed at least once. This is helpful in ensuring correct functionality, but not sufficient since the same code may process different inputs correctly or incorrectly.[25] Pseudo-tested functions and methods are those that are covered but not specified (it is possible to remove their body without breaking any test case).[26]

Black-box testing

Black box diagram

Black-box testing (also known as functional testing) treats the software as a "black box," examining functionality without any knowledge of internal implementation, without seeing the source code. The testers are only aware of what the software is supposed to do, not how it does it.[27] Black-box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory testing, and specification-based testing.[19][20][24]

Specification-based testing aims to test the functionality of software according to the applicable requirements.[28] This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional, though usually functional.

Specification-based testing may be necessary to assure correct functionality, but it is insufficient to guard against complex or high-risk situations.[29]

One advantage of the black box technique is that no programming knowledge is required. Whatever biases the programmers may have had, the tester likely has a different set and may emphasize different areas of functionality. On the other hand, black-box testing has been said to be "like a walk in a dark labyrinth without a flashlight."[30] Because they do not examine the source code, there are situations when a tester writes many test cases to check something that could have been tested by only one test case or leaves some parts of the program untested.

This method of test can be applied to all levels of software testing: unit, integration, system and acceptance.[21] It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

Component interface testing

Component interface testing is a variation of black-box testing, with the focus on the data values beyond just the related actions of a subsystem component.[31] The practice of component interface testing can be used to check the handling of data passed between various units, or subsystem components, beyond full integration testing between those units.[32][33] The data being passed can be considered as "message

packets" and the range or data types can be checked, for data generated from one unit, and tested for validity before being passed into another unit. One option for interface testing is to keep a separate log file of data items being passed, often with a timestamp logged to allow analysis of thousands of cases of data passed between units for days or weeks. Tests can include checking the handling of some extreme data values while other interface variables are passed as normal values.[32] Unusual data values in an interface can help explain unexpected performance in the next unit.

Visual testing

The aim of visual testing is to provide developers with the ability to examine what was happening at the point of software failure by presenting the data in such a way that the developer can easily find the information she or he requires, and the information is expressed clearly.[34][35]

At the core of visual testing is the idea that showing someone a problem (or a test failure), rather than just describing it, greatly increases clarity and understanding. Visual testing, therefore, requires the recording of the entire test process – capturing everything that occurs on the test system in video format. Output videos are supplemented by real-time tester input via picture-in-a-picture webcam and audio commentary from microphones.

Visual testing provides a number of advantages. The quality of communication is increased drastically because testers can show the problem (and the events leading up to it) to the developer as opposed to just describing it and the need to replicate test failures will cease to exist in many cases. The developer will have all the evidence she or he requires of a test failure and can instead focus on the cause of the fault and how it should be fixed.

Ad hoc testing and exploratory testing are important methodologies for checking software integrity, because they require less preparation time to implement, while the important bugs can be found quickly.[36] In ad hoc testing, where testing takes place in an improvised, impromptu way, the ability of the tester(s) to base testing off documented methods and then improvise variations of those tests can result in more rigorous examination of defect fixes.[36] However, unless strict documentation of the procedures are maintained, one of the limits of ad hoc testing is lack of repeatability.[36]

Grey-box testing

Grey-box testing (American spelling: gray-box testing) involves having knowledge of internal data structures and algorithms for purposes of designing tests while executing those tests at the user, or black-box level. The tester will often have access to both "the source code and the executable binary."[37] Grey-box testing may also include reverse engineering (using dynamic code analysis) to determine, for instance, boundary values or error messages.[37] Manipulating input data and formatting output do not qualify as grey-box, as the input and output are clearly outside of the "black box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for the test.

By knowing the underlying concepts of how the software works, the tester makes better-informed testing choices while testing the software from outside. Typically, a grey-box tester will be permitted to set up an isolated testing environment with activities such as seeding a database. The tester can observe the state of the product being tested after performing certain actions such as executing SQL statements against the database and then executing queries to ensure that the expected changes have been reflected. Grey-box testing implements intelligent test scenarios, based on limited information. This will particularly apply to data type handling, exception handling, and so on.[38]

Testing levels

Broadly speaking, there are at least three levels of testing: unit testing, integration testing, and system testing.[39][40][41][42] However, a fourth level, acceptance testing, may be included by developers. This may be in the form of operational acceptance testing or be simple end-user (beta) testing, testing to ensure the software meets functional expectations.[43][44][45] Based on the ISTQB Certified Test Foundation Level syllabus, test levels includes those four levels, and the fourth level is named acceptance testing.[46] Tests are frequently grouped into one of these levels by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.[47]

These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Unit testing is a software development process that involves a synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development life cycle. Unit testing aims to eliminate construction errors before code is promoted to additional testing; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

Depending on the organization's expectations for software development, unit testing might include static code analysis, data-flow analysis, metrics analysis, peer code reviews, code coverage analysis and other software testing practices.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.[48]

Integration tests usually involve a lot of code, and produce traces that are larger than those produced by unit tests. This has an impact on the ease of localizing the fault when an integration test fails. To overcome this issue, it has been proposed to automatically cut the large tests in smaller pieces to improve fault localization.[49]

System testing

System testing tests a completely integrated system to verify that the system meets its requirements.[6]:74 For example, a system test might involve testing a login interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

Acceptance testing

Commonly this level of Acceptance testing include the following four types:[46]

- User acceptance testing
- Operational acceptance testing
- Contractual and regulatory acceptance testing
- Alpha and beta testing

User acceptance testing and Alpha and beta testing are described in the next testing types section.

Operational acceptance is used to conduct operational readiness (pre-release) of a product, service or system as part of a quality management system. OAT is a common type of non-functional software testing, used mainly in software development and software maintenance projects. This type of testing focuses on the operational readiness of the system to be supported, or to become part of the production environment. Hence, it is also known as operational readiness testing (ORT) or Operations readiness and assurance (OR&A) testing. Functional testing within OAT is limited to those tests that are required to verify the *non-functional* aspects of the system.

In addition, the software testing should ensure that the portability of the system, as well as working as expected, does not also damage or partially corrupt its operating environment or cause other processes within that environment to become inoperative.[50]

Contractual acceptance testing is performed based on the contract's acceptance criteria defined during the agreement of the contract, while regulatory acceptance testing is performed based on the relevant regulations to the software product. Both of these two testings can be performed by users or independent testers. Regulation acceptance testing sometimes involves the regulatory agencies auditing the test results.[46]

Testing types, techniques and tactics

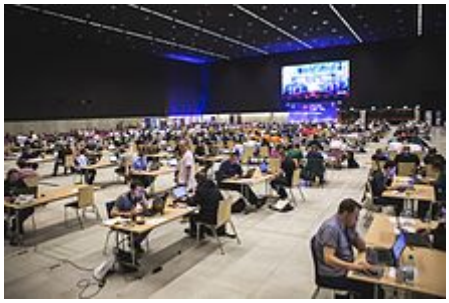
Different labels and ways of grouping testing may be testing types, software testing tactics or techniques.[51]

Installation testing

Most software systems have installation procedures that are needed before they can be used for their main purpose. Testing these procedures to achieve an installed software system that may be used is known as installation testing.

Compatibility testing

A common cause of software failure (real or perceived) is a lack of its compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a Web application, which must render in a Web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment were capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.



TestingCup - Polish Championship in Software Testing, Katowice, May 2016

Smoke and sanity testing

Sanity testing determines whether it is reasonable to proceed with further testing.

Smoke testing consists of minimal attempts to operate the software, designed to determine whether there are any basic problems that will prevent it from working at all. Such tests can be used as build verification test.

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, as degraded or lost features, including old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly, stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Regression testing is typically the largest test effort in commercial software development,[52] due to checking numerous details in prior software features, and even new software can be developed while using some old test cases to test parts of the new design to ensure prior functionality is still supported.

Common methods of regression testing include re-running previous sets of test cases and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, or be very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk. In regression testing, it is important to have strong assertions on the existing behavior. For this, it is possible to generate and add new assertions in existing test cases,[53] this is known as automatic test amplification.[54]

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as a build acceptance test prior to further testing, e.g., before integration or regression.
2. Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be

performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing before the software goes to beta testing.[55]

Beta testing

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team known as beta testers. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Beta versions can be made available to the open public to increase the feedback field to a maximal number of future users and to deliver value earlier, for an extended or even indefinite period of time (perpetual beta).[56]

Functional vs non-functional testing

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work."

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security. Testing will determine the breaking point, the point at which extremes of scalability or performance leads to unstable execution. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

Continuous testing

Continuous testing is the process of executing automated tests as part of the software delivery pipeline to obtain immediate feedback on the business risks associated with a software release candidate.[57][58] Continuous testing includes the validation of both functional requirements and non-functional requirements; the scope of testing extends from validating bottom-up requirements or user stories to assessing the system requirements associated with overarching business goals.[59][60]

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail. It verifies that the software functions properly even when it receives invalid or unexpected inputs, thereby establishing the robustness of input validation and error-management routines. Software fault injection, in the form of fuzzing, is an example of failure testing. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform destructive testing.

Software performance testing

Performance testing is generally executed to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage.

Load testing is primarily concerned with testing that the system can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*. *Volume testing* is a way to test software functions even when certain components (for example a file or database) increase radically in size. *Stress testing* is a way to test reliability under unexpected or rare workloads. *Stability testing* (often referred to as load or endurance testing) checks to see if the software can continuously function well in or above an acceptable period.

There is little agreement on what the specific goals of performance testing are. The terms load testing, performance testing, scalability testing, and volume testing, are often used interchangeably.

Real-time software systems have strict timing constraints. To test if timing constraints are met, real-time testing is used.

Usability testing

Usability testing is to check if the user interface is easy to use and understand. It is concerned mainly with the use of the application. This is not a kind of testing that can be automated; actual human users are needed, being monitored by skilled UI designers.

Accessibility testing

Accessibility testing may include compliance with standards such as:

- Americans with Disabilities Act of 1990
- Section 508 Amendment to the Rehabilitation Act of 1973
- Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C)

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

The International Organization for Standardization (ISO) defines this as a "type of testing conducted to evaluate the degree to which a test item, and associated data and information, are protected so that unauthorised persons or systems cannot use, read or modify them, and authorized persons or systems are not denied access to them."[61]

Internationalization and localization

Testing for internationalization and localization validates that the software can be used with different languages and geographic regions. The process of pseudolocalization is used to test the ability of an application to be translated to another language, and make it easier to identify when the localization process

may introduce new bugs into the product.

Globalization testing verifies that the software is adapted for a new culture (such as different currencies or time zones).[62]

Actual translation to human languages must be tested, too. Possible localization and globalization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- Technical terminology may become inconsistent, if the project is translated by several people without proper coordination or if the translator is imprudent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically at run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut that has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes that are appropriate in the source language may be inappropriate in the target language; for example, CJK characters may become unreadable, if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to crash or malfunction.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that was not localized.
- Localized operating systems may have differently named system configuration files and environment variables and different formats for date and currency.

Development testing

Development Testing is a software development process that involves the synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development lifecycle. Development Testing aims to eliminate construction errors before code is promoted to other testing; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

Depending on the organization's expectations for software development, Development Testing might include static code analysis, data flow analysis, metrics analysis, peer code reviews, unit testing, code coverage analysis, traceability, and other software testing practices.

A/B testing

A/B testing is a method of running a controlled experiment to determine if a proposed change is more effective than the current approach. Customers are routed to either a current version (control) of a feature, or to a modified version (treatment) and data is collected to determine which version is better at achieving the desired outcome.

Concurrent testing

Concurrent or concurrency testing assesses the behaviour and performance of software and systems that use concurrent computing, generally under normal usage conditions. Typical problems this type of testing will expose are deadlocks, race conditions and problems with shared memory/resource handling.

Conformance testing or type testing

In software testing, conformance testing verifies that a product performs according to its specified standards. Compilers, for instance, are extensively tested to determine whether they meet the recognized standard for that language.

Output comparison testing

Creating a display expected output, whether as data comparison of text or screenshots of the UI,[4]:195 is sometimes called snapshot testing or Golden Master Testing unlike many other forms of testing, this cannot detect failures automatically and instead requires that a human evaluate the output for inconsistencies.

Property testing

Property testing is a testing technique where, instead of asserting that specific inputs produce specific expected outputs, the practitioner randomly generates many inputs, runs the program on all of them, and asserts the truth of some "property" that should be true for every pair of input and output. For example, every input to a sort function should have the same length as its output. Every output from a sort function should be a monotonically increasing list.

Property testing libraries allow the user to control the strategy by which random inputs are constructed, to ensure coverage of degenerate cases, or inputs featuring specific patterns that are needed to fully exercise aspects of the implementation under test.

Property testing is also sometimes known as "generative testing" or "QuickCheck testing" since it was introduced and popularized by the Haskell library "QuickCheck (https://hackage.haskell.org/package/QuickCheck)."[63]

VCR testing

VCR testing, also known as "playback testing" or "record/replay" testing, is a testing technique for increasing the reliability and speed of regression tests that involve a component that is slow or unreliable to communicate with, often a third-party API outside of the tester's control. It involves making a recording ("cassette") of the system's interactions with the external component, and then replaying the recorded interactions as a substitute for communicating with the external system on subsequent runs of the test.

The technique was popularized in web development by the Ruby library vcr (https://github.com/vcr/vcr).

Testing process

Traditional waterfall development model

A common practice in waterfall development is that testing is performed by an independent group of testers. This can happen:

- after the functionality is developed, but before it is shipped to the customer.[64] This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.[14]:145–146
- at the same moment the development project starts, as a continuous process until the project finishes.[65]

However, even in the waterfall development model, unit testing is often done by the software development team even when further testing is done by a separate team.[66]

Agile or XP development model

In contrast, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). The tests are expected to fail initially. Each failing test is followed by writing just enough code to make it pass.[67] This means the test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).

The ultimate goals of this test process are to support continuous integration and to reduce defect rates.[68][67]

This methodology increases the testing effort done by development, before reaching any formal testing team. In some other development models, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing.[2] The sample below is common among organizations employing the Waterfall development model. The same practices are commonly found in other development models, but might not be as clear or explicit.

- Requirements analysis: Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work to determine what aspects of a design are testable and with what parameters those tests work.
- Test planning: Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- Test development: Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.
- Test execution: Testers execute the software based on the plans and test documents then report any errors found to the development team. This part could be complex when running tests with a lack of programming knowledge.
- Test reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
- Test result analysis: Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be assigned, fixed, rejected (i.e. found

software working properly) or deferred to be dealt with later.
- Defect Retesting: Once a defect has been dealt with by the development team, it is retested by the testing team.
- Regression testing: It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything and that the software product as a whole is still working correctly.
- Test Closure: Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code, including:
  - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
  - Hypervisor, permitting complete control of the execution of program code including:-
  - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
  - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional Graphical User Interface (GUI) testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into a single composite tool or an Integrated Development Environment (IDE).

Capture and replay

Capture and replay consists in collecting end-to-end usage scenario while interacting with an application and in turning these scenarios into test cases. Possible applications of capture and replay include the generation of regression tests. The SCARPE tool [69] selectively captures a subset of the application under

study as it executes. JRapture captures the sequence of interactions between an executing Java program and components on the host system such as files, or events on graphical user interfaces. These sequences can then be replayed for observation-based testing.[70] Saieva et al. propose to generate ad-hoc tests that replay recorded user execution traces in order to test candidate patches for critical security bugs.[71] Pankti collects object profiles in production to generate focused differential unit tests. This tool enhances capture and replay with the systematic generation of derived test oracles.[72]

Measurement in software testing

Quality measures include such topics as correctness, completeness, security and ISO/IEC 9126 requirements such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently used software metrics, or measures, which are used to assist in determining the state of the software or the adequacy of the testing.

Hierarchy of testing difficulty

Based on the number of test cases required to construct a complete test suite in each context (i.e. a test suite such that, if it is applied to the implementation under test, then we collect enough information to precisely determine whether the system is correct or incorrect according to some specification), a hierarchy of testing difficulty has been proposed.[73] [74] It includes the following testability classes:

- Class I: there exists a finite complete test suite.
- Class II: any partial distinguishing rate (i.e., any incomplete capability to distinguish correct systems from incorrect systems) can be reached with a finite test suite.
- Class III: there exists a countable complete test suite.
- Class IV: there exists a complete test suite.
- Class V: all cases.

It has been proved that each class is strictly included in the next. For instance, testing when we assume that the behavior of the implementation under test can be denoted by a deterministic finite-state machine for some known finite sets of inputs and outputs and with some known number of states belongs to Class I (and all subsequent classes). However, if the number of states is not known, then it only belongs to all classes from Class II on. If the implementation under test must be a deterministic finite-state machine failing the specification for a single trace (and its continuations), and its number of states is unknown, then it only belongs to classes from Class III on. Testing temporal machines where transitions are triggered if inputs are produced within some real-bounded interval only belongs to classes from Class IV on, whereas testing many non-deterministic systems only belongs to Class V (but not all, and some even belong to Class I). The inclusion into Class I does not require the simplicity of the assumed computation model, as some testing cases involving implementations written in any programming language, and testing implementations defined as machines depending on continuous magnitudes, have been proved to be in Class I. Other elaborated cases, such as the testing framework by Matthew Hennessy under must semantics, and temporal machines with rational timeouts, belong to Class II.

Testing artifacts

A software testing process can produce several artifacts. The actual artifacts produced are a factor of the software development model used, stakeholder and organisational needs.

Test plan

A test plan is a document detailing the approach that will be taken for intended test activities. The plan may include aspects such as objectives, scope, processes and procedures, personnel requirements, and contingency plans.[43] The test plan could come in the form of a single plan that includes all test types (like an acceptance or system test plan) and planning considerations, or it may be issued as a master test plan that provides an overview of more than one detailed test plan (a plan of a plan).[43] A test plan can be, in some cases, part of a wide "test strategy" which documents overall testing approaches, which may itself be a master test plan or even a separate artifact.

Traceability matrix

A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when related source documents are changed, to select test cases for execution when planning for regression tests by considering requirement coverage.

Test case

A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and the actual result. Clinically defined, a test case is an input and an expected result.[75] This can be as terse as 'for condition x your derived result is y', although normally test cases describe in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repositories. In a database system, you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script

A test script is a procedure or programming code that replicates user actions. Initially, the term was derived from the product of work created by automated regression test tools. A test case will be a baseline to create test scripts using a tool or a program.

Test suite

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test fixture or test data

In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project. There are techniques to generate test data.

Test harness

The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Test run
A report of the results from running a test case or a test suite

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. Note that a few practitioners argue that the testing field is not ready for certification, as mentioned in the controversy section.

Controversy

Some of the major software testing controversies include:

Agile vs. traditional
Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles,[76][77] whereas government and military[78] software providers use this methodology but also the traditional test-last models (e.g., in the Waterfall model).

Manual vs. automated testing
Some writers believe that test automation is so expensive relative to its value that it should be used sparingly.[79] The test automation then can be considered as a way to capture and implement the requirements. As a general rule, the larger the system and the greater the complexity, the greater the ROI in test automation. Also, the investment in tools and expertise can be amortized over multiple projects with the right level of knowledge sharing within an organization.

Is the existence of the ISO 29119 software testing standard justified?
Significant opposition has formed out of the ranks of the context-driven school of software testing about the ISO 29119 standard. Professional testing associations, such as the International Society for Software Testing, have attempted to have the standard withdrawn.[80][81]

Some practitioners declare that the testing field is not ready for certification
[82] No certification now offered actually requires the applicant to show their ability to test software. No certification is based on a widely accepted body of knowledge. Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.[83]

Studies used to show the relative expense of fixing defects
There are opposing views on the applicability of studies used to show the relative expense of fixing defects depending on their introduction and detection. For example:

> It is commonly believed that the earlier a defect is found, the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found.[84] For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review. With the advent of modern continuous deployment practices and cloud-based services, the cost of re-deployment and maintenance may lessen over time.

Cost to fix a defect		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5–10×	10×	10–100×
	Architecture	–	1×	10×	15×	25–100×
	Construction	–	–	1×	10×	10–25×

The data from which this table is extrapolated is scant. Laurent Bossavit says in his analysis:

> The "smaller projects" curve turns out to be from only two teams of first-year students, a sample size so small that extrapolating to "smaller projects in general" is totally indefensible. The GTE study does not explain its data, other than to say it came from two projects, one large and one small. The paper cited for the Bell Labs "Safeguard" project specifically disclaims having collected the fine-grained data that Boehm's data points suggest. The IBM study (Fagan's paper) contains claims that seem to contradict Boehm's graph and no numerical results that clearly correspond to his data points.
>
> Boehm doesn't even cite a paper for the TRW data, except when writing for "Making Software" in 2010, and there he cited the original 1976 article. There exists a large study conducted at TRW at the right time for Boehm to cite it, but that paper doesn't contain the sort of data that would support Boehm's claims.[85]

Related processes

Software verification and validation

Software testing is used in association with verification and validation:[86]

- Verification: Have we built the software right? (i.e., does it implement the requirements).
- Validation: Have we built the right software? (i.e., do the deliverables satisfy the customer).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms defined with contradictory definitions. According to the *IEEE Standard Glossary of Software Engineering Terminology*:[6]:80–81

> Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
> Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

And, according to the ISO 9000 standard:

> Verification is confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.
> Validation is confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

The contradiction is caused by the use of the concepts of requirements and specified requirements but with different meanings.

In the case of IEEE standards, the specified requirements, mentioned in the definition of validation, are the set of problems, needs and wants of the stakeholders that the software must solve and satisfy. Such requirements are documented in a Software Requirements Specification (SRS). And, the products mentioned in the definition of verification, are the output artifacts of every phase of the software development process. These products are, in fact, specifications such as Architectural Design Specification, Detailed Design Specification, etc. The SRS is also a specification, but it cannot be verified (at least not in the sense used here, more on this subject below).

But, for the ISO 9000, the specified requirements are the set of specifications, as just mentioned above, that must be verified. A specification, as previously explained, is the product of a software development process phase that receives another specification as input. A specification is verified successfully when it correctly implements its input specification. All the specifications can be verified except the SRS because it is the first one (it can be validated, though). Examples: The Design Specification must implement the SRS; and, the Construction phase artifacts must implement the Design Specification.

So, when these words are defined in common terms, the apparent contradiction disappears.

Both the SRS and the software must be validated. The SRS can be validated statically by consulting with the stakeholders. Nevertheless, running some partial implementation of the software or a prototype of any kind (dynamic testing) and obtaining positive feedback from them, can further increase the certainty that the SRS is correctly formulated. On the other hand, the software, as a final and running product (not its artifacts and documents, including the source code) must be validated dynamically with the stakeholders by executing the software and having them to try it.

Some might argue that, for SRS, the input is the words of stakeholders and, therefore, SRS validation is the same as SRS verification. Thinking this way is not advisable as it only causes more confusion. It is better to think of verification as a process involving a formal and technical input document.

Software quality assurance

Software testing may be considered a part of a software quality assurance (SQA) process.[4]:347 In SQA, software process specialists and auditors are concerned with the software development process rather than just the artifacts such as documentation, code and systems. They examine and change the software engineering process itself to reduce the number of faults that end up in the delivered software: the so-called defect rate. What constitutes an acceptable defect rate depends on the nature of the software; a flight simulator video game would have much higher defect tolerance than software for an actual airplane. Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is an activity to investigate software under test in order to provide quality-related information to stakeholders. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from reaching customers.

See also

- Data validation
- Dynamic program analysis
- Formal verification
- Graphical user interface testing
- Independent test organization
- Manual testing
- Orthogonal array testing
- Pair testing
- Reverse semantic traceability
- Software testing tactics
- Test data generation
- Test management tools
- Trace table
- Web testing

References

1. Kaner, Cem (November 17, 2006). *Exploratory Testing* (http://www.kaner.com/pdfs/ETatQAI.pdf) (PDF). Quality Assurance Institute Worldwide Annual Software Testing Conference. Orlando, FL. Retrieved November 22, 2014.
2. Pan, Jiantao (Spring 1999). "Software Testing" (http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/) (coursework). Carnegie Mellon University. Retrieved November 21, 2017.
3. Leitner, Andreas; Ciupa, Ilinca; Oriol, Manuel; Meyer, Bertrand; Fiva, Arno (September 2007). *Contract Driven Development = Test Driven Development – Writing Test Cases* (http://se.inf.ethz.ch/people/leitner/publications/cdd_leitner_esec_fse_2007.pdf) (PDF). ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007. Dubrovnik, Croatia. Retrieved December 8, 2017.
4. Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc (1999). *Testing Computer Software* (2nd ed.). New York: John Wiley and Sons. ISBN 978-0-471-35846-6.
5. Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html). Wiley-IEEE Computer Society Press. ISBN 978-0-470-04212-0.
6. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990, doi:10.1109/IEEESTD.1990.101064 (https://doi.org/10.1109%2FIEEESTD.1990.101064), ISBN 9781559370677
7. "Certified Tester Foundation Level Syllabus" (https://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html) (pdf). International Software Testing Qualifications Board. March 31, 2011. Section 1.1.2. Retrieved December 15, 2017.
8. "Certified Tester Foundation Level Syllabus" (http://www.bcs.org/upload/pdf/istqbsyll.pdf) (PDF). International Software Testing Qualifications Board. July 1, 2005. Principle 2, Section 1.3. Retrieved December 15, 2017.
9. Ramler, Rudolf; Kopetzky, Theodorich; Platz, Wolfgang (April 17, 2012). *Combinatorial Test Design in the TOSCA Testsuite: Lessons Learned and Practical Implications*. IEEE Fifth International Conference on Software Testing and Validation (ICST). Montreal, QC, Canada. doi:10.1109/ICST.2012.142 (https://doi.org/10.1109%2FICST.2012.142).
10. "The Economic Impacts of Inadequate Infrastructure for Software Testing" (https://www.nist.gov/director/planning/upload/report02-3.pdf) (PDF). National Institute of Standards and Technology. May 2002. Retrieved December 19, 2017.
11. Sharma, Bharadwaj (April 2016). "Ardentia Technologies: Providing Cutting Edge Software Solutions and Comprehensive Testing Services" (http://www.cioreviewindia.com/magazine/Ardentia-Technologies-Providing-Cutting-Edge-Software-Solutions-and-Comprehensive-Testing-Services---JSAH430576969.html). *CIO Review* (India ed.). Retrieved December 20, 2017.
12. Gelperin, David; Hetzel, Bill (June 1, 1988). "The growth of software testing". *Communications of the ACM*. **31** (6): 687–695. doi:10.1145/62959.62965 (https://doi.org/10.1145%2F62959.62965). S2CID 14731341 (https://api.semanticscholar.org/CorpusID:14731341).

13. Gregory, Janet; Crispin, Lisa (2014). *More Agile Testing*. Addison-Wesley Professional. pp. 23–39. ISBN 9780133749564.
14. Myers, Glenford J. (1979). *The Art of Software Testing* (https://archive.org/details/artofsoftwaretes00myer). John Wiley and Sons. ISBN 978-0-471-04328-7.
15. Graham, D.; Van Veenendaal, E.; Evans, I. (2008). *Foundations of Software Testing* (https://books.google.com/books?id=Ss62LSqCa1MC&pg=PA57). Cengage Learning. pp. 57–58. ISBN 9781844809899.
16. Oberkampf, W.L.; Roy, C.J. (2010). *Verification and Validation in Scientific Computing* (https://books.google.com/books?id=7d26zLEJ1FUC&pg=PA155). Cambridge University Press. pp. 154–5. ISBN 9781139491761.
17. Lee, D.; Netravali, A.N.; Sabnani, K.K.; Sugla, B.; John, A. (1997). "Passive testing and applications to network management". *Proceedings 1997 International Conference on Network Protocols*. IEEE Comput. Soc: 113–122. doi:10.1109/icnp.1997.643699 (https://doi.org/10.1109%2Ficnp.1997.643699). ISBN 081868061X. S2CID 42596126 (https://api.semanticscholar.org/CorpusID:42596126).
18. Cem Kaner (2008), *A Tutorial in Exploratory Testing* (http://www.kaner.com/pdfs/QAIExploring.pdf) (PDF)
19. Limaye, M.G. (2009). *Software Testing* (https://books.google.com/books?id=zUm8My7SiakC&pg=PA108). Tata McGraw-Hill Education. pp. 108–11. ISBN 9780070139909.
20. Saleh, K.A. (2009). *Software Engineering* (https://books.google.com/books?id=N69KPjBEWygC&pg=PA224). J. Ross Publishing. pp. 224–41. ISBN 9781932159943.
21. Ammann, P.; Offutt, J. (2016). *Introduction to Software Testing* (https://books.google.com/books?id=58LeDQAAQBAJ&pg=PA26). Cambridge University Press. p. 26. ISBN 9781316773123.
22. Everatt, G.D.; McLeod Jr., R. (2007). "Chapter 7: Functional Testing". *Software Testing: Testing Across the Entire Software Development Life Cycle*. John Wiley & Sons. pp. 99–121. ISBN 9780470146347.
23. Cornett, Steve (c. 1996). "Code Coverage Analysis" (http://www.bullseye.com/coverage.html#intro). Bullseye Testing Technology. Introduction. Retrieved November 21, 2017.
24. Black, R. (2011). *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional* (https://books.google.com/books?id=n-bTHNW97kYC&pg=PA44). John Wiley & Sons. pp. 44–6. ISBN 9781118079386.
25. As a simple example, the C function `int f(int x){return x*x-6*x+8;}` consists of only one statement. All tests against a specification `f(x)>=0` will succeed, except if `x=3` happens to be chosen.
26. Vera-Pérez, Oscar Luis; Danglot, Benjamin; Monperrus, Martin; Baudry, Benoit (2018). "A comprehensive study of pseudo-tested methods" (https://hal.archives-ouvertes.fr/hal-01867423/document). *Empirical Software Engineering*. **24** (3): 1195–1225. arXiv:1807.05030 (https://arxiv.org/abs/1807.05030). Bibcode:2018arXiv180705030V (https://ui.adsabs.harvard.edu/abs/2018arXiv180705030V). doi:10.1007/s10664-018-9653-2 (https://doi.org/10.1007%2Fs10664-018-9653-2). S2CID 49744829 (https://api.semanticscholar.org/CorpusID:49744829).
27. Patton, Ron (2005). *Software Testing* (https://archive.org/details/softwaretesting0000patt) (2nd ed.). Indianapolis: Sams Publishing. ISBN 978-0672327988.
28. Laycock, Gilbert T. (1993). *The Theory and Practice of Specification Based Software Testing* (http://www.cs.le.ac.uk/people/glaycock/thesis.pdf) (PDF) (dissertation). Department of Computer Science, University of Sheffield. Retrieved January 2, 2018.
29. Bach, James (June 1999). "Risk and Requirements-Based Testing" (http://www.satisfice.com/articles/requirements_based_testing.pdf) (PDF). *Computer*. **32** (6): 113–114. Retrieved August 19, 2008.

30. Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
31. Mathur, A.P. (2011). *Foundations of Software Testing* (https://books.google.com/books?id=hyaQobu44xUC&pg=PA18). Pearson Education India. p. 63. ISBN 9788131759080.
32. Clapp, Judith A. (1995). *Software Quality Control, Error Analysis, and Testing* (https://books.google.com/books?id=wAq0rnyiGMEC&pg=PA313). p. 313. ISBN 978-0815513636. Retrieved January 5, 2018.
33. Mathur, Aditya P. (2007). *Foundations of Software Testing* (https://books.google.com/books?id=yU-rTcurys8C&pg=PR38). Pearson Education India. p. 18. ISBN 978-8131716601.
34. Lönnberg, Jan (October 7, 2003). *Visual testing of software* (http://www.cs.hut.fi/~jlonnber/VisualTesting.pdf) (PDF) (MSc). Helsinki University of Technology. Retrieved January 13, 2012.
35. Chima, Raspal. "Visual testing" (https://web.archive.org/web/20120724162657/http://www.testmagazine.co.uk/2011/04/visual-testing/). *TEST Magazine*. Archived from the original (http://www.testmagazine.co.uk/2011/04/visual-testing) on July 24, 2012. Retrieved January 13, 2012.
36. Lewis, W.E. (2016). *Software Testing and Continuous Quality Improvement* (https://books.google.com/books?id=fgaBDd0TfT8C&pg=PA68) (3rd ed.). CRC Press. pp. 68–73. ISBN 9781439834367.
37. Ransome, J.; Misra, A. (2013). *Core Software Security: Security at the Source* (https://books.google.com/books?id=MX5cAgAAQBAJ&pg=PA140). CRC Press. pp. 140–3. ISBN 9781466560956.
38. "SOA Testing Tools for Black, White and Gray Box" (https://web.archive.org/web/20181001010542/http://www.crosschecknet.com:80/soa_testing_black_white_gray_box.php) (white paper). Crosscheck Networks. Archived from the original (http://www.crosschecknet.com/soa_testing_black_white_gray_box.php) on October 1, 2018. Retrieved December 10, 2012.
39. Bourque, Pierre; Fairley, Richard E., eds. (2014). "Chapter 5" (https://www.computer.org/web/swebok/v3). *Guide to the Software Engineering Body of Knowledge*. 3.0. IEEE Computer Society. ISBN 978-0-7695-5166-1. Retrieved January 2, 2018.
40. Bourque, P.; Fairley, R.D., eds. (2014). "Chapter 4: Software Testing" (http://www4.ncsu.edu/~tjmenzie/cs510/pdf/SWEBOKv3.pdf) (PDF). *SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge*. IEEE. pp. 4–1–4–17. ISBN 9780769551661. Retrieved July 13, 2018.
41. Dooley, J. (2011). *Software Development and Professional Practice* (https://books.google.com/books?id=iOqP9_6w-18C&pg=PA193). APress. pp. 193–4. ISBN 9781430238010.
42. Wiegers, K. (2013). *Creating a Software Engineering Culture* (https://books.google.com/books?id=uVsUAAAAQBAJ&pg=PA212). Addison-Wesley. pp. 211–2. ISBN 9780133489293.
43. Lewis, W.E. (2016). *Software Testing and Continuous Quality Improvement* (https://books.google.com/books?id=fgaBDd0TfT8C&pg=PA92) (3rd ed.). CRC Press. pp. 92–6. ISBN 9781439834367.
44. Machado, P.; Vincenzi, A.; Maldonado, J.C. (2010). "Chapter 1: Software Testing: An Overview" (https://books.google.com/books?id=ZOHrm02GFCEC&pg=PA13). In Borba, P.; Cavalcanti, A.; Sampaio, A.; Woodcook, J. (eds.). *Testing Techniques in Software Engineering*. Springer Science & Business Media. pp. 13–14. ISBN 9783642143342.
45. Clapp, J.A.; Stanten, S.F.; Peng, W.W.; et al. (1995). *Software Quality Control, Error Analysis, and Testing* (https://books.google.com/books?id=wAq0rnyiGMEC&pg=PA254). Nova Data Corporation. p. 254. ISBN 978-0815513636.
46. "ISTQB CTFL Syllabus 2018" (https://www.istqb.org/component/jdownloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html). *ISTQB - International Software Testing Qualifications Board*.

47. Binder, Robert V. (1999). *Testing Object-Oriented Systems: Objects, Patterns, and Tools* (https://archive.org/details/testingobjectori00bind/page/45). Addison-Wesley Professional. p. 45 (https://archive.org/details/testingobjectori00bind/page/45). ISBN 978-0-201-80938-1.
48. Beizer, Boris (1990). *Software Testing Techniques* (Second ed.). New York: Van Nostrand Reinhold. pp. 21, 430. ISBN 978-0-442-20672-7.
49. Xuan, Jifeng; Monperrus, Martin (2014). "Test case purification for improving fault localization". *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*: 52–63. arXiv:1409.3176 (https://arxiv.org/abs/1409.3176). Bibcode:2014arXiv1409.3176X (https://ui.adsabs.harvard.edu/abs/2014arXiv1409.3176X). doi:10.1145/2635868.2635906 (https://doi.org/10.1145%2F2635868.2635906). ISBN 9781450330565. S2CID 14540841 (https://api.semanticscholar.org/CorpusID:14540841).
50. Woods, Anthony J. (June 5, 2015). "Operational Acceptance – an application of the ISO 29119 Software Testing standard" (https://www.scribd.com/document/257086897/Operational-Acceptance-Test-White-Paper-2015-Capgemini) (Whitepaper). Capgemini Australia. Retrieved January 9, 2018.
51. Kaner, Cem; Bach, James; Pettichord, Bret (2001). *Lessons Learned in Software Testing: A Context-Driven Approach* (https://archive.org/details/lessonslearnedso00kane). Wiley. pp. 31 (https://archive.org/details/lessonslearnedso00kane/page/n55)–43. ISBN 9780471081128.
52. Ammann, Paul; Offutt, Jeff (January 28, 2008). *Introduction to Software Testing* (https://books.google.com/books?id=leokXF8pLY0C&pg=PA215). Cambridge University Press. p. 215. ISBN 978-0-521-88038-1. Retrieved November 29, 2017.
53. Danglot, Benjamin; Vera-Pérez, Oscar Luis; Baudry, Benoit; Monperrus, Martin (2019). "Automatic test improvement with DSpot: a study with ten mature open-source projects" (https://hal.archives-ouvertes.fr/hal-01923575/document). *Empirical Software Engineering*. **24** (4): 2603–2635. arXiv:1811.08330 (https://arxiv.org/abs/1811.08330). doi:10.1007/s10664-019-09692-y (https://doi.org/10.1007%2Fs10664-019-09692-y). S2CID 53748524 (https://api.semanticscholar.org/CorpusID:53748524).
54. Danglot, Benjamin; Vera-Perez, Oscar; Yu, Zhongxing; Zaidman, Andy; Monperrus, Martin; Baudry, Benoit (November 2019). "A snowballing literature study on test amplification" (https://hal.inria.fr/hal-02290742/file/1705.10692.pdf) (PDF). *Journal of Systems and Software*. **157**: 110398. arXiv:1705.10692 (https://arxiv.org/abs/1705.10692). doi:10.1016/j.jss.2019.110398 (https://doi.org/10.1016%2Fj.jss.2019.110398). S2CID 20959692 (https://api.semanticscholar.org/CorpusID:20959692).
55. "Standard Glossary of Terms used in Software Testing" (https://www.astqb.org/documents/Glossary-of-Software-Testing-Terms-v3.pdf) (PDF). Version 3.1. International Software Testing Qualifications Board. Retrieved January 9, 2018.
56. O'Reilly, Tim (September 30, 2005). "What is Web 2.0" (http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=4). O'Reilly Media. Section 4. End of the Software Release Cycle. Retrieved January 11, 2018.
57. Auerbach, Adam (August 3, 2015). "Part of the Pipeline: Why Continuous Testing Is Essential" (https://www.techwell.com/techwell-insights/2015/08/part-pipeline-why-continuous-testing-essential). *TechWell Insights*. TechWell Corp. Retrieved January 12, 2018.
58. Philipp-Edmonds, Cameron (December 5, 2014). "The Relationship between Risk and Continuous Testing: An Interview with Wayne Ariola" (http://www.stickyminds.com/interview/relationship-between-risk-and-continuous-testing-interview-wayne-ariola). *Stickyminds*. Retrieved January 16, 2018.
59. Ariola, Wayne; Dunlop, Cynthia (October 2015). *DevOps: Are You Pushing Bugs to Clients Faster?* (http://uploads.pnsqc.org/2015/papers/t-007_Ariola_paper.pdf) (PDF). Pacific Northwest Software Quality Conference. Retrieved January 16, 2018.

60. Auerbach, Adam (October 2, 2014). "Shift Left and Put Quality First" (https://www.techwell.com/techwell-insights/2014/10/shift-left-and-put-quality-first). *TechWell Insights*. TechWell Corp. Retrieved January 16, 2018.
61. "Section 4.38". *ISO/IEC/IEEE 29119-1:2013 – Software and Systems Engineering – Software Testing – Part 1 – Concepts and Definitions* (https://www.iso.org/standard/45142.html). International Organization for Standardization. Retrieved January 17, 2018.
62. "Globalization Step-by-Step: The World-Ready Approach to Testing. Microsoft Developer Network" (https://web.archive.org/web/20120623050851/https://msdn.microsoft.com/en-us/goglobal/bb688148). Msdn.microsoft.com. Archived from the original (https://msdn.microsoft.com/en-us/goglobal/bb688148) on June 23, 2012. Retrieved January 13, 2012.
63. " "QuickCheck: a lightweight tool for random testing of Haskell programs" " (https://dl.acm.org/doi/abs/10.1145/351240.351266). *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. 2000. doi:10.1145/351240.351266 (https://doi.org/10.1145%2F351240.351266). S2CID 5668071 (https://api.semanticscholar.org/CorpusID:5668071).
64. "Software Testing Lifecycle" (http://www.etestinghub.com/testing_lifecycles.php#2). *etestinghub*. Testing Phase in Software Testing. Retrieved January 13, 2012.
65. Dustin, Elfriede (2002). *Effective Software Testing* (https://books.google.com/books?id=K0qWBUOAf6IC&pg=PA3). Addison-Wesley Professional. p. 3. ISBN 978-0-201-79429-8.
66. Brown, Chris; Cobb, Gary; Culbertson, Robert (April 12, 2002). *Introduction to Rapid Software Testing* (http://www.informit.com/articles/article.aspx?p=26320&seqNum=6).
67. "What is Test Driven Development (TDD)?" (https://www.agilealliance.org/glossary/tdd/). *Agile Alliance*. December 5, 2015. Retrieved March 17, 2018.
68. "Test-Driven Development and Continuous Integration for Mobile Applications" (https://msdn.microsoft.com/en-us/library/bb985498.aspx#_Continuous_Integration). *msdn.microsoft.com*. Retrieved March 17, 2018.
69. Joshi, Shrinivas; Orso, Alessandro (October 2007). "SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions" (https://ieeexplore.ieee.org/document/4362636/). *2007 IEEE International Conference on Software Maintenance*: 234–243. doi:10.1109/ICSM.2007.4362636 (https://doi.org/10.1109%2FICSM.2007.4362636).
70. Steven, John; Chandra, Pravir; Fleck, Bob; Podgurski, Andy (September 2000). "jRapture: A Capture/Replay tool for observation-based testing" (https://dl.acm.org/doi/10.1145/347636.348993). *ACM SIGSOFT Software Engineering Notes*. **25** (5): 158–167. doi:10.1145/347636.348993 (https://doi.org/10.1145%2F347636.348993). ISSN 0163-5948 (https://www.worldcat.org/issn/0163-5948).
71. Saieva, Anthony; Singh, Shirish; Kaiser, Gail (September 2020). "Ad hoc Test Generation Through Binary Rewriting" (https://ieeexplore.ieee.org/document/9252025/). *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE: 115–126. doi:10.1109/SCAM51674.2020.00018 (https://doi.org/10.1109%2FSCAM51674.2020.00018). ISBN 978-1-7281-9248-2.
72. Tiwari, Deepika; Zhang, Long; Monperrus, Martin; Baudry, Benoit (2021). "Production Monitoring to Improve Test Suites" (https://ieeexplore.ieee.org/document/9526340/). *IEEE Transactions on Reliability*: 1–17. arXiv:2012.01198 (https://arxiv.org/abs/2012.01198). doi:10.1109/TR.2021.3101318 (https://doi.org/10.1109%2FTR.2021.3101318). ISSN 0018-9529 (https://www.worldcat.org/issn/0018-9529).
73. Rodríguez, Ismael; Llana, Luis; Rabanal, Pablo (2014). "A General Testability Theory: Classes, properties, complexity, and testing reductions" (https://zenodo.org/record/1008509). *IEEE Transactions on Software Engineering*. **40** (9): 862–894. doi:10.1109/TSE.2014.2331690 (https://doi.org/10.1109%2FTSE.2014.2331690). ISSN 0098-5589 (https://www.worldcat.org/issn/0098-5589). S2CID 6015996 (https://api.semanticscholar.org/CorpusID:6015996).

74. Rodríguez, Ismael (2009). "A General Testability Theory". *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1–4, 2009. Proceedings*. pp. 572–586. doi:10.1007/978-3-642-04081-8_38 (https://doi.org/10.1007%2F978-3-642-04081-8_38). ISBN 978-3-642-04080-1.
75. IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 978-0-7381-1443-9.
76. Strom, David (July 1, 2009). "We're All Part of the Story" (https://web.archive.org/web/20090831182649/http://stpcollaborative.com/knowledge/272-were-all-part-of-the-story). Software Test & Performance Collaborative. Archived from the original (http://stpcollaborative.com/knowledge/272-were-all-part-of-the-story) on August 31, 2009.
77. Griffiths, M. (2005). "Teaching agile project management to the PMI". *Agile Development Conference (ADC'05). ieee.org*. pp. 318–322. doi:10.1109/ADC.2005.45 (https://doi.org/10.1109%2FADC.2005.45). ISBN 0-7695-2487-7. S2CID 30322339 (https://api.semanticscholar.org/CorpusID:30322339).
78. Willison, John S. (April 2004). "Agile Software Development for an Agile Force" (https://web.archive.org/web/20051029135922/http://www.stsc.hill.af.mil/crosstalk/2004/04/0404willison.html). *CrossTalk*. STSC (April 2004). Archived from the original (http://www.stsc.hill.af.mil/crosstalk/2004/04/0404willison.htm) on October 29, 2005.
79. An example is Mark Fewster, Dorothy Graham: *Software Test Automation.* Addison Wesley, 1999, ISBN 0-201-33140-3.
80. "stop29119" (https://web.archive.org/web/20141002033046/http://commonsensetesting.org/stop29119). *commonsensetesting.org*. Archived from the original (http://commonsensetesting.org/stop29119) on October 2, 2014.
81. Paul Krill (August 22, 2014). "Software testers balk at ISO 29119 standards proposal" (http://www.infoworld.com/t/application-development/software-testers-balk-iso-29119-standards-proposal-249031). *InfoWorld*.
82. Kaner, Cem (2001). "NSF grant proposal to 'lay a foundation for significant improvements in the quality of academic and commercial courses in software testing' " (http://www.testingeducation.org/general/nsf_grant.pdf) (PDF).
83. Kaner, Cem (2003). *Measuring the Effectiveness of Software Testers* (http://www.testingeducation.org/a/mest.pdf) (PDF). STAR East. Retrieved January 18, 2018.
84. McConnell, Steve (2004). *Code Complete* (https://archive.org/details/codecomplete0000mcco) (2nd ed.). Microsoft Press. p. 29 (https://archive.org/details/codecomplete0000mcco/page/29). ISBN 978-0735619678.
85. Bossavit, Laurent (November 20, 2013). *The Leprechauns of Software Engineering: How folklore turns into fact and what to do about it* (https://leanpub.com/leprechauns). Chapter 10: leanpub.
86. Tran, Eushiuan (1999). "Verification/Validation/Certification" (http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html) (coursework). Carnegie Mellon University. Retrieved August 13, 2008.

Further reading

- Meyer, Bertrand (August 2008). "Seven Principles of Software Testing" (http://se.ethz.ch/~meyer/publications/testing/principles.pdf) (PDF). *Computer*. Vol. 41 no. 8. pp. 99–101. doi:10.1109/MC.2008.306 (https://doi.org/10.1109%2FMC.2008.306). Retrieved November 21, 2017.
- What is Software Testing? (https://softwaretestingboard.com/q2a/3379) - Answered by community of Software Testers at Software Testing Board

External links

- Software testing tools and products (https://curlie.org/Computers/Programming/Software_Testing/Products_and_Tools) at Curlie
- "Software that makes Software better" Economist.com (http://www.economist.com/science/tq/displaystory.cfm?story_id=10789417)