Software testing

From Wikipedia, the free encyclopedia

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.[1] 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 are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

It involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically (but not exclusively) attempts to execute a program or application with the intent of finding software bugs (errors or other defects).

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

Software testing can be conducted as soon as executable software (even if partially complete) exists. The overall approach to software development often determines when and how testing is conducted. For example, in a phased process, most testing occurs after system requirements have been defined and then implemented in testable programs. In contrast, under an Agile approach, requirements, programming, and testing are often done concurrently.

Contents

- 1 Overview
  - 1.1 Defects and failures
  - 1.2 Input combinations and preconditions
  - 1.3 Economics
  - 1.4 Roles
- 2 History

- 3 Testing methods
  - 3.1 Static vs. dynamic testing
  - 3.2 The box approach
    - 3.2.1 White-Box testing
    - 3.2.2 Black-box testing
      - 3.2.2.1 Visual testing
    - 3.2.3 Grey-box testing
- 4 Testing levels
  - 4.1 Unit testing
  - 4.2 Integration testing
  - 4.3 Component interface testing
  - 4.4 System testing
- 5 Testing Types
  - 5.1 Installation testing
  - 5.2 Compatibility testing
  - 5.3 Smoke and sanity testing
  - 5.4 Regression testing
  - 5.5 Acceptance testing
  - 5.6 Alpha testing
  - 5.7 Beta testing
  - 5.8 Functional vs non-functional testing
  - 5.9 Destructive testing
  - 5.10 Software performance testing
  - 5.11 Usability testing
  - 5.12 Accessibility testing
  - 5.13 Security testing
  - 5.14 Internationalization and localization
  - 5.15 Development testing
  - 5.16 A/B testing
  - 5.17 Concurrent testing
  - 5.18 Conformance testing or type testing
- 6 Testing process
  - 6.1 Traditional waterfall development model
  - 6.2 Agile or Extreme development model
  - 6.3 Top-down and bottom-up
  - 6.4 A sample testing cycle

- 7 Automated testing
  - 7.1 Testing tools
  - 7.2 Measurement in software testing
    - 7.2.1 Hierarchy of testing difficulty
- 8 Testing artifacts
- 9 Certifications
- 10 Controversy
- 11 Related processes
  - 11.1 Software verification and validation
  - 11.2 Software quality assurance (SQA)
- 12 See also
- 13 References
- 14 Further reading
- 15 External links

Overview

Although testing can determine the correctness of software under the assumption of some specific hypotheses (see hierarchy of testing difficulty below), testing cannot identify all the defects within software.[2] Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against 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 can only establish that it does not function properly under specific conditions.[4] The scope of software testing often includes examination of code as well as 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]

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 is the process of attempting to make this assessment.

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is requirement gaps, e.g., unrecognized requirements which result in errors of omission by the program designer.[6] Requirement gaps can often be non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure.[7] Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into 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 defect may result in a wide range of failure symptoms.

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][8] This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*)—usability, scalability, performance, compatibility, 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] Note that "coverage", as used here, is referring to combinatorial coverage, not requirements coverage.

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]

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.[11] 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 which seem to contradict Boehm's graph, and no numerical results which 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.[12]

Roles

Software testing can be done by 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,[13] different roles have been established: *manager*, *test lead*, *test analyst*, *test designer*, *tester*, *automation developer*, and *test administrator*.

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979.[14] Although his attention was on breakage testing ("a successful test is one that finds a bug"[14][15]) it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:[16]

- Until 1956 – Debugging oriented[17]
- 1957–1978 – Demonstration oriented[18]
- 1979–1982 – Destruction oriented[19]
- 1983–1987 – Evaluation oriented[20]
- 1988–2000 – Prevention oriented[21]

Testing methods

Static vs. dynamic testing

There are many approaches available in software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing is often implicit, as 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. Typical techniques for this are either using stubs/drivers or execution from a debugger environment.

Static testing involves verification, whereas dynamic testing involves validation. Together they help improve software quality. Among the techniques for static analysis, mutation testing can be used to ensure the test-cases will detect errors which are introduced by mutating the source code.

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 a test engineer takes when designing test cases.

White-Box testing

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing** and **structural testing**) tests 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, 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. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

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. 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:

- API testing (application programming interface) – testing of the application using public and private APIs
- 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.[22] Code coverage as a software

metric can be reported as a percentage for:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

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.

Black-box testing

Black box diagram

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The testers are only aware of what the software is supposed to do, not how it does it.[23] 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.

Specification-based testing aims to test the functionality of software according to the applicable requirements.[24] 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.[25]

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."[26] 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. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

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.[27][28]

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 dramatically 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 he or she requires of a test failure and can instead focus on the cause of the fault and how it should be fixed.

Visual testing is particularly well-suited for environments that deploy agile methods in their development of software, since agile methods require greater communication between testers and developers and collaboration within small teams.

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. In ad hoc testing, where testing takes place in an improvised, impromptu way, the ability of a test tool to visually record everything that occurs on a system becomes very important.

Visual testing is gathering recognition in customer acceptance and usability testing, because the test can be used by many individuals involved in the development process. For the customer, it becomes easy to provide detailed bug reports and feedback, and for program users, visual testing can record user actions on screen, as well as their voice and image, to provide a complete picture at the time of software failure for the developer.

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 is not required to have full access to the software's source code.[29] Manipulating input data and formatting output do not qualify as grey-box, because 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 test.

However, tests that require modifying a back-end data repository such as a database or a log file does qualify as grey-box, as the user would not normally be able to change the data repository in normal production operations. Grey-box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

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.[30]

Testing levels

There are generally four recognized levels of tests: unit testing, integration testing, system testing, and acceptance testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the SWEBOK guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model.[31] Other test levels are classified by the testing objective.[31]

Unit testing

Unit testing, also known as component 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.[32]

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 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. Rather than replace traditional QA focuses, it augments it. Unit testing aims to eliminate construction errors before code is promoted to QA; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development and QA 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 verification 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.[33]

Component interface testing

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.[34][35] 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.[34] Unusual data values in an interface can help explain unexpected performance in the next unit. Component interface testing is a variation of black-box testing,[35] with the focus on the data values beyond just the related actions of a subsystem component.

System testing

System testing, or end-to-end testing, tests a completely integrated system to verify that it meets its requirements.[36] For example, a system test might involve testing a logon interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

In addition, the software testing should ensure that the program, as well as working as expected, does not also destroy or partially corrupt its operating environment or cause other processes within that environment to become inoperative (this includes not corrupting shared memory, not consuming or locking up excessive resources and leaving any parallel processes unharmed by its presence).

Testing Types

Installation testing

An installation test assures that the system is installed correctly and working at actual customer's hardware.

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 was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

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. 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. Regression testing is typically the largest test effort in commercial software development,[37] 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.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. 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.[38]

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 infinite period of time (perpetual beta).

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.

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.

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 to that unauthorised persons or systems cannot use, read or modify them, and authorized persons or systems are not denied access to them."[39]

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).[40]

Actual translation to human languages must be tested, too. Possible localization 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 which 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 which 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 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. Rather than replace traditional QA focuses, it augments it. Development Testing aims to eliminate construction errors before code is promoted to QA; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development and QA 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 verification practices.

A/B testing

A/B testing is basically a comparison of two outputs, generally when only one variable has changed: run a test, change one thing, run the test again, compare the results. This is more useful with more small-scale situations, but very useful in fine-tuning any program. With more complex projects, multivariant testing can be done.

Concurrent testing

In concurrent testing, the focus is more on what the performance is like when continuously running with normal input and under normal operation as opposed to stress testing, or fuzz testing. Memory leak is more easily found and resolved using this method, as well as more basic faults.

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.

Testing process

Traditional waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer.[41] 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.[42]

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.[43]

Agile or Extreme 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). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. 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 goal of this test process is to achieve continuous integration where software updates can be published to the public frequently. [44] [45]

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.

Top-down and bottom-up

Bottom Up Testing is an approach to integrated testing where the lowest level components (modules, procedures, and functions) are tested first, then integrated and used to facilitate the testing of higher level components. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. The process is repeated until the components at the top of the hierarchy are tested. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

In both, method stubs and drivers are used to stand-in for missing components and are replaced as the levels are completed.

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing.[46] 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.
- **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. AKA Resolution testing.
- **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
  - 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 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 an Integrated Development Environment (IDE).

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, 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 amount 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.[47] [48] 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 into 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

The software testing process can produce several artifacts.

Test plan

A test plan is a document detailing the objectives, target market, internal beta team, and processes for a specific beta test. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

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 actual result. Clinically defined a test case is an input and an expected result.[49] This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described 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 repository. 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 programing code that replicates user actions. Initially the term was derived from the product of work created by automated regression test tools. 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.

Test harness

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

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. 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. This has led some to declare that the testing field is not ready for certification.[50] Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.[51]

Software testing certification types

Exam-based: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI][52]

Education-based: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)]

Testing certifications

ISEB offered by the Information Systems Examinations Board

ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board[53][54]

ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board[53][54]

Quality assurance certifications

CSQE offered by the American Society for Quality (ASQ)[55]

CQIA offered by the American Society for Quality (ASQ)[55]

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?

Members of the "context-driven" school of testing[56] believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.[57]

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,[58][59] whereas government and military[60] software providers use this methodology but also the traditional test-last models (e.g. in the Waterfall model).

Exploratory test vs. scripted

[61] Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly.[62] More in particular, test-driven development states that developers should write unit-tests, as those of XUnit, before coding the functionality. The tests 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.

Software design vs. software implementation

Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?

The idea is that any form of observation is also an interaction — the act of testing can also affect that which is being tested.[63]

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, are driving the efforts to have the standard withdrawn.[64][65]

Related processes

Software verification and validation

Software testing is used in association with verification and validation:[66]

- 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 incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

> 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.

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.

Software quality assurance (SQA)

Software testing is a part of the software quality assurance (SQA) process.[4] In SQA, software process specialists and auditors are concerned for 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 a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

See also

- Category:Software testing

- Dynamic program analysis
- Formal verification
- Independent test organization
- Manual testing
- Orthogonal array testing
- Pair testing
- Reverse semantic traceability
- Software testability
- Orthogonal Defect Classification
- Test Environment Management
- Test management tools
- Web testing

References

1. Kaner, Cem (November 17, 2006). "Exploratory Testing" (http://www.kaner.com/pdfs/ETatQAI.pdf). Florida Institute of Technology, *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL. Retrieved November 22, 2014.
2. Software Testing (http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/) by Jiantao Pan, Carnegie Mellon University
3. Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development – Writing Test Cases" (http://se.inf.ethz.ch/people/leitner/publications/cdd_leitner_esec_fse_2007.pdf), Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
4. Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc (1999). *Testing Computer Software, 2nd Ed*. New York, et al: John Wiley and Sons, Inc. p. 480. ISBN 0-471-35846-0.
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. pp. 41–43. ISBN 0-470-04212-5.
6. 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. p. 426. ISBN 0-470-04212-5.
7. Section 1.1.2, Certified Tester Foundation Level Syllabus (http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf), International Software Testing Qualifications Board
8. Principle 2, Section 1.3, Certified Tester Foundation Level Syllabus (http://www.bcs.org/upload/pdf/istqbsyll.pdf), International Software Testing Qualifications Board
9. "Proceedings from the 5th International Conference on Software Testing and Validation (ICST). Software

Competence Center Hagenberg. "Test Design: Lessons Learned and Practical Implications." (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4578383).
10. Software errors cost U.S. economy $59.5 billion annually (http://www.nist.gov/director/planning/upload/report02-3.pdf), NIST report
11. McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. p. 29. ISBN 0735619670.
12. Bossavit, Laurent (2013-11-20). *The Leprechauns of Software Engineering--How folklore turns into fact and what to do about it* (https://leanpub.com/leprechauns). Chapter 10: leanpub.
13. see D. Gelperin and W.C. Hetzel
14. Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. ISBN 0-471-04328-1.
15. Company, People's Computer (1987). "Dr. Dobb's journal of software tools for the professional programmer" (http://books.google.com/?id=7RoIAAAAIAAJ). *Dr. Dobb's journal of software tools for the professional programmer* (M&T Pub) **12** (1–6): 116.
16. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6): 687–695. doi:10.1145/62959.62965 (https://dx.doi.org/10.1145%2F62959.62965). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
17. *until 1956 it was the debugging oriented period, when testing was often associated to debugging: there was no clear difference between testing and debugging.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
18. *From 1957–1978 there was the demonstration oriented period where debugging and testing was distinguished now – in this period it was shown, that software satisfies the requirements.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
19. *The time between 1979–1982 is announced as the destruction oriented period, where the goal was to find errors.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
20. *1983–1987 is classified as the evaluation oriented period: intention here is that during the software lifecycle a product evaluation is provided and measuring quality.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
21. *From 1988 on it was seen as prevention oriented period where tests were to demonstrate that software satisfies its specification, to detect faults and to prevent faults.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).
22. Introduction (http://www.bullseye.com/coverage.html#intro), Code Coverage Analysis, Steve Cornett
23. Ron, Patton. *Software Testing*.
24. Laycock, G. T. (1993). "The Theory and Practice of Specification Based Software Testing" (http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz) (PostScript). Dept of Computer Science, Sheffield University, UK. Retrieved 2008-02-13.
25. 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 2008-08-19.
26. Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
27. "Visual testing of software – Helsinki University of Technology" (http://www.cs.hut.fi/~jlonnber/VisualTesting.pdf) (PDF). Retrieved 2012-01-13.

28. "Article on visual testing in Test Magazine" (http://www.testmagazine.co.uk/2011/04/visual-testing). Testmagazine.co.uk. Retrieved 2012-01-13.
29. Patton, Ron. *Software Testing*.
30. "SOA Testing Tools for Black, White and Gray Box SOA Testing Techniques" (http://www.crosschecknet.com/soa_testing_black_white_gray_box.php). Crosschecknet.com. Retrieved 2012-12-10.
31. "SWEBOK Guide – Chapter 5" (http://www.computer.org/portal/web/swebok/html/ch5#Ref2.1). Computer.org. Retrieved 2012-01-13.
32. Binder, Robert V. (1999). *Testing Object-Oriented Systems: Objects, Patterns, and Tools*. Addison-Wesley Professional. p. 45. ISBN 0-201-80938-9.
33. Beizer, Boris (1990). *Software Testing Techniques* (Second ed.). New York: Van Nostrand Reinhold. pp. 21,430. ISBN 0-442-20672-0.
34. Clapp, Judith A. (1995). *Software Quality Control, Error Analysis, and Testing* (http://books.google.com/books?id=wAq0rnyiGMEC&pg=PA313). p. 313. ISBN 0815513631.
35. Mathur, Aditya P. (2008). *Foundations of Software Testing* (http://books.google.com/books?id=yU-rTcurys8C&pg=PA18). Purdue University. p. 18. ISBN 978-8131716601.
36. IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York: IEEE. ISBN 1-55937-079-3.
37. Paul Ammann; Jeff Offutt (2008). *Introduction to Software Testing* (http://books.google.com/books?id=leokXF8pLY0C&pg=PA215). p. 215 of 322 pages.
38. van Veenendaal, Erik. "Standard glossary of terms used in Software Testing" (http://www.astqb.org/get-certified/istqb-syllabi-the-istqb-software-tester-certification-body-of-knowledge/). Retrieved 4 January 2013.
39. ISO/IEC/IEEE 29119-1:2013 – Software and Systems Engineering – Software Testing – Part 1 – Concepts and Definitions; Section 4.38
40. "Globalization Step-by-Step: The World-Ready Approach to Testing. Microsoft Developer Network" (http://msdn.microsoft.com/en-us/goglobal/bb688148). Msdn.microsoft.com. Retrieved 2012-01-13.
41. EtestingHub-Online Free Software Testing Tutorial. "e)Testing Phase in Software Testing:" (http://www.etestinghub.com/testing_lifecycles.php#2). Etestinghub.com. Retrieved 2012-01-13.
42. Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. pp. 145–146. ISBN 0-471-04328-1.
43. Dustin, Elfriede (2002). *Effective Software Testing*. Addison Wesley. p. 3. ISBN 0-201-79429-2.
44. Marchenko, Artem (November 16, 2007). "XP Practice: Continuous Integration" (http://agilesoftwaredevelopment.com/xp/practices/continuous-integration). Retrieved 2009-11-16.
45. Gurses, Levent (February 19, 2007). "Agile 101: What is Continuous Integration?" (http://www.jacoozi.com/blog/?p=18). Retrieved 2009-11-16.
46. Pan, Jiantao (Spring 1999). "Software Testing (18-849b Dependable Embedded Systems)". *Topics in Dependable Embedded Systems* (http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/). Electrical and Computer Engineering Department, Carnegie Mellon University.
47. Rodríguez, Ismael; Llana, Luis; Rabanal, Pablo (2014). "A General Testability Theory: Classes, properties, complexity, and testing reductions". *IEEE Transactions on Software Engineering* **40** (9): 862–894. doi:10.1109/TSE.2014.2331690 (https://dx.doi.org/10.1109%2FTSE.2014.2331690). ISSN 0098-5589 (https://www.worldcat.org/issn/0098-5589).

48. 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://dx.doi.org/10.1007%2F978-3-642-04081-8_38). ISBN 978-3-642-04080-1.
49. IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 0-7381-1443-X.
50. 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).
51. Kaner, Cem (2003). "Measuring the Effectiveness of Software Testers" (http://www.testingeducation.org/a/mest.pdf) (PDF).
52. Black, Rex (December 2008). *Advanced Software Testing- Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*. Santa Barbara: Rocky Nook Publisher. ISBN 1-933952-36-9.
53. "ISTQB" (http://www.istqb.org/).
54. "ISTQB in the U.S." (http://www.astqb.org/).
55. "American Society for Quality" (http://www.asq.org/). Asq.org. Retrieved 2012-01-13.
56. "context-driven-testing.com" (http://www.context-driven-testing.com). context-driven-testing.com. Retrieved 2012-01-13.
57. "Article on taking agile traits without the agile method" (http://www.technicat.com/writing/process.html). Technicat.com. Retrieved 2012-01-13.
58. "We're all part of the story" (http://stpcollaborative.com/knowledge/272-were-all-part-of-the-story) by David Strom, July 1, 2009
59. IEEE article about differences in adoption of agile trends between experienced managers vs. young students of the Project Management Institute (http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/10705/33795/01609838.pdf?temp=x). See also Agile adoption study from 2007 (http://www.ambysoft.com/downloads/surveys/AgileAdoption2007.ppt)
60. Willison, John S. (April 2004). "Agile Software Development for an Agile Force" (http://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.
61. "IEEE article on Exploratory vs. Non Exploratory testing" (http://ieeexplore.ieee.org/iel5/10351/32923/01541817.pdf?arnumber=1541817). Ieeexplore.ieee.org. Retrieved 2012-01-13.
62. An example is Mark Fewster, Dorothy Graham: *Software Test Automation.* Addison Wesley, 1999, ISBN 0-201-33140-3.
63. Microsoft Development Network Discussion on exactly this topic (http://channel9.msdn.com/forums/Coffeehouse/402611-Are-you-a-Test-Driven-Developer/)
64. Stop 29119 (http://commonsensetesting.org/stop29119)
65. Infoworld.com (http://www.infoworld.com/t/application-development/software-testers-balk-iso-29119-standards-proposal-249031)
66. Tran, Eushiuan (1999). "Verification/Validation/Certification" (http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html). In Koopman, P. *Topics in Dependable Embedded Systems*. USA: Carnegie Mellon University. Retrieved 2008-01-13.

Embedded Systems. USA: Carnegie Mellon University. Retrieved 2008-01-13.

Further reading

- Bertrand Meyer, "Seven Principles of Software Testing," Computer, vol. 41, no. 8, pp. 99–101, Aug. 2008, doi:10.1109/MC.2008.306; available online (http://se.ethz.ch/~meyer/publications/testing/principles.pdf).

External links

- Software testing tools and products (https://www.dmoz.org/Computers/Programming/Software_Testing/Products_and_Tools) at DMOZ
- "Software that makes Software better" Economist.com (http://www.economist.com/science/tq/displaystory.cfm?story_id=10789417)
- "What You Need to Know About Software Beta Tests" Centercode.com (http://www.centercode.com/blog/2015/03/what-you-need-to-know-about-software-beta-tests/)