# Software testing tactics

This article discusses a set of tactics useful in software testing. It is intended as a comprehensive list of tactical approaches to Software Quality Assurance (more widely colloquially known as Quality Assurance (traditionally called by the acronym "QA") and general application of the test method (usually just called "testing" or sometimes "developer testing").

## Contents

## Installation testing

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

## 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, by seeing the source code) 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 – 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.[1] 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
- *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.

## Black-box testing

Input
Blackbox
Output
Black box diagram

Black-box 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.[2] 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.[3] 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.[4]

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."[5] 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.[6][7]

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 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 in order to document the steps taken to uncover the bug.

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 developers.

### 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.[2] 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.[8]

## 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.

## Automated 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 GUI(Graphical User Interface) 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).

## Abstraction of application layers as applied to automated testing

There are generally four recognized levels of tests: unit testing, integration testing, component interface testing, and system 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.[9] Other test levels are classified by the testing objective.[9]

There are two different levels of tests from the perspective of customers: low-level testing (LLT) and high-level testing (HLT). LLT is a group of tests for different level components of software application or product. HLT is a group of tests for the whole software application or product.

### 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.[10]

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.[11]

### 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.[12][13] 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.[12] 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,[13] with the focus on the data values beyond just the related actions of a subsystem component.

### System testing

System testing tests a completely integrated system to verify that the system meets its requirements.[14] 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.

### Operational acceptance testing

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, and/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 which 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.[15]

## 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,[16] 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.[17]

## 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).

## 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.[18][19] 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.[20][21][22]

## 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 so that unauthorised persons or systems cannot use, read or modify them, and authorized persons or systems are not denied access to them."[23]

## Internationalization and localization testing

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).[24]

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 on the performance while continuously running with normal input and under normal operational conditions, as opposed to stress testing, or fuzz testing. Memory leaks, as well as basic faults are easier to find with this method.

## 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.

## References

1. Introduction (http://www.bullseye.com/coverage.html#intro), Code Coverage Analysis, Steve Cornett
2. Patton, Ron (2006). *Software Testing* (https://archive.org/details/softwaretesting0000patt) (2nd ed.). Sams Publishing (published July 26, 2005). ISBN 978-0672327988.
3. Laycock, G. T. (1993). "The Theory and Practice of Specification Based Software Testing" (https://web.archive.org/web/20070214130159/http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz). Dept of Computer Science, Sheffield University, UK. Archived from the original (http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz) (PostScript) on 2007-02-14. Retrieved 2008-02-13.
4. 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.
5. Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
6. "Visual testing of software – Helsinki University of Technology" (http://www.cs.hut.fi/~jlonnber/VisualTesting.pdf) (PDF). Retrieved 2012-01-13.
7. "Article on visual testing in Test Magazine" (https://web.archive.org/web/20120724162657/http://www.testmagazine.co.uk/2011/04/visual-testing/). Testmagazine.co.uk. Archived from the original (http://www.testmagazine.co.uk/2011/04/visual-testing) on 2012-07-24. Retrieved 2012-01-13.
8. "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.
9. "SWEBOK Guide – Chapter 5" (http://www.computer.org/portal/web/swebok/html/ch5#Ref2.1). Computer.org. Retrieved 2012-01-13.
10. 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 0-201-80938-9.
11. Beizer, Boris (1990). *Software Testing Techniques* (Second ed.). New York: Van Nostrand Reinhold. pp. 21, 430. ISBN 0-442-20672-0.
12. Clapp, Judith A. (1995). *Software Quality Control, Error Analysis, and Testing* (https://books.google.com/books?id=wAq0rnyiGMEC&pg=PA313). p. 313. ISBN 0815513631.
13. Mathur, Aditya P. (2008). *Foundations of Software Testing* (https://books.google.com/books?id=yU-rTcurys8C&pg=PA18). Purdue University. p. 18. ISBN 978-8131716601.
14. IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York: IEEE. ISBN 1-55937-079-3.
15. Whitepaper: Operational Acceptance – an application of the ISO 29119 Software Testing standard. May 2015 Anthony Woods, Capgemini

16. Paul Ammann; Jeff Offutt (2008). *Introduction to Software Testing* (https://books.google.com/books?id=leokXF8pLY0C&pg=PA215). p. 215 of 322 pages.
17. 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.
18. Part of the Pipeline: Why Continuous Testing Is Essential (https://www.techwell.com/techwell-insights/2015/08/part-pipeline-why-continuous-testing-essential), by Adam Auerbach, TechWell Insights August 2015
19. 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), by Cameron Philipp-Edmonds, Stickyminds December 2015
20. DevOps: Are You Pushing Bugs to Clients Faster (http://uploads.pnsqc.org/2015/papers/t-007_Ariola_paper.pdf), by Wayne Ariola and Cynthia Dunlop, PNSQC October 2015
21. DevOps and QA: What's the real cost of quality? (http://devops.com/2015/06/11/devops-and-qa-whats-the-real-cost-of-quality/), by Ericka Chickowski, DevOps.com June 2015
22. Shift Left and Put Quality First (https://www.techwell.com/techwell-insights/2014/10/shift-left-and-put-quality-first), by Adam Auerbach, TechWell Insights October 2014
23. ISO/IEC/IEEE 29119-1:2013 – Software and Systems Engineering – Software Testing – Part 1 – Concepts and Definitions; Section 4.38
24. "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.

## External links

- Software testing tools and products (https://curlie.org/Computers/Programming/Software_Testing/Products_and_Tools) at Curlie