**Overview**

Testing can never completely identify all the defects within software. Instead, it furnishes a criticism or comparison that compares the state and behavior of the product against [oracles](http://en.wikipedia.org/wiki/Oracle_(software_testing))—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, [contracts](http://en.wikipedia.org/wiki/Design_by_Contract), 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]](http://en.wikipedia.org/wiki/Software_testing#cite_note-Kaner1-4) 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.

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]](http://en.wikipedia.org/wiki/Software_testing#cite_note-6) Requirement gaps can often be [non-functional requirements](http://en.wikipedia.org/wiki/Non-functional_requirements) such as [testability](http://en.wikipedia.org/wiki/Software_testability), [scalability](http://en.wikipedia.org/wiki/Scalability), [maintainability](http://en.wikipedia.org/wiki/Maintainability), [usability](http://en.wikipedia.org/wiki/Usability), [performance](http://en.wikipedia.org/wiki/Computer_performance), and [security](http://en.wikipedia.org/wiki/Computer_security).

Software faults occur through the following processes. A programmer makes an [error](http://en.wikipedia.org/wiki/Human_error) (mistake), which results in a [defect](http://en.wikipedia.org/wiki/Fault_(technology)) (fault, bug) in the software [source code](http://en.wikipedia.org/wiki/Source_code). If this defect is executed, in certain situations the system will produce wrong results, causing a [failure](http://en.wikipedia.org/wiki/Failure).[[7]](http://en.wikipedia.org/wiki/Software_testing#cite_note-ctfl-7) Not all defects will necessarily result in failures. For example, defects in [dead code](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Computer_hardware) platform, alterations in [source data](http://en.wikipedia.org/wiki/Source_data), or interacting with different software. A single defect may result in a wide range of failure symptoms.

**Input combinations and preconditions**

A very 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]](http://en.wikipedia.org/wiki/Software_testing#cite_note-Kaner1-4)[[8]](http://en.wikipedia.org/wiki/Software_testing#cite_note-8) This means that the number of [defects](http://en.wikipedia.org/wiki/Software_bug) in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, [non-functional](http://en.wikipedia.org/wiki/Non-functional_requirements) dimensions of quality (how it is supposed to be versus what it is supposed to do)—[usability](http://en.wikipedia.org/wiki/Usability), [scalability](http://en.wikipedia.org/wiki/Scalability), [performance](http://en.wikipedia.org/wiki/Computer_performance), [compatibility](http://en.wikipedia.org/wiki/Backward_compatibility), [reliability](http://en.wikipedia.org/wiki/Reliability_(engineering))—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]](http://en.wikipedia.org/wiki/Software_testing#cite_note-9)

**Economics**

A study conducted by [NIST](http://en.wikipedia.org/wiki/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.

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]](http://en.wikipedia.org/wiki/Software_testing#cite_note-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](http://en.wikipedia.org/wiki/Continuous_deployment) practices and cloud-based services, the cost of re-deployment and maintenance may lessen over time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | |  | | | | |
|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**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, different roles have been established: manager, test lead, test analyst, test designer, tester, automation developer, and test administrator.

History[[edit source](http://en.wikipedia.org/w/index.php?title=Software_testing&action=edit&section=6" \o "Edit section: History) | [edit](http://en.wikipedia.org/w/index.php?title=Software_testing&veaction=edit&section=6)]

The separation of [debugging](http://en.wikipedia.org/wiki/Debugging) from testing was initially introduced by Glenford J. Myers in 1979.[[13]](http://en.wikipedia.org/wiki/Software_testing#cite_note-Myers_1979-13) Although his attention was on breakage testing ("a successful test is one that finds a bug"[[13]](http://en.wikipedia.org/wiki/Software_testing#cite_note-Myers_1979-13)[[14]](http://en.wikipedia.org/wiki/Software_testing#cite_note-14)) it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. [Dave Gelperin](http://en.wikipedia.org/wiki/Dave_Gelperin) and [William C. Hetzel](http://en.wikipedia.org/wiki/William_C._Hetzel) classified in 1988 the phases and goals in software testing in the following stages:[[15]](http://en.wikipedia.org/wiki/Software_testing" \l "cite_note-15)

Until 1956 – Debugging oriented

1957–1978 – Demonstration oriented

1979–1982 – Destruction oriented

1983–1987 – Evaluation oriented

1988–2000 – Prevention oriented

**Testing methods**

Static vs. dynamic

There are many approaches to software testing. [Reviews](http://en.wikipedia.org/wiki/Code_review), [walkthroughs](http://en.wikipedia.org/wiki/Software_walkthrough), or [inspections](http://en.wikipedia.org/wiki/Software_inspection) are referred to as [static testing](http://en.wikipedia.org/wiki/Static_testing), whereas actually executing programmed code with a given set of [test cases](http://en.wikipedia.org/wiki/Test_case) is referred to as [dynamic testing](http://en.wikipedia.org/wiki/Dynamic_testing). Static testing can be omitted, and in practice often is. Dynamic testing takes place when the program itself is used. Dynamic testing may begin before the program is 100% complete in order to test particular sections of code and are applied to discrete [functions](http://en.wikipedia.org/wiki/Function_(computer_science)) or modules. Typical techniques for this are either using [stubs](http://en.wikipedia.org/wiki/Method_stub)/drivers or execution from a [debugger](http://en.wikipedia.org/wiki/Debugger) environment.

Static testing involves verification whereas dynamic testing involves validation. Together they help improve software quality.

**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**

Main article: [White-box testing](http://en.wikipedia.org/wiki/White-box_testing)

**White-box testing** (also known as clear box testing, glass box testing, and 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](http://en.wikipedia.org/wiki/In-circuit_test) (ICT).

While white-box testing can be applied at the [unit](http://en.wikipedia.org/wiki/Unit_testing), [integration](http://en.wikipedia.org/wiki/Integration_testing) and [system](http://en.wikipedia.org/wiki/System_testing) 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](http://en.wikipedia.org/wiki/Application_programming_interface) testing (application programming interface) – testing of the application using public and private APIs

[Code coverage](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Fault_injection) methods – intentionally introducing faults to gauge the efficacy of testing strategies

[Mutation testing](http://en.wikipedia.org/wiki/Mutation_testing) methods

[Static testing](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Function_points) have been tested.[[21]](http://en.wikipedia.org/wiki/Software_testing#cite_note-21) Code coverage as a [software metric](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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**

Main article: [Black-box testing](http://en.wikipedia.org/wiki/Black-box_testing)

[http://upload.wikimedia.org/wikipedia/commons/thumb/f/f6/Blackbox.svg/200px-Blackbox.svg.png](http://en.wikipedia.org/wiki/File:Blackbox.svg)

[http://bits.wikimedia.org/static-1.22wmf12/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Blackbox.svg)

Black box diagram

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The tester is only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: [equivalence partitioning](http://en.wikipedia.org/wiki/Equivalence_partitioning), [boundary value analysis](http://en.wikipedia.org/wiki/Boundary_value_analysis), [all-pairs testing](http://en.wikipedia.org/wiki/All-pairs_testing), [state transition tables](http://en.wikipedia.org/wiki/State_transition_table), [decision table](http://en.wikipedia.org/wiki/Decision_table) testing, [fuzz testing](http://en.wikipedia.org/wiki/Fuzz_testing), [model-based testing](http://en.wikipedia.org/wiki/Model-based_testing), [use case](http://en.wikipedia.org/wiki/Use_case) testing, [exploratory testing](http://en.wikipedia.org/wiki/Exploratory_testing) and specification-based testing.

Specification-based testing aims to test the functionality of software according to the applicable requirements. This level of testing usually requires thorough [test cases](http://en.wikipedia.org/wiki/Test_case) to be provided to the tester, who then can simply verify that for a given input, the output value (or behaviour), 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](http://en.wikipedia.org/wiki/Functional_testing) or [non-functional](http://en.wikipedia.org/wiki/Non-functional_testing), 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.

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." 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](http://en.wikipedia.org/wiki/Unit_test), [integration](http://en.wikipedia.org/wiki/Integration_testing), [system](http://en.wikipedia.org/wiki/System_testing) and [acceptance](http://en.wikipedia.org/wiki/Acceptance_test). 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 he or she requires, and the information is expressed clearly.

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](http://en.wikipedia.org/wiki/Agile_testing#Agile_methods) in their development of software, since agile methods require greater communication between testers and developers and collaboration within small teams.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)]

[Ad hoc testing](http://en.wikipedia.org/wiki/Ad_hoc_testing) and [exploratory testing](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Acceptance_testing#customer_acceptance) and [usability testing](http://en.wikipedia.org/wiki/Usability_testing), because the test can be used by many individuals involved in the development process.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] 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.

Further information: [Graphical user interface testing](http://en.wikipedia.org/wiki/Graphical_user_interface_testing)

**Grey-box testing**

Main article: [Gray box testing](http://en.wikipedia.org/wiki/Gray_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. 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](http://en.wikipedia.org/wiki/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.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] Grey-box testing may also include [reverse engineering](http://en.wikipedia.org/wiki/Reverse_coding) 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](http://en.wikipedia.org/wiki/Database). The tester can observe the state of the product being tested after performing certain actions such as executing [SQL](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Exception_handling), and so on.

**Testing levels**

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](http://en.wikipedia.org/wiki/SWEBOK) guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective.

**Unit testing**

Main article: [Unit testing](http://en.wikipedia.org/wiki/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.

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](http://en.wikipedia.org/wiki/Corner_case) or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of 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](http://en.wikipedia.org/wiki/Static_code_analysis), data flow analysis metrics analysis, peer code reviews, code coverage analysis and other software verification practices.

**Integration testing**

Main article: [Integration testing](http://en.wikipedia.org/wiki/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.

**System testing**

Main article: [System testing](http://en.wikipedia.org/wiki/System_testing)

System testing tests a completely integrated system to verify that it meets its requirements.[[33]](http://en.wikipedia.org/wiki/Software_testing#cite_note-ieee-33)

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).[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

**Acceptance testing**

Main article: [Acceptance testing](http://en.wikipedia.org/wiki/Acceptance_testing)

At last the system is delivered to the user for Acceptance testing.

**Testing Types**

Installation testing

Main article: [Installation testing](http://en.wikipedia.org/wiki/Installation_testing)

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

Compatibility testing

Main article: [Compatibility testing](http://en.wikipedia.org/wiki/Compatibility_testing)

A common cause of software failure (real or perceived) is a lack of its [compatibility](http://en.wikipedia.org/wiki/Computer_compatibility) with other [application software](http://en.wikipedia.org/wiki/Application_software), [operating systems](http://en.wikipedia.org/wiki/Operating_system) (or operating system [versions](http://en.wikipedia.org/wiki/Software_versioning), old or new), or target environments that differ greatly from the original (such as a [terminal](http://en.wikipedia.org/wiki/Computer_terminal) or [GUI](http://en.wikipedia.org/wiki/GUI) application intended to be run on the [desktop](http://en.wikipedia.org/wiki/Desktop_metaphor) now being required to become a [web application](http://en.wikipedia.org/wiki/Web_application), which must render in a [web browser](http://en.wikipedia.org/wiki/Web_browser)). For example, in the case of a lack of [backward compatibility](http://en.wikipedia.org/wiki/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 result 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](http://en.wikipedia.org/wiki/Abstraction_(computer_science)) operating system functionality into a separate program [module](http://en.wikipedia.org/wiki/Modular_programming) or [library](http://en.wikipedia.org/wiki/Library_(computing)).

**Smoke and sanity testing**

[Sanity testing](http://en.wikipedia.org/wiki/Sanity_testing) determines whether it is reasonable to proceed with further testing.

[Smoke testing](http://en.wikipedia.org/wiki/Smoke_testing) is used to determine whether there are serious problems with a piece of software, for example as a [build verification test](http://en.wikipedia.org/wiki/Build_verification_test).

**Regression testing**

Main article: [Regression testing](http://en.wikipedia.org/wiki/Regression_testing)

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover [software regressions](http://en.wikipedia.org/wiki/Software_regression), or 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](http://en.wikipedia.org/wiki/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 previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the [risk](http://en.wikipedia.org/wiki/Risk_management) 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.

**Acceptance testing**

Main article: [Acceptance testing](http://en.wikipedia.org/wiki/Acceptance_testing)

Acceptance testing can mean one of two things:

A [smoke test](http://en.wikipedia.org/wiki/Smoke_testing#Software_development) is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before [integration](http://en.wikipedia.org/wiki/Integration_testing) or [regression](http://en.wikipedia.org/wiki/Regression_testing).

Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as [user acceptance testing](http://en.wikipedia.org/wiki/User_acceptance_testing) (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)]

**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.

**Beta testing**

Beta testing comes after alpha testing and can be considered a form of external [user acceptance testing](http://en.wikipedia.org/wiki/User_acceptance_testing). Versions of the software, known as [beta versions](http://en.wikipedia.org/wiki/Beta_version), are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or [bugs](http://en.wikipedia.org/wiki/Computer_bug). Sometimes, beta versions are made available to the open public to increase the [feedback](http://en.wikipedia.org/wiki/Feedback#In_organizations) field to a maximal number of future users.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)]

**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](http://en.wikipedia.org/wiki/Scalability) or other [performance](http://en.wikipedia.org/wiki/Performance), behavior under certain [constraints](http://en.wikipedia.org/wiki/Constraints), or [security](http://en.wikipedia.org/wiki/Computer_security). Testing will determine the [flake point](http://en.wikipedia.org/w/index.php?title=Flake_point&action=edit&redlink=1), 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**

Main article: [Destructive testing](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Robustness_(computer_science)) of input validation and error-management routines.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] [Software fault injection](http://en.wikipedia.org/wiki/Fault_injection), in the form of [fuzzing](http://en.wikipedia.org/wiki/Fuzz_testing), is an example of failure testing. Various commercial non-functional testing tools are linked from the [software fault injection](http://en.wikipedia.org/wiki/Fault_injection) page; there are also numerous open-source and free software tools available that perform destructive testing.

Further information: [Exception handling](http://en.wikipedia.org/wiki/Exception_handling) and [Recovery testing](http://en.wikipedia.org/wiki/Recovery_testing)

**Software performance testing**

[Performance testing](http://en.wikipedia.org/wiki/Software_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](http://en.wikipedia.org/wiki/Load_testing#Software_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](http://en.wikipedia.org/wiki/Load_testing). This is generally referred to as software [scalability](http://en.wikipedia.org/wiki/Scalability). The related load testing activity of when performed as a non-functional activity is often referred to as endurance testing. [Volume testing](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Scalability_testing), and volume testing, are often used interchangeably.

[Real-time software](http://en.wikipedia.org/wiki/Real-time_computing) systems have strict timing constraints. To test if timing constraints are met, [real-time testing](http://en.wikipedia.org/wiki/Real-time_testing) is used.

**Usability testing**

[Usability testing](http://en.wikipedia.org/wiki/Usability_testing) is needed to check if the user interface is easy to use and understand. It is concerned mainly with the use of the application.

**Accessibility**

[Accessibility](http://en.wikipedia.org/wiki/Accessibility) testing may include compliance with standards such as:

[Americans with Disabilities Act of 1990](http://en.wikipedia.org/wiki/Americans_with_Disabilities_Act_of_1990)

[Section 508 Amendment to the Rehabilitation Act of 1973](http://en.wikipedia.org/wiki/Section_508_Amendment_to_the_Rehabilitation_Act_of_1973)

[Web Accessibility Initiative](http://en.wikipedia.org/wiki/Web_Accessibility_Initiative) (WAI) of the [World Wide Web Consortium](http://en.wikipedia.org/wiki/World_Wide_Web_Consortium) (W3C)

**Security testing**

[Security testing](http://en.wikipedia.org/wiki/Security_testing) is essential for software that processes confidential data to prevent [system intrusion](http://en.wikipedia.org/wiki/Backdoor_(computing)) by [hackers](http://en.wikipedia.org/wiki/Hacker_(computer_security)).

Internationalization and localization [[edit source](http://en.wikipedia.org/w/index.php?title=Software_testing&action=edit&section=33) | [edit](http://en.wikipedia.org/w/index.php?title=Software_testing&veaction=edit&section=33)]

The general ability of software to be [internationalized and localized](http://en.wikipedia.org/wiki/Internationalization_and_localization) can be automatically tested without actual translation, by using [pseudolocalization](http://en.wikipedia.org/wiki/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).[[35]](http://en.wikipedia.org/wiki/Software_testing#cite_note-35)

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

Software is often localized by translating a list of [strings](http://en.wikipedia.org/wiki/String_(computer_science)) 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](http://en.wikipedia.org/wiki/Hard_coding) in the source code.

Some messages may be created automatically at [run time](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.

Software may use a [keyboard shortcut](http://en.wikipedia.org/wiki/Keyboard_shortcut) which has no function on the source language's [keyboard layout](http://en.wikipedia.org/wiki/Keyboard_layout), but is used for typing characters in the layout of the target language.

Software may lack support for the [character encoding](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Bi-directional_text).

Software may display images with text that was not localized.

Localized operating systems may have differently named system [configuration files](http://en.wikipedia.org/wiki/Configuration_file) and [environment variables](http://en.wikipedia.org/wiki/Environment_variable) and different [formats for date](http://en.wikipedia.org/wiki/Date_and_time_notation_by_country) and [currency](http://en.wikipedia.org/wiki/Currency).

**Development testing**

Main article: [Development Testing](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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**

Main article: [A/B testing](http://en.wikipedia.org/wiki/A/B_testing)

**Testing process**

*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. This practice often results in the testing phase being used as a [project](http://en.wikipedia.org/wiki/Project_management) buffer to compensate for project delays, thereby compromising the time devoted to testing.

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

***Agile or Extreme development model***

In contrast, some emerging software disciplines such as [extreme programming](http://en.wikipedia.org/wiki/Extreme_programming) and the [agile software development](http://en.wikipedia.org/wiki/Agile_software_development) movement, adhere to a "[test-driven software development](http://en.wikipedia.org/wiki/Test-driven_development)" model. In this process, [unit tests](http://en.wikipedia.org/wiki/Unit_tests) are written first, by the [software engineers](http://en.wikipedia.org/wiki/Software_engineering) (often with [pair programming](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Continuous_integration) where software updates can be published to the public frequently.

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.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] 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.[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

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](http://en.wikipedia.org/wiki/Method_stub) 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.[[41]](http://en.wikipedia.org/wiki/Software_testing#cite_note-41) The sample below is common among organizations employing the [Waterfall development](http://en.wikipedia.org/wiki/Waterfall_development) model. The same practices are commonly found in other development models, but might not be as clear or explicit.

[Requirements analysis](http://en.wikipedia.org/wiki/Requirements_analysis): Testing should begin in the requirements phase of the [software development life cycle](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Test_strategy), [test plan](http://en.wikipedia.org/wiki/Test_plan), [testbed](http://en.wikipedia.org/wiki/Testbed) creation. Since many activities will be carried out during testing, a plan is needed.

Test development: Test procedures, [test scenarios](http://en.wikipedia.org/wiki/Scenario_test), [test cases](http://en.wikipedia.org/wiki/Test_case), 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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/w/index.php?title=Resolution_testing&action=edit&redlink=1).

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**

Main article: [Test automation](http://en.wikipedia.org/wiki/Test_automation)

Many programming groups are relying more and more on [automated testing](http://en.wikipedia.org/wiki/Test_automation), especially groups that use [test-driven development](http://en.wikipedia.org/wiki/Test-driven_development). There are many frameworks to write tests in, and [continuous integration](http://en.wikipedia.org/wiki/Continuous_integration) software will run tests automatically every time code is checked into a [version control](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Debugger). Testing/debug tools include features such as:

Program monitors, permitting full or partial monitoring of program code including:

[Instruction set simulator](http://en.wikipedia.org/wiki/Instruction_set_simulator), permitting complete instruction level monitoring and trace facilities

[Program animation](http://en.wikipedia.org/wiki/Program_animation), permitting step-by-step execution and conditional [breakpoint](http://en.wikipedia.org/wiki/Breakpoint) at source level or in [machine code](http://en.wikipedia.org/wiki/Machine_code)

[Code coverage](http://en.wikipedia.org/wiki/Code_coverage) reports

Formatted dump or [symbolic debugging](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Benchmark_(computing)), allowing run-time performance comparisons to be made

[Performance analysis](http://en.wikipedia.org/wiki/Performance_analysis) (or profiling tools) that can help to highlight [hot spots](http://en.wikipedia.org/wiki/Hot_spot_(computer_science)) and resource usage

Some of these features may be incorporated into an [Integrated Development Environment](http://en.wikipedia.org/wiki/Integrated_Development_Environment) (IDE).

**Measurement in software testing**

Main article: [Software quality](http://en.wikipedia.org/wiki/Software_quality)

Usually, quality is constrained to such topics as [correctness](http://en.wikipedia.org/wiki/Correctness_(computer_science)), completeness, [security](http://en.wikipedia.org/wiki/Computer_security_audit),[[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] but can also include more technical requirements as described under the [ISO](http://en.wikipedia.org/wiki/International_Organization_for_Standardization) standard [ISO/IEC 9126](http://en.wikipedia.org/wiki/ISO/IEC_9126), such as capability, [reliability](http://en.wikipedia.org/wiki/Reliability_engineering), [efficiency](http://en.wikipedia.org/wiki/Algorithmic_efficiency), [portability](http://en.wikipedia.org/wiki/Porting), [maintainability](http://en.wikipedia.org/wiki/Maintainability), compatibility, and [usability](http://en.wikipedia.org/wiki/Usability).

There are a number of frequently used [software metrics](http://en.wikipedia.org/wiki/Software_metric), or measures, which are used to assist in determining the state of the software or the adequacy of the testing.

**Testing artifacts**

The software testing process can produce several [artifacts](http://en.wikipedia.org/wiki/Artifact_(software_development)).

**Test plan**

A test specification is called a [test plan](http://en.wikipedia.org/wiki/Test_plan). 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](http://en.wikipedia.org/wiki/Test_strategy).

**Traceability matrix**

A [traceability matrix](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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. 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](http://en.wikipedia.org/wiki/Test_script) is a procedure, or programming code that replicates user actions. Initially the term was derived from the product of work created by automated regression test tools. 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](http://en.wikipedia.org/wiki/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**](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Test_harness).