Software Testing Techniques Classification

Various types of software testing performed to achieve different objectives when testing a software application are discussed as follows:

**Ad-hoc testing**

This type of software testing is very informal and unstructured and can be performed by any one with no reference to any test case or test design documents.
The person performing Ad-hoc testing has a good understanding of the domain and workflows of the application to be able to find defects and break the software. Ad-hoc testing is intended to find defects that were not found by existing test cases.

**Acceptance Testing**

Acceptance testing is a formal type of software testing that is performed by end user when the features have been delivered by developers. The aim of this testing is to check if the software confirms to their business needs and to the requirements provided earlier. Acceptance tests are normally documented at the beginning of the  set period of time during which specific work has to be completed and made ready for review, and is a means for testers and developers to work towards a common understanding and shared business domain knowledge.

**Accessibility Testing**

In accessibility testing, the aim of the testing is to determine if the contents of the website can be easily accessed by disable people. Various checks such as color and contrast, for color blind people, font size for visually impaired, clear and concise text that is easy to read and understand are being done.

**Automated testing**

This is a testing approach that makes use of testing tools and/or programming to run the test cases using software or custom developed test utilities. Most of the automated tools provide capture and playback facility, however there are tools that require writing extensive scripting or programming to automate test cases.

**All Pairs testing**

Also known as Pair wise testing, is a black box testing approach and a testing method where in for each input is tested in pairs of inputs, which helps to test software works as expected with all possible input combinations.

**Beta Testing**

This is a formal type of software testing that is carried out by end customers before releasing or handing over software to end users. Successful completion of Beta testing means customer acceptance of the software.

**Black Box testing**

Black box testing is a software testing method where in testers are not required to know coding or internal structure of the software. Black box testing method relies on testing software with various inputs and validating results against expected output.

**Backward Compatibility Testing**

Type of software testing performed to check newer version of the software can work successfully installed over previous version of the software and newer version of the software works as fine with table structure, data structures, files that were created by previous version of the software.

**Boundary Value Testing (BVT)**

Boundary Value Testing is a testing technique that is based on concept “error aggregates at boundaries”. In this testing technique, testing is done extensively to check for defects at boundary conditions. If a field accepts value 1 to 100 then testing is done for values 0, 1, 2, 99, 100 and 101.

**Bottom up Integration testing**

Bottom up integration testing is an integration testing approach where in testing starts with smaller pieces or sub systems of the software till all the way up covering entire software system. Bottom up integration testing begins with smaller portion of the software and eventually scale up in terms of size, complexity and completeness.

**Branch Testing**

It is a white box testing method for designing test cases to test code for every branching condition. Branch testing method is applied during unit testing.

**sCompatibility testing**

Compatibility testing is one of the test types performed by testing team. Compatibility testing checks if the software can be run on different hardware, operating system, bandwidth, databases, web servers, application servers, hardware peripherals, emulators, different configuration, processor, different browsers and different versions of the browsers etc.

**Component Testing**

This type of software testing is performed by developers. Component testing is carried out after completing unit testing. Component testing involves testing a group of units as code together as a whole rather than testing individual functions, methods.

**Condition Coverage Testing**

Condition coverage testing is a testing technique used during unit testing, where in developer tests for all the conditional statements like if, if else, case etc., in the code being unit tested.

**Dynamic Testing**

Dynamic Testing is a kind of software testing technique using which the dynamic behavior of the code is analyzed. For Performing dynamic, testing the software should be compiled and executed and parameters such as memory usage, CPU usage, response time and overall performance of the software are analyzed.

**Decision Coverage Testing**

Decision coverage or Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed. That is, every decision is taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application.

**End-to-end Testing**

End to end testing is performed by testing team. It is a technique used to test whether the flow of an application right from start to finish is behaving as expected. The purpose of performing end-to-end testing is to identify system dependencies and to ensure that the data integrity is maintained between various system components and systems.

**Equivalence Partitioning**

Equivalence partitioning (EP) is a specification-based or black-box technique. It can be applied at any level of testing and is often a good technique to use first.

The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence ‘equivalence partitioning’. Equivalence partitions are also known as equivalence classes – the two terms mean exactly the same thing.

In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

**Functional Testing**

Functional testing is a formal type of testing performed by testers. Functional testing focuses on testing software against design document, Use cases and requirements document. Functional testing is a black box type of testing and does not require internal working of the software unlike white box testing.

**Fuzz Testing**

Fuzz testing or fuzzing is a software testing technique that involves testing with unexpected or random inputs. Software is monitored for failures or error messages that are presented due to the input errors.

**GUI (Graphical User Interface) testing**

This type of software testing is aimed at testing the Graphical User Interface of the software to see if it meets the requirements as mentioned in the GUI mockups and Detailed designed documents. For e.g. checking the length and capacity of the input fields provided on the form, type of input field provided, e.g. some of the form fields can be displayed as dropdown box or a set of radio buttons. So GUI testing ensures GUI elements of the software are as per approved GUI mockups, detailed design documents and functional requirements. Most of the functional test automation tools work on GUI capture and playback capabilities. This makes script recording faster at the same time increases the effort on script maintenance.

**Integration Testing**

Integration testing is one of the important types of software testing. Once the individual units or components are tested by developers as working then testing team will run tests that will test the connectivity among these units/component or multiple units/components. There are different approaches for Integration testing namely, Top-down integration testing, Bottom-up integration testing and a combination of these two known as Sandwich testing.

**Internationalization Testing**

Internationalization testing is a type of testing that is performed by software testing team to check the extent to which software can support Internationalization i.e., usage of different languages, different character sets, double byte characters etc., For e.g.: Gmail, is a web application that is used by people all over world with different languages, single byte or multi byte character sets.

**Keyword-driven Testing**

Keyword driver testing  is a type of functional automation testing framework which is also known as table-driven testing or action word based testing. In Keyword-driven testing, we use a table format, usually a spreadsheet, to define keywords or action words for each function that we would like to execute.

**Load Testing**

Load testing is a type of non-functional testing; load testing is done to check the behavior of the software under normal and over peak load conditions. Load testing is usually performed using automated testing tools. Load testing intends to find bottlenecks or issues that prevent software from performing as intended at its peak workloads.

**Localization Testing**

Localization testing a type of software testing performed by software testers, in this type of testing, software is expected to adapt to a particular locale, it should support a particular locale/language in terms of display, accepting input in that particular locale, display, font, date time, currency etc., related to a particular locale. For e.g. many web applications allow choice of locale like English, French, German or Japanese. So once locale is defined or set in the configuration of software, software is expected to work as expected with a set language/locale.

**Negative Testing**

Negative testing is performed to ensure that the product or application under test does NOT fail when an unexpected input is given. The purpose of Negative testing is to break the system and to verify the application response during unintentional inputs. It is also a positive test for an error condition.

**Performance Testing**

It is a type of software testing and part of performance engineering that is performed to check some of the quality attributes of software like Stability, reliability, availability. Performance testing is carried out by performance engineering team. Unlike Functional testing, Performance testing is done to check non-functional requirements. Performance testing checks how well software works in anticipated and peak workloads. There are different variations or sub types of performance like load testing, stress testing, volume testing, soak testing and configuration testing.

**Regression Testing:** It is a type of software testing that is carried out by software testers as functional regression tests and developers as Unit regression tests. Objective of regression tests are to find defects that got introduced to defect fix(es) or introduction of new feature(s). Regression tests are ideal candidate for automation.

**Smoke testing**

is a type of testing that is carried out by software testers to check if the new build provided by development team is stable enough i.e., major functionality is working as expected in order to carry out further or detailed testing. Smoke testing is intended to find “show stopper” defects that can prevent testers from testing the application in detail. Smoke testing carried out for a build is also known as build verification test.

**Security Testing**

is a type of software testing carried out by specialized team of software testers. Objective of security testing is to secure the software to external or internal threats from humans and malicious programs. Security testing basically checks, how good is software’s authorization mechanism, how strong is authentication, how software maintains confidentiality of the data, how does the software maintain integrity of the data, what is the availability of the software in an event of an attack on the software by hackers and malicious programs. Security testing requires good knowledge of application, technology, networking, security testing tools. With increasing number of web applications necessity of security testing has increased to a greater extent.

**Scalability Testing**

is a nonfunctional test intended to test one of the software quality attributes i.e. “Scalability”. Scalability test is not focused on just one or few functionality of the software, instead, performance of software as a whole. Scalability testing is usually done by performance engineering team. Objective of scalability testing is to test the ability of the software to scale up with increased users, increased transactions, increase in database size etc., It is not necessary that software’s performance increases with increase in hardware configuration. Scalability tests helps to find out how much more workload the software can support with expanding user base, transactions, data storage etc.

**Stability Testing**

is a nonfunctional test intended to test one of the software quality attributes i.e. “Stability”. Stability testing focuses on testing how stable software is when it is subject to loads at acceptable levels, peak loads, loads generated in spikes, with more volumes of data to be processed. Scalability testing will involve performing different types of performance tests like load testing, stress testing, spike testing, soak testing, spike testing etc.

**Static Testing**

is a form of testing where in approaches like reviews, walkthroughs are employed to evaluate the correctness of the deliverable. In static testing software code is not executed instead it is reviewed for syntax, commenting, naming convention, size of the functions and methods etc. Static testing usually has check lists against which deliverables are evaluated. Static testing can be applied for requirements, designs, test cases by using approaches like reviews or walkthroughs.

**Stress Testing**

is a type of performance testing, in which software is subjected to peak loads and even to a break point to observe how the software would behave at breakpoint. Stress testing also tests the behavior of the software with insufficient resources like CPU, Memory, Network bandwidth, Disk space etc. Stress testing enables to check some of the quality attributes like robustness and reliability.

**System Testing**

This includes multiple software testing types that will enable to validate the software as a whole (software, hardware and network) against the requirements for which it was built. Different types of tests (GUI testing, Functional testing, Regression testing, Smoke testing, load testing, stress testing, security testing, stress testing, ad-hoc testing etc.,) are carried out to complete system testing.

**System Integration Testing**

known as SIT (in short) is a type of testing conducted by software testing team. As the name suggests, focus of System integration testing is to test for errors related to integration among different applications, services, third party vendor applications etc., As part of SIT, end-to-end scenarios are tested that would require software to interact (send or receive data) with other upstream or downstream applications, services, third party application calls etc.

**Unit testing**

is a type of testing that is performed by software developers. Unit testing follows white box testing approach where developer will test units of source code like statements, branches, functions, methods OR class, interface in OOP (object oriented programming). Unit testing usually involves in developing stubs and drivers. Unit tests are ideal candidates for automation. Automated tests can run as Unit regression tests on new builds or new versions of the software. There are many useful unit testing frames works like Junit, Nunit etc., available that can make unit testing more effective.

**Usability testing**

is a type of software testing that is performed to understand how user friendly the software is. Objective of usability testing is to allow end users to use the software, observe their behavior, their emotional response - whether users liked using software or were they stressed using it etc., and collect their feedback on how the software can be made more useable or user friendly, and incorporate the changes that make the software easier to use.

**User Acceptance testing (UAT )**

User Acceptance testing is a must for any project; it is performed by clients/end users of the software. User Acceptance testing allows SMEs (Subject matter experts) from client to test the software with their actual business or real-world scenarios and to check if the software meets their business requirements.

**Volume testing**

is a non-functional type of testing carried out by performance engineering team. Volume testing is one of the types of performance testing. Volume testing is carried out to find the response of the software with different sizes of the data being received or to be processed by the software. For e.g. If you were to be testing Microsoft word, volume testing would be to see if word can open, save and work on files of different sizes (10 to 100 MB).

**Vulnerability Testing**

involves identifying, exposing the software, hardware or network vulnerabilities that can be exploited by hackers and other malicious programs likes viruses or worms. Vulnerability Testing is key to software security and availability. With increase number of hackers and malicious programs, Vulnerability Testing is critical for success of a Business.

**White box Testing**

White box testing is also known as clear box testing, transparent box testing and glass box testing. White box testing is a software testing approach, which intends to test software with knowledge of internal working of the software. White box testing approach is used in Unit testing which is usually performed by software developers. White box testing intends to execute code and test statements, branches, path, decisions and data flow within the program being tested.

White box testing and Black box testing complement each other as each of the testing approaches have potential to un-cover specific category of errors

**White Box Testing**

White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality. It is a methodology to design the test cases that uses the control structure of the application to design test cases. White box testing is used to test the program code, code structure and the internal design flow.

A number of defects get amplified because of incorrect translation of requirements and design into program code. Let us see different techniques of white box testing.

Primarily White Box Testing comprises of two sub-streams of testing like:

**1) Static White box Testing**

 It is a type of testing in which the program source code is tested without running it. We only need to examine and review the code. It means that we need not execute the code. We need to find out whether:

a)The code works according to the functional requirements.

b)The code has been written in accordance with the design developed earlier in the project life cycle.

c) The code for any functionality has been missed out.

d) The code handles errors properly.

Static testing can be done by human being or with the help of specialized tools. Thus we can define static white box testing as a process involving a methodical & careful examination of the software architecture, basic design or its code with a view to hunt for bugs without executing it. It is called structural analysis as well sometimes.

**2) Dynamic white box testing**

This involves testing a running program. So, now binaries and executables are desired. We try to test the internal logic of the program now. It entails running the actual product against some pre-designed test cases to exercise as much of the code as possible.
Some of the key Dynamic White Box Testing processes are as under:

1. **Unit / Code Functional Testing**

It is the process of testing in which the developer performs some quick checks prior to subjecting the code to more extensive code coverage testing or code complexity testing. It can be performed in many ways

 a) At the initial stages, the developer or tester can perform certain tests based on the input variables and the corresponding expected output variables. This can be a quick test. If we repeat these tests for multiple values of input variables also then the confidence level of the developer to go to the next level increases.

 b) For complex modules, the tester can insert some print statements in between to check whether the program control passes through all statements and loops. It is important to remove the intermediate print statements after the defects are fixed.

c) Another method is to run the product under a debugger or an Integrated Development Environment (IDE). These tools involve single stepping of instructions, setting break points at any function or instruction.

All these initial tests, actually fall under "Debugging" category rather than under “'Testing" category of activities. These are placed under "White Box Testing" head as all are related to the knowledge of code structure.

1. **Code Coverage Testing**

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage of code covered by a test is found by adopting a technique called as the instrumentation of code. These tools rebuild the code, do product linking with a set of libraries provided by the tool, monitor the portions of code covered, reporting on the portions of the code that are covered frequently, so that the critical or most-often portions of code can be identified.

**Black Box Testing**

Black box testing treats the system as a **“black-box”**, so it doesn’t explicitly use Knowledge of the internal structure or code. Or in other words the Test engineer need not know the internal working of the “Black box” or application

1. **Static Black-Box Testing: Testing the Specification**

Testing the specification is static black-box testing. The specification is a document, not an executing program, so it's considered static. It's also something that was created using data from many sources usability studies, focus groups, marketing input, and so on. It is not necessary to know how or why that information was obtained or the details of the process used to obtain it, just that it's been boiled down into a product specification. That document can then be taken to perform static black-box testing, and carefully examine it for bugs.

You can test a specification with static black-box techniques no matter what the format of the specification. It can be a written or graphical document or a combination of both. You can even test an unwritten specification by questioning the people who are designing and writing the software.

### Dynamic Black-Box Testing

Testing software without having an insight into the details of underlying code is dynamic black-box testing. It's dynamic because the program is running you're using it as a customer would. And, it's black-box because you're testing it without knowing exactly how it works with blinders on.

 You're entering inputs, receiving outputs, and checking the results. Another name commonly used for dynamic black-box testing is behavioral testing because you're testing how the software actually behaves when it's used.

To do this effectively requires some definition of what the software does namely, a requirements document or product specification. One does not need to know what happens inside the software "box" you just need to know that inputting A outputs B or that performing operation C results in D. A good product spec will provide these details.

Once the ins and outs of the software to be tested is known, the next step is to start defining the test cases. Test cases are the specific inputs to be tried and the procedures to be followed when the software is to be tested.

**Test case Design Technique**

Following are the typical design techniques in software engineering:

1. Deriving test cases directly from a requirement specification or black box test design technique. The Techniques include:

* Boundary Value Analysis (BVA)
* Equivalence Partitioning (EP)
* Decision Table Testing
* State Transition Diagrams
* Use Case Testing

2. Deriving test cases directly from the structure of a component or system:

* Statement Coverage
* Branch Coverage
* Path Coverage

3. Deriving test cases based on tester's experience on similar systems or testers intuition:

* Error Guessing
* Exploratory Testing

**Additional Black Box Testing Approaches**

Typical black-box test design techniques include:

* Decision table testing
* Equivalence partitioning
* Boundary value analysis
* Cause-effect graph
* Error guessing
* Use case testing
1. **Decision table testing**

Decision tables are a precise yet compact way to model complex rule sets and their corresponding actions.

Decision tables, like flowcharts, if-then-else, and switch-case statements, associate conditions with actions to perform.

The structure is a s follows:

**The four quadrants**

|  |  |
| --- | --- |
| Conditions | Condition alternatives |
| Actions | Action entries |

Each decision corresponds to a variable, relation or predicate whose possible values are listed among the condition alternatives.

Each action is a procedure or operation to perform, and the entries specify whether (or in what order) the action is to be performed for the set of condition alternatives the entry corresponds to. Many decision tables include in their condition alternatives the don't care symbol, a hyphen. Using don't cares can simplify decision tables, especially when a given condition has little influence on the actions to be performed. In some cases, entire conditions thought to be important initially are found to be irrelevant when none of the conditions influence which actions are performed.

Aside from the basic four quadrant structure, decision tables vary widely in the way the condition alternatives and action entries are represented.  Some decision tables use simple true/false values to represent the alternatives to a condition (akin to if-then-else), other tables may use numbered alternatives (akin to switch-case), and some tables even use fuzzy logic or probabilistic representations for condition alternatives. In a similar way, action entries can simply represent whether an action is to be performed (check the actions to perform), or in more advanced decision tables, the sequencing of actions to perform (number the actions to perform).

**Example**

The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.

A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.

The following is a **balanced decision table**

|  |  |  |
| --- | --- | --- |
|  |  | **Rules** |
| **Conditions** | Printer does not print | Y | Y | Y | Y | N | N | N | N |
| A red light is flashing | Y | Y | N | N | Y | Y | N | N |
| Printer is unrecognized | Y | N | Y | N | Y | N | Y | N |
| **Actions** | Check the power cable |  |  | X |  |  |  |  |  |
| Check the printer-computer cable | X |  | X |  |  |  |  |  |
| Ensure printer software is installed | X |  | X |  | X |  | X |  |
| Check/replace ink | X | X |  |  | X | X |  |  |
| Check for paper jam |  | X |  | X |  |  |  |  |

**Advantages**

1. Decision tables, especially when coupled with the use of a domain-specific language, allow developers and policy experts to work from the same information, the decision tables themselves.
2. Tools to render nested if statements from traditional programming languages into decision tables can also be used as a debugging tool.
3. Decision tables are easier to understand and review than code, and have been used extensively and successfully to produce specifications for complex systems.
4. **Equivalence partitioning**

Please refer to lecture 1 (Software testing techniques classification).

1. **Boundary Value Analysis**

It  is a software testing technique in which tests are designed to include representatives of boundary values in a range. The idea comes from the boundary. Given that we have a set of test vectors to test the system, a topology can be defined on that set. Those inputs which belong to the same equivalence class as defined by the equivalence partitioning theory would constitute the basis. Given that the basis sets are neighbors, there would exist a boundary between them. The test vectors on either side of the boundary are called boundary values. In practice this would require that the test vectors can be ordered, and that the individual parameters follows some kind of order (either partial order or total order).

1. **Cause-effect graph**

In software testing, a cause–effect graph is a directed graph that maps a set of causes to a set of effects. The causes may be thought of as the input to the program, and the effects may be thought of as the output. Usually the graph shows the nodes representing the causes on the left side and the nodes representing the effects on the right side. There may be intermediate nodes in between that combine inputs using logical operators such as AND and OR.

Constraints may be added to the causes and effects. These are represented as edges labeled with the constraint symbol using a dashed line. For causes, valid constraint symbols are:

* 1. **E (exclusive):** The exclusive constraint states that at most one of the causes 1 and 2 can be true, i.e. both cannot be true simultaneously.
	2. **(one and only one):** The one and only one (OaOO or simply O) constraint states that only one of the causes 1, 2 or 3 can be true.
	3. **I (at least one)**: The Inclusive (at least one) constraint states that at least one of the causes 1, 2 or 3 must be true, i.e. all cannot be false simultaneously
	4. **R (Requires):** The Requires constraint states that if cause 1 is true, then cause 2 must be true, and it is impossible for 1 to be true and 2 to be false

For effects, valid constraint symbol is M (Mask). The mask constraint states that if effect 1 is true then effect 2 is false. The mask constraint relates to the effects and not the causes like the other constraints.

The graph's direction is as follows:

**Causes --> intermediate nodes --> Effects**

The graph can always be rearranged so there is only one node between any input and any output..

A cause–effect graph is useful for generating a reduced decision table.

The Cause-Effect Diagram can be used under these Circumstances:

* To determine the current problem so that right decision can be taken very fast.
* To narrate the connections of the system with the factors affecting a particular process or effect.
* To recognize the probable root causes, the cause for a exact effect, problem, or outcome.

## **Symbols used in Cause-effect graphs:**



Just assume that each node is having the value 0 or 1 where 0 shows the ‘absent state’ and 1 shows the ‘present state’.

* The identity function states when c1 = 1, e1 = 1 or we can say if c0 = 0 and e0 = 0.
* The NOT function states that, if C1 = 1, e1= 0 and vice-versa.
* Likewise, OR function states that, if C1 or C2 or C3 = 1, e1 = 1 else e1 = 0.
* The AND function states that, if both C1 and C2 = 1, e1 = 1, else e1 = 0.
* The AND and OR functions are permitted to have any number of inputs.

**Example**

# Test cases design for the triangle problem

1. **Recognize and describe the input conditions (causes) and actions (effect).**

**The causes allocated by letter “C” are as follows,**

C1: Side “x” is less than sum of “y” and “z”

C2: Side “y” is less than sum of “x” and “z”

C3: Side “z” is less then sum of “x” and “y”

C4: Side “x” is equal to side “y”

C5: Side “x” is equal to side “z”

C6: Side “y” is equal to side “z”

**The effects designated by letter “e” are as follows,**

e1: Not a triangle

e2: Scalene triangle

e3: Isosceles triangle.

e4: Equilateral triangle

e5: Impossible

1. **Build up a cause-effect graph**



1. **11 test cases according to the 11 rules.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **X** | **Y** | **Z** | **Expected Result** |
| 1 | 4 | 1 | 2 | Not a triangle |
| 2 | 1 | 4 | 2 | Not a triangle |
| 3 | 1 | 2 | 4 | Not a triangle |
| 4 | 5 | 5 | 5 | Equilateral |
| 5 | ? | ? | ? | Impossible |
| 6 | ? | ? | ? | Impossible |
| 7 | 2 | 2 | 3 | Isosceles |
| 8 | ? | ? | ? | Impossible |
| 9 | 2 | 3 | 2 | Isosceles |
| 10 | 3 | 2 | 2 | Isosceles |
| 11 | 3 | 4 | 5 | Scalene |

**Advantages**

* It finds out the areas where data is collected for additional study.
* It motivates team contribution and uses the team data of the process.
* Uses synchronize and easy to read format to diagram cause-and-effect relationships.
* Point out probable reasons of difference in a process.
* It enhances facts of the procedure by helping everyone to learn more about the factors at work and how they relate.
* It assists us to decide the root reasons of a problem or quality using a structured approach.
1. **Error guessing**

It  is a test method in which test cases used to find bugs in programs are established based on experience in prior testing. The scope of test cases usually rely on the software tester involved, who uses past experience and intuition to determine what situations commonly cause software failure, or may cause errors to appear. Typical errors include divide by zero, null pointers, or invalid parameters.

Error guessing has no explicit rules for testing; test cases can be designed depending on the situation, either drawing from functional documents or when an unexpected/undocumented error is found while testing operations.

1. **Use-Case testing**

In software and systems engineering, a **use case** is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an *actor*) and a system, to achieve a goal. The actor can be a human or other external system. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. The detailed requirements may then be captured in the Systems Modeling Language (SysML) or as contractual statements.

Use case analysis is an important and valuable requirement analysis technique that has been widely used in modern software engineering since its formal introduction by Ivar Jacobson in 1992. With its inherent iterative, incremental and evolutionary nature, use case also fits well for agile development.

**Advantages**

* 1. **User focused**

Use cases constitute a powerful, user-centric tool for the software requirements specification process. Use case modeling typically starts from identifying key stakeholder roles (*actors*) interacting with the system, and their goals or objectives the system must fulfill (an outside perspective). These user goals then become the ideal candidates for the names or titles of the use cases which represent the desired functional features or services provided by the system. This user-centered approach ensure that what has the real business value and the user really want is developed, not those trivial functions speculated from a developer or system (inside) perspective.

* 1. **Better communication**

Use cases are often written in natural languages with structured templates. This narrative textual form (legible requirement stories), understandable by almost everyone, complemented by visual UML diagrams foster better and deeper communications among all stakeholders, including customers, end-users, developers, testers and managers. Better communications result in quality requirements and thus quality systems delivered.

* 1. **Quality requirements by structured exploration**

One of the most powerful things about use cases reside in the formats of the use case templates, especially the main success scenario (basic flow) and the extension scenario fragments (extensions, exceptional and/or alternative flows). Analyzing a use case step by step from preconditions to postconditions, exploring and investigating every action step of the use case flows, from basic to extensions, to identify those tricky, normally hidden and ignored, seemingly trivial but realistically often costly requirements (as Cockburn mentioned above), is a structured and beneficial way to get clear, stable and quality requirements systematically.

Minimizing and optimizing the action steps of a use case to achieve the user goal also contribute to a better interaction design and user experience of the system.

* 1. **Facilitate testing and user documentation**

With content based upon an action or event flow structure, a model of well-written use cases also serves as an excellent groundwork and valuable guidelines for the design of test cases and user manuals of the system or product, which is an effort-worthy investment up-front. There is obvious connections between the flow paths of a use case and its test cases. Deriving functional test cases from a use case through its scenarios (running instances of a use case) is straightforward.

**Limitations**

Use cases are not well suited for capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or non-functional requirements(such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.

* As there are no fully standard definitions of use cases, each project must form its own interpretation.
* Some use case relationships, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand.
* Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case. While use case theory suggests that UI not be reflected in use cases, it can be awkward to abstract out this aspect of design, as it makes the use cases difficult to visualize. In software engineering, this difficulty is resolved by applying requirements traceability, for example with a [traceability matrix](https://en.wikipedia.org/wiki/Traceability_matrix). Another approach to associate UI elements with use cases, is to attach a UI design to each step in the use case. This is called a use case storyboard.
* Use cases are a starting point for test design, but since each test needs its own success criteria, use cases may need to be modified to provide separate post-conditions for each path

**Additional Black Box Testing Approaches**  continued…

1. **Error guessing**

It  is a test method in which test cases used to find bugs in programs are established based on experience in prior testing. The scope of test cases usually rely on the software tester involved, who uses past experience and intuition to determine what situations commonly cause software failure, or may cause errors to appear. Typical errors include divide by zero, null pointers, or invalid parameters.

Error guessing has no explicit rules for testing; test cases can be designed depending on the situation, either drawing from functional documents or when an unexpected/undocumented error is found while testing operations.

1. **Use-Case testing**

In software and systems engineering, a **use case** is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an *actor*) and a system, to achieve a goal. The actor can be a human or other external system. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. The detailed requirements may then be captured in the Systems Modeling Language (SysML) or as contractual statements.

Use case analysis is an important and valuable requirement analysis technique that has been widely used in modern software engineering since its formal introduction by Ivar Jacobson in 1992. Use case driven development is a key characteristic of many process models and frameworks such as ICONIX, the Unified Process (UP), the IBM Rational Unified Process (RUP), and the Oracle Unified Method (OUM). With its inherent iterative, incremental and evolutionary nature, use case also fits well for agile development.

**Advantages**

* 1. **User focused**

Use cases constitute a powerful, user-centric tool for the software requirements specification process.[[16]](https://en.wikipedia.org/wiki/Use_case#cite_note-16) Use case modeling typically starts from identifying key stakeholder roles (*actors*) interacting with the system, and their goals or objectives the system must fulfill (an outside perspective). These user goals then become the ideal candidates for the names or titles of the use cases which represent the desired functional features or services provided by the system. This user-centered approach ensure that what has the real business value and the user really want is developed, not those trivial functions speculated from a developer or system (inside) perspective.

* 1. **Better communication**

Use cases are often written in natural languages with structured templates. This narrative textual form (legible requirement stories), understandable by almost everyone, complemented by visual UML diagrams foster better and deeper communications among all stakeholders, including customers, end-users, developers, testers and managers. Better communications result in quality requirements and thus quality systems delivered.

* 1. **Quality requirements by structured exploration**

One of the most powerful things about use cases reside in the formats of the use case templates, especially the main success scenario (basic flow) and the extension scenario fragments (extensions, exceptional and/or alternative flows). Analyzing a use case step by step from preconditions to postconditions, exploring and investigating every action step of the use case flows, from basic to extensions, to identify those tricky, normally hidden and ignored, seemingly trivial but realistically often costly requirements (as Cockburn mentioned above), is a structured and beneficial way to get clear, stable and quality requirements systematically.

Minimizing and optimizing the action steps of a use case to achieve the user goal also contribute to a better interaction design and user experience of the system.

* 1. **Facilitate testing and user documentation**

With content based upon an action or event flow structure, a model of well-written use cases also serves as an excellent groundwork and valuable guidelines for the design of test cases and user manuals of the system or product, which is an effort-worthy investment up-front. There is obvious connections between the flow paths of a use case and its test cases. Deriving functional test cases from a use case through its scenarios (running instances of a use case) is straightforward.

**Limitations**

* Use cases are not well suited for capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or non-functional requirements(such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.
* As there are no fully standard definitions of use cases, each project must form its own interpretation.
* Some use case relationships, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand.
* Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case. While use case theory suggests that UI not be reflected in use cases, it can be awkward to abstract out this aspect of design, as it makes the use cases difficult to visualize. In software engineering, this difficulty is resolved by applying requirements traceability, for example with a [traceability matrix](https://en.wikipedia.org/wiki/Traceability_matrix). Another approach to associate UI elements with use cases, is to attach a UI design to each step in the use case. This is called a use case storyboard.
* Use cases are a starting point for test design, but since each test needs its own success criteria, use cases may need to be modified to provide separate post-conditions for each path

**Test case selection and adequacy criteria**

**Definition OF Terms**

1. Test case: a set of inputs, execution conditions, and a pass/fail criterion.
2. Test case specification: a requirement to be satisfied by one or more test cases.
3. Test obligation: a partial test case specification, requiring some property deemed important to thorough testing.
4. Test suite: a set of test cases.
5. Test or test execution: the activity of executing test cases and evaluating their results.
6. Adequacy criterion: a predicate that is true (satisfied) or false (not satisfied) of a 〈program, test suite〉 pair.

The design of software testing can be a challenging process. The objective of testing is to have the highest likelihood of finding the most errors with a minimum amount of timing and effort. A large number of test case design methods have been developed that offer the developer with a systematic approach to testing. Methods offer an approach that can ensure the completeness of tests and offer the highest likelihood for uncovering errors in software.

A Test Case Includes:

* Input
* the expected output
* pass/fail criteria
* the environment in which the test is being conducted.

Here the term input means all the data that is required to make a test case. A Test Case specification is a requirement to be satisfied by one or more test cases. Specification-based testing uses the specification of the program as the point of reference for test input data selection and adequacy. A test specification can be drawn from system, interface or program. The distinction between a test case and test specification is similar to the distinction between program specification and program.

 Software products can be tested in two ways:

 (1) Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational

 (2) knowing the internal workings of a product, tests can be performed to see if they jell.

The first test approach is known as a black box testing and the second white box testing. Software Test cases derived from test specifications based on system requirements are generally termed as functional or black box testing.

Software Test cases derived from specifications of interface and programs are generally termed as glass box or white box testing.

Test cases should uncover errors like:

* Comparison of different data types
* Incorrect logical operators are precedence
* Expectation of equality when precision error makes equality unlikely
* Incorrect comparison or variables
* Improper or non-existent loop termination.
* Failure to exit when divergent iteration is encountered
* Improperly modified loop variables

Good design dictates that error conditions be anticipated and error handling paths set up to reroute or cleanly terminate processing when an error does occur.

TEST ADEQUECY

* Adequacy criterion = set of test obligations

 • A test suite satisfies an adequacy criterion

 if – all the tests succeed (pass)

 Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite

 – Example:

the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”.

Test adequacy criteria is useful to

 • Test selection approaches

– Guidance in devising a thorough test suite

 • Example:

A specification-based criterion may suggest test cases covering combinations of values

 • Revealing missing tests

 – Post hoc analysis:

What might I have missed with this test suite?

 • Often in combination

– Example:

Design test suite from specifications, then use structural criterion (e.g., coverage of all branches) to highlight missed logic

• Adequacy criteria provide a way to define a notion of “thoroughness” in a test suite – But they don’t offer guarantees; more like design rules to highlight inadequacy

 • Defined in terms of “covering” some information

 – Derived from many sources: Specs, code, models, ...

• May be used for selection as well as measurement – With caution! An aid to thoughtful test design, not a substitute

**Structural Testing**

In structural testing, the software is viewed as a white box and test cases are determined from the implementation of the software. Structural testing techniques include:

1. Control flow testing
2. Data flow testing.

**Advantages of Structural Testing:**

* Forces test developer to reason carefully about implementation
* Reveals errors in "hidden" code
* Spots the Dead Code or other issues with respect to best programming practices.

**Disadvantages of Structural Box Testing:**

* Expensive as one has to spend both time and money to perform white box testing.
* Every possibility that few lines of code is missed accidentally.
* In-depth knowledge about the programming language is necessary to perform white box testing.

**Mutation Testing**

Mutation testing is a structural testing technique, which uses the structure of the code to guide the testing process. On a very high level, it is the process of rewriting the source code in small ways in order to remove the redundancies in the source code

These ambiguities might cause failures in the software if not fixed and can easily pass through testing phase undetected.

**Mutation Testing Benefits:**

* It brings a whole new kind of errors to the developer's attention.
* It is the most powerful method to detect hidden defects, which might be impossible to identify using the conventional testing techniques.
* Tools such as Insure++ help us to find defects in the code using the state-of-the-art.
* Increased customer satisfaction index as the product would be less buggy.
* Debugging and Maintaining the product would be more easier than ever.

**Mutation Testing Types:**

* **Value Mutations:** An attempt to change the values to detect errors in the programs. We usually change one value to a much larger value or one value to a much smaller value. The most common strategy is to change the constants.
* **Decision Mutations:** The decisions/conditions are changed to check for the design errors. Typically, one changes the arithmetic operators to locate the defects and also we can consider mutating all relational operators and logical operators (AND, OR , NOT)
* **Statement Mutations:** Changes done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.

**Code Coverage**

Code Coverage testing is determining how much code is being tested. It can be calculated using the formula:

Code Coverage = (Number of lines of code exercised)/(Total Number of lines of code) \* 100%

Following are the types of code coverage Analysis:

* **Function coverage** - Has each function (or subroutine) in the program been called?
* **Statement coverage** - Has each statement in the program been executed?
* **Branch coverage** - Has each branch (also called DD-path) of each control structure (such as in *if* and *case* statements) been executed? For example, given an *if* statement, have both the true and false branches been executed? Another way of saying this is, has every edge in the program been executed?
* **Condition coverage** (or predicate coverage) - Has each Boolean sub-expression evaluated both to true and false?