Software Testing Tutorial

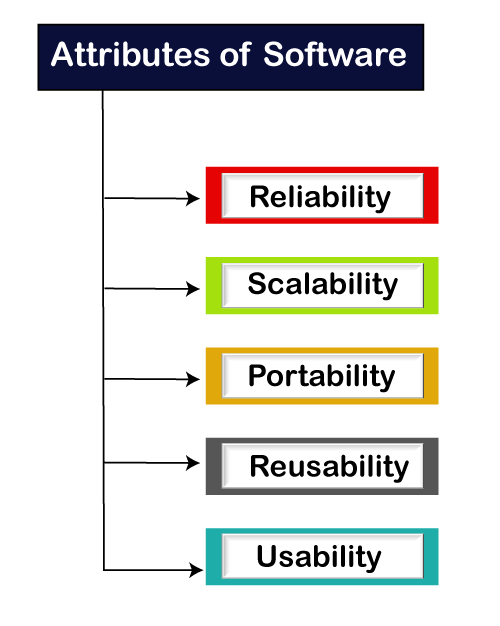
Software testing tutorial provides basic and advanced concepts of software testing. Our software testing tutorial is designed for beginners and professionals.

Software testing is widely used technology because it is compulsory to test each and every software before deployment.

Our Software testing tutorial includes all topics of Software testing such as Methods such as Black Box Testing, White Box Testing, Visual Box Testing and Gray Box Testing. Levels such as Unit Testing, Integration Testing, Regression Testing, Functional Testing. System Testing, Acceptance Testing, Alpha Testing, Beta Testing, Non-Functional testing, Security Testing, Portability Testing.

What is Software Testing

Software testing is a process of identifying the correctness of software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.



Software testing provides an independent view and objective of the software and gives surety of fitness of the software. It involves testing of all components under the required services to confirm that whether it is satisfying the specified requirements or not. The process is also providing the client with information about the quality of the software.

Testing is mandatory because it will be a dangerous situation if the software fails any of time due to lack of testing. So, without testing software cannot be deployed to the end user.

What is Testing

Testing is a group of techniques to determine the correctness of the application under the predefined script but, testing cannot find all the defect of application. The main intent of testing is to detect failures of the application so that failures can be discovered and corrected. It does not demonstrate that a product functions properly under all conditions but only that it is not working in some specific conditions.

Testing furnishes comparison that compares the behavior and state of software against mechanisms because the problem can be recognized by the mechanism. The mechanism may include past versions of the same specified product, comparable products, and interfaces of expected purpose, relevant standards, or other criteria but not limited up to these.

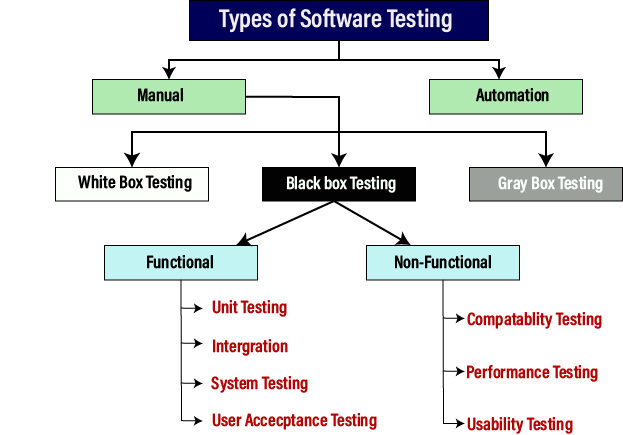
Testing includes an examination of code and also the execution of code in various environments, conditions as well as all the examining aspects of the code. In the current scenario of software development, a testing team may be separate from the development team so that Information derived from testing can be used to correct the process of software development.

The success of software depends upon acceptance of its targeted audience, easy graphical user interface, strong functionality load test, etc. For example, the audience of banking is totally different from the audience of a video game. Therefore, when an organization develops a software product, it can assess whether the software product will be beneficial to its purchasers and other audience.

Type of Software testing

We have various types of testing available in the market, which are used to test the application or the software.

With the help of below image, we can easily understand the type of software testing:



Manual testing

The process of checking the functionality of an application as per the customer needs without taking any help of automation tools is known as manual testing. While performing the manual testing on any application, we do not need any specific knowledge of any testing tool, rather than have a proper understanding of the product so we can easily prepare the test document.

Manual testing can be further divided into three types of testing, which are as follows:

* **White box testing**
* **Black box testing**
* **Gray box testing**

For more information about manual testing, refers to the below link:

Automation testing

Automation testing is a process of converting any manual test cases into the test scripts with the help of automation tools, or any programming language is known as automation testing. With the help of automation testing, we can enhance the speed of our test execution because here, we do not require any human efforts. We need to write a test script and execute those scripts.

For more information about manual testing, refers to the below link:

Prerequisite

Before learning software testing, you should have basic knowledge of basic computer functionality, basic mathematics, computer language, and logical operators.

Audience

Our software testing tutorial is designed for beginners and professionals.

Problems

We assure that you will not find any problem in this Software Testing Tutorial. But if there is any mistake, please post the problem in contact form.

Difference between Bug, Defect, Error, Fault & Failure

In this section, we are going to discuss the difference between the **Bug, Defect, Error, Fault & Failure** as we understood that all the terms are used whenever the system or an application act abnormally.

Sometimes we call it an **error** and sometimes a bug or a **defect** and so on. In software testing, many of the new test engineers have confusion in using these terminologies.

Generally, we used these terms in the [Software Development Life Cycle (SDLC)](https://www.javatpoint.com/software-development-life-cycle) based on the phases. But there is a conflict in the usage of these terms.

In other words, we can say that in the era of **software testing,** the terms **bugs, defects, error, fault, and failure** come across every second of the day.

But for a beginner or the inexperienced in this field, all these terminologies may seem synonyms. It became essential to understand each of these terms independently if the software doesn't work as expected.

What is a bug?

In [software testing](https://www.javatpoint.com/software-testing-tutorial), a [bug](https://www.javatpoint.com/bug-in-software-testing) is the informal name of defects, which means that software or application is not working as per the requirement. When we have some coding error, it leads a program to its breakdown, which is known as **a bug**. The **test engineers** use the terminology **Bug**.

If a [**QA (Quality Analyst)**](https://www.javatpoint.com/quality-assurance) detect a bug, they can reproduce the bug and record it with the help of the **bug report template**.

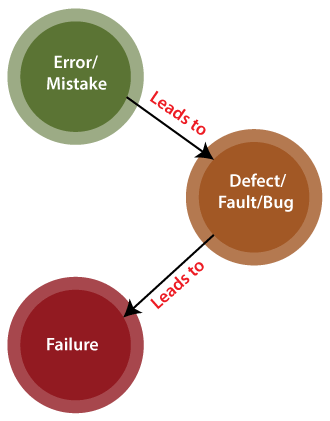
What is a Defect?

When the application is not working as per the requirement is knows as **defects**. It is specified as the aberration from the **actual and expected result** of the application or software.

In other words, we can say that the bug announced by the **programmer** and inside the code is called a [**Defect**](https://www.javatpoint.com/defect-or-bug-tracking-tool)**.**

What is Error?

The Problem in code leads to errors, which means that a mistake can occur due to the developer's coding error as the developer misunderstood the requirement or the requirement was not defined correctly. The **developers** use the term **error**.



What is Fault?

The fault may occur in software because it has not added the code for fault tolerance, making an application act up.

A fault may happen in a program because of the following reasons:

* Lack of resources
* An invalid step
* Inappropriate data definition

What is Failure?

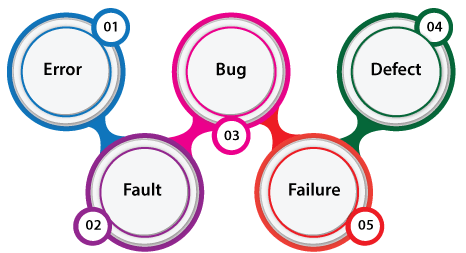
Many defects lead to the **software's failure**, which means that a loss specifies a fatal issue in software/ application or in its module, which makes the system unresponsive or broken.

In other words, we can say that if an end-user detects an issue in the product, then that particular issue is called a **failure**.

Possibilities are there one defect that might lead to one failure or several failures.

**For example**, in a bank application if the **Amount Transfer** module is not working for end-users when the end-user tries to **transfer money**, submit button is not working. Hence, this is a **failure**.

The flow of the above terminologies are shown in the following image:



Bug Vs. Defect Vs. Error Vs. Fault Vs. Failure

We have listed some of the vital differences between **bug, defect, error, fault, and failure in the below table**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Comparison basis** | **Bug** | **Defect** | **Error** | **Fault** | **Failure** |
| **Definition** | It is an informal name specified to the defect. | The **Defect** is the difference between the actual outcomes and expected outputs. | An **Error** is a mistake made in the code; that's why we cannot execute or compile code. | The **Fault** is a state that causes the software to fail to accomplish its essential function. | If the software has lots of defects, it leads to failure or causes failure. |
| **Raised by** | The **Test Engineers** submit the bug. | The **Testers** identify the defect. And it was also solved by the developer in the development phase or stage. | The **Developers and automation test engineers** raise the error. | **Human mistakes** cause fault. | The failure finds by the manual test engineer through the **development cycle**. |
| **Different types** | Different type of bugs are as follows:   * Logic bugs * Algorithmic bugs * Resource bugs | Different type of Defects are as follows: Based on **priority**:   * High * Medium * Low   And based on the severity:   * Critical * Major * Minor * Trivial | Different type of Error is as below:   * Syntactic Error * User interface error * Flow control error * Error handling error * Calculation error * Hardware error * Testing Error | Different type of Fault are as follows:   * Business Logic Faults * Functional and Logical Faults * Faulty GUI * Performance Faults * Security Faults * Software/ hardware fault | ----- |
| **Reasons behind** | Following are reasons which may cause the **bugs:** Missing coding Wrong coding Extra coding | The below reason leads to the **defects**: Giving incorrect and wrong inputs. Dilemmas and errors in the outside behavior and inside structure and design. An error in coding or logic affects the software and causes it to breakdown or the failure. | The reasons for having an **error** are as follows: Errors in the code. The Mistake of some values. If a developer is unable **to compile or run a program successfully.** Confusions and issues in programming. Invalid login, loop, and syntax. Inconsistency between actual and expected outcomes. Blunders in design or requirement actions. Misperception in understanding the requirements of the application. | The reasons behind the **fault** are as follows: A Fault may occur by an improper step in the initial stage, process, or data definition. Inconsistency or issue in the program. An irregularity or loophole in the software that leads the software to perform improperly. | Following are some of the most important reasons behind the **failure:** Environmental condition System usage Users Human error |
| **Way to prevent the reasons** | Following are the way to stop the **bugs**: Test-driven development. Offer programming language support. Adjusting, advanced, and operative development procedures. Evaluating the code systematically. | With the help of the following, we can prevent the **Defects**: Implementing several innovative programming methods. Use of primary and correct software development techniques. Peer review It is executing consistent code reviews to evaluate its quality and correctness. | Below are ways to prevent the **Errors**: Enhance the software quality with system review and programming. Detect the issues and prepare a suitable mitigation plan. Validate the fixes and verify their quality and precision. | The **fault** can be prevented with the help of the following: Peer review. Assess the functional necessities of the software. Execute the detailed code analysis. Verify the correctness of software design and programming. | The way to prevent **failure** are as follows: Confirm re-testing. Review the requirements and revisit the specifications. Implement current protective techniques. Categorize and evaluate errors and issues. |

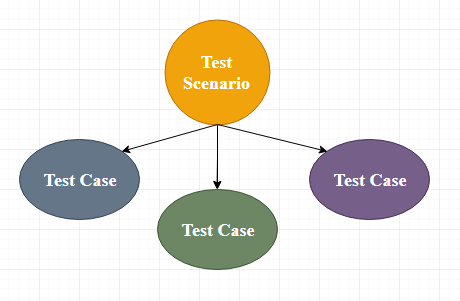
Conclusion

After seeing all the significant differences between **bug, defect, error, fault, and failure**, we can say that the several issues and inconsistencies found throughout software are linked and dependent on each other.

All the above terminology affects and change different parts of the software and differ from one another massively. However, all these differences between **bug, defect, errors, faults, and failures** slow down the software's excellence and performance.

# Test Case

The test case is defined as a group of conditions under which a tester determines whether a software application is working as per the customer's requirements or not. Test case designing includes preconditions, case name, input conditions, and expected result. A test case is a first level action and derived from test scenarios.



It is an in-details document that contains all possible inputs (positive as well as negative) and the navigation steps, which are used for the test execution process. Writing of test cases is a one-time attempt that can be used in the future at the time of regression testing.

Test case gives detailed information about testing strategy, testing process, preconditions, and expected output. These are executed during the testing process to check whether the software application is performing the task for that it was developed or not.

Test case helps the tester in defect reporting by linking defect with test case ID. Detailed test case documentation works as a full proof guard for the testing team because if developer missed something, then it can be caught during execution of these full-proof test cases.

To write the test case, we must have the requirements to derive the inputs, and the test scenarios must be written so that we do not miss out on any features for testing. Then we should have the test case template to maintain the uniformity, or every test engineer follows the same approach to prepare the test document.

Generally, we will write the test case whenever the developer is busy in writing the code.

## When do we write a test case?

We will write the test case when we get the following:

* When the customer gives the business needs then, the developer starts developing and says that they need 3.5 months to build this product.
* And In the meantime, the testing team will **start writing the test cases**.
* Once it is done, it will send it to the Test Lead for the review process.
* And when the developers finish developing the product, it is handed over to the testing team.
* The test engineers never look at the requirement while testing the product document because testing is constant and does not depends on the mood of the person rather than the quality of the test engineer.

#### **Note: When writing the test case, the actual result should never be written as the product is still being in the development process. That?s why the actual result should be written only after the execution of the test cases.**

## Why we write the test cases?

We will write the test for the following reasons:

* **To require consistency in the test case execution**
* **To make sure a better test coverage**
* **It depends on the process rather than on a person**
* **To avoid training for every new test engineer on the product**

**To require consistency in the test case execution:** we will see the test case and start testing the application.

**To make sure a better test coverage:** for this, we should cover all possible scenarios and document it, so that we need not remember all the scenarios again and again.

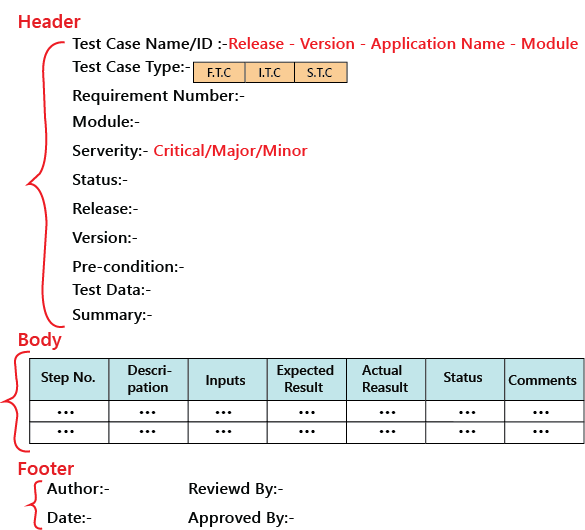
**It depends on the process rather than on a person:** A test engineer has tested an application during the first release, second release, and left the company at the time of third release. As the test engineer understood a module and tested the application thoroughly by deriving many values. If the person is not there for the third release, it becomes difficult for the new person. Hence all the derived values are documented so that it can be used in the future.

**To avoid giving training for every new test engineer on the product:** When the test engineer leaves, he/she leaves with a lot of knowledge and scenarios. Those scenarios should be documented so that the new test engineer can test with the given scenarios and also can write the new scenarios.

#### **Note: when the developers are developing the first product for the First release, the test engineer writes the test cases. And in the second release, when the new features are added, the test engineer writes the test cases for that also, and in the next release, when the elements are modified, the test engineer will change the test cases or writes the new test cases as well.**

## Test case template

The primary purpose of writing a test case is to achieve the efficiency of the application.



As we know, the **actual result** is written after the test case execution, and most of the time, it would be same as the **expected result**. But if the test step will fail, it will be different. So, the actual result field can be skipped, and in **the Comments** section, we can write about the bugs.

And also, the **Input field** can be removed, and this information can be added to the **Description field**.

The above template we discuss above is not the standard one because it can be different for each company and also with each application, which is based on the test engineer and the test lead. But, for testing one application, all the test engineers should follow a usual template, which is formulated.

The test case should be written in simple language so that a new test engineer can also understand and execute the same.

In the above sample template, the header contains the following:

**Step number**

It is also essential because if step number 20 is failing, we can document the bug report and hence prioritize working and also decide if it’s a critical bug.

**Test case type**

It can be functional, integration or system test cases or positive or negative or positive and negative test cases.

**Release**

One release can contain many versions of the release.

**Pre-condition**

These are the necessary conditions that need to be satisfied by every test engineer before starting the test execution process. Or it is the data configuration or the data setup that needs to be created for the testing.

**For example**: In an application, we are writing test cases to add users, edit users, and delete users. The per-condition will be seen if user A is added before editing it and removing it.

**Test data**

These are the values or the input we need to create as per the per-condition.

**For example**, Username, Password, and account number of the users.

The test lead may be given the test data like username or password to test the application, or the test engineer may themself generate the username and password.

**Severity**

The severity can be **major, minor, and critical**, the severity in the test case talks about the importance of that particular test cases. All the text execution process always depends on the severity of the test cases.

We can choose the severity based on the module. There are many features include in a module, even if one element is critical, we claim that test case to be critical. It depends on the functions for which we are writing the test case.

**For example,** we will take the Gmail application and let us see the severity based on the modules:

|  |  |
| --- | --- |
| **Modules** | **Severity** |
| Login | Critical |
| Help | Minor |
| Compose mail | Critical |
| Setting | Minor |
| Inbox | Critical |
| Sent items | Major |
| Logout | Critical |

And for the banking application, the severity could be as follows:

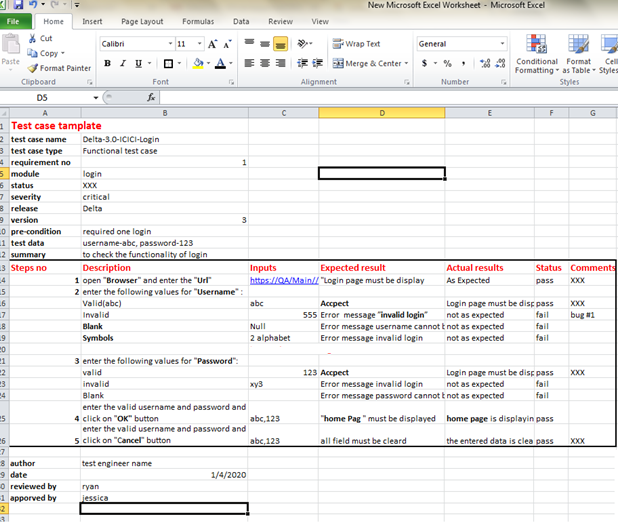
|  |  |
| --- | --- |
| **Modules** | **Severity** |
| Amount transfer | critical |
| Feedback | minor |

**Brief description**

The test engineer has written a test case for a particular feature. If he/she comes and reads the test cases for the moment, he/she will not know for what feature has written it. So, the brief description will help them in which feature test case is written.

## Example of a test case template

Here, we are writing a test case for the **ICICI application’s Login** module:



## Types of test cases

We have a different kind of test cases, which are as follows:

* **Function test cases**
* **Integration test cases**
* **System test cases**

### The functional test cases

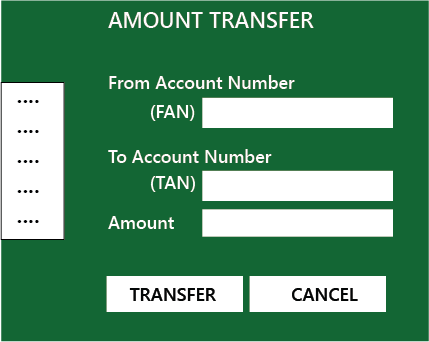
Firstly, we check for which field we will write test cases and then describe accordingly.

In functional testing or if the application is data-driven, we require the input column else; it is a bit time-consuming.

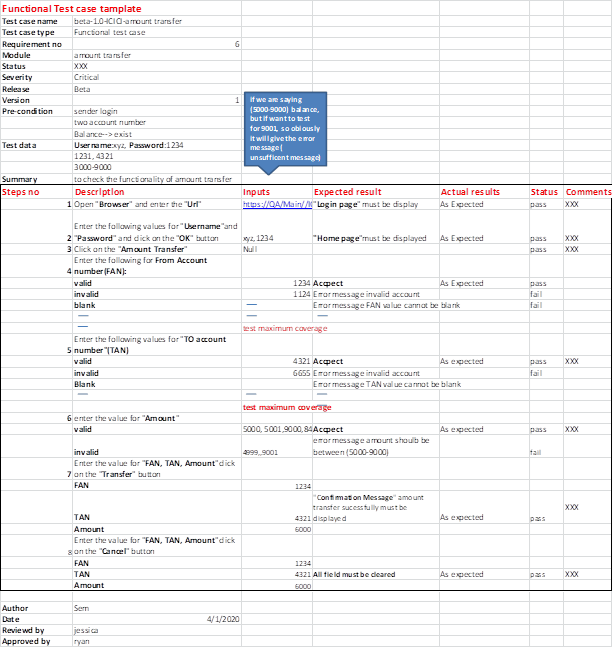
**Rules to write functional test cases:**

* In the expected results column, try to use **should be** or **must be**.
* Highlight the Object names.
* We have to describe only those steps which we required the most; otherwise, we do not need to define all the steps.
* To reduce the excess execution time, we will write steps correctly.
* Write a generic test case; do not try to hard code it.

Let say it is the amount transfer module, so we are writing the functional test cases for it and then also specifies that it is not a login feature.



The functional test case for amount transfer module is in the below Excel file:



### Integration test case

In this, we should not write something which we already covered in the functional test cases, and something we have written in the integration test case should not be written in the system test case again.

**Rules to write integration test cases**

* Firstly, understand the product
* Identify the possible scenarios
* Write the test case based on the priority

When the test engineer writing the test cases, they may need to consider the following aspects:

If the test cases are in details:

* They will try to achieve maximum test coverage.
* All test case values or scenarios are correctly described.
* They will try to think about the execution point of view.
* The template which is used to write the test case must be unique.

#### **Note: when we involve fewer numbers of steps or coverage is more, it should be the best test case, and when these test cases are given to anyone, they will understand easily.**

### System test cases

We will write the system test cases for the end-to-end business flows. And we have the entire modules ready to write the system test cases.

## The process to write test cases

The method of writing a test case can be completed into the following steps, which are as below:



**System study**

In this, we will understand the application by looking at the requirements or the SRS, which is given by the customer.

**Identify all scenarios:**

* When the product is launched, what are the possible ways the end-user may use the software to identify all the possible ways.
* I have documented all possible scenarios in a document, which is called test design/high-level design.
* The test design is a record having all the possible scenarios.

**Write test cases**

Convert all the identified scenarios to test claims and group the scenarios related to their features, prioritize the module, and write test cases by applying test case design techniques and use the standard test case template, which means that the one which is decided for the project.

**Review the test cases**

Review the test case by giving it to the head of the team and, after that, fix the review feedback given by the reviewer.

**Test case approval**

After fixing the test case based on the feedback, send it again for the approval.

**Store in the test case repository**

After the approval of the particular test case, store in the familiar place that is known as the test case repository.

# Dynamic Testing

In this section, we are going to understand **Dynamic testing**, which is done when the code is executed in the run time environment.

And we also learn about Dynamic testing, **why we use it, how to perform it, what are a different technique for Dynamic testing, various tools for Dynamic Testing**.

Why do we need to perform Dynamic Testing?

We can easily understand how to implement dynamic testing during the [STLC [Software Testing Life Cycle]](https://www.javatpoint.com/software-testing-life-cycle) if we consider the characteristics accessible by dynamic testing.

Using dynamic testing, the team can verify the software's critical features, but some of those can be left without any assessment. And they can also affect the functioning, reliability, and performance of the software product.

Hence, we can perform **Dynamic testing** to fulfill the various below aspects:

* We will perform dynamic testing to check whether the application or software is working fine during and after installing the application without any error.
* We can perform dynamic testing to verify the efficient behavior of the software.
* The software should be compiled and run if we want to perform dynamic testing.
* Generally, Dynamic Testing is implemented to define the dynamic behavior of code.
* The team implements the code to test the software application's performance in a run-time environment during the dynamic testing process.
* It makes sure that the concurrency of the software application with the customer's potentials, needs and the end-user.
* It is an operative technique to measure the effect of several environmental stresses on the software application like **network, hardware**

Characteristic of Dynamic Testing

For understanding the fundamental of the **software testing techniques**, we have to learn their attribute and several other components. Hence, following are some of the important characteristics of **dynamic testing**:

* It is implemented throughout the **validation stage** of software testing.
* Dynamic Testing is done by performing the program.
* Both functional and non-functional testing include in dynamic testing.
* In Dynamic testing, we can easily identify the bugs for the particular software.
* It helps the team in validating the reliability of the software application.
* Unlike static testing, the team implements the software's code to get expected outputs in dynamic testing.
* Dynamic testing is performed directly on the software application as compare to other testing techniques.
* Dynamic testing is a more formal testing approach for different testing activities such as **test execution, coverage consideration, reporting and test case identification.**

Black box testing

Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.

Black box testing

Generic steps of black box testing

* The black box test is based on the specification of requirements, so it is examined in the beginning.
* In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
* In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
* The fourth phase includes the execution of all test cases.
* In the fifth step, the tester compares the expected output against the actual output.
* In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

It does not require programming knowledge of the software. All test cases are designed by considering the input and output of a particular function.A tester knows about the definite output of a particular input, but not about how the result is arising. There are various techniques used in black box testing for testing like decision table technique, boundary value analysis technique, state transition, All-pair testing, cause-effect graph technique, equivalence partitioning technique, error guessing technique, use case technique and user story technique. All these techniques have been explained in detail within the tutorial.

Test cases

Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.

Techniques Used in Black Box Testing

|  |  |
| --- | --- |
| [Decision Table Technique](https://www.javatpoint.com/decision-table-technique-in-black-box-testing) | Decision Table Technique is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form. It is appropriate for the functions that have a logical relationship between two and more than two inputs. |
| [Boundary Value Technique](https://www.javatpoint.com/boundary-value-analysis-in-black-box-testing) | Boundary Value Technique is used to test boundary values, boundary values are those that contain the upper and lower limit of a variable. It tests, while entering boundary value whether the software is producing correct output  or not. |
| [State Transition Technique](https://www.javatpoint.com/state-transition-technique-in-black-box-testing) | State Transition Technique is used to capture the behavior of the software application when different input  values are given to the same function. This applies to those types of applications that provide the specific number  of attempts to access the application. |
| [All-pair Testing Technique](https://www.javatpoint.com/all-pairs-testing-technique-in-black-box-testing) | All-pair testing Technique is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input, list box, text box, etc. |
| [Cause-Effect Technique](https://www.javatpoint.com/cause-and-effect-graph-technique-in-black-box-testing) | Cause-Effect Technique underlines the relationship between a given result and all the factors affecting the  result.It is based on a collection of requirements. |
| [Equivalence Partitioning Technique](https://www.javatpoint.com/equivalence-partitioning-technique-in-black-box-testing) | Equivalence partitioning is a technique of software testing in which input data divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior. |
| [Error Guessing Technique](https://www.javatpoint.com/error-guessing-technique-in-black-box-testing) | Error guessing is a technique in which there is no specific method for identifying the error. It is based on the experience of the test analyst, where the tester uses the experience to guess the problematic areas of the software. |
| [Use Case Technique](https://www.javatpoint.com/use-case-technique-in-black-box-testing) | Use case Technique used to identify the test cases from the beginning to the end of the system as per the usage  of the system. By using this technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end. |

# Boundary Value Analysis

Boundary value analysis is one of the widely used case design technique for black box testing. It is used to test boundary values because the input values near the boundary have higher chances of error.

Whenever we do the testing by boundary value analysis, the tester focuses on, while entering boundary value whether the software is producing correct output or not.

Boundary values are those that contain the upper and lower limit of a variable. Assume that, age is a variable of any function, and its minimum value is 18 and the maximum value is 30, both 18 and 30 will be considered as boundary values.

The basic assumption of boundary value analysis is, the test cases that are created using boundary values are most likely to cause an error.

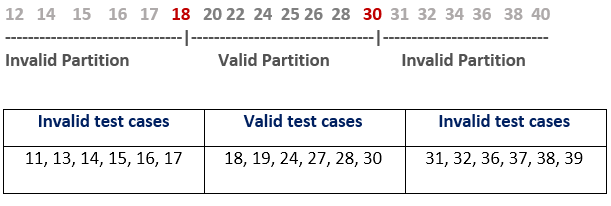
There is 18 and 30 are the boundary values that's why tester pays more attention to these values, but this doesn't mean that the middle values like 19, 20, 21, 27, 29 are ignored. Test cases are developed for each and every value of the range.

Boundary Value Analysis

Testing of boundary values is done by making valid and invalid partitions. Invalid partitions are tested because testing of output in adverse condition is also essential.

**Let's understand via practical:**

Imagine, there is a function that accepts a number between 18 to 30, where 18 is the minimum and 30 is the maximum value of valid partition, the other values of this partition are 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29. The invalid partition consists of the numbers which are less than 18 such as 12, 14, 15, 16 and 17, and more than 30 such as 31, 32, 34, 36 and 40. Tester develops test cases for both valid and invalid partitions to capture the behavior of the system on different input conditions.



The software system will be passed in the test if it accepts a valid number and gives the desired output, if it is not, then it is unsuccessful. In another scenario, the software system should not accept invalid numbers, and if the entered number is invalid, then it should display error massage.

If the software which is under test, follows all the testing guidelines and specifications then it is sent to the releasing team otherwise to the development team to fix the defects.

White Box Testing

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**. It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing. In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Developers do white box testing. In this, the developer will test every line of the code of the program. The developers perform the White-box testing and then send the application or the software to the testing team, where they will perform the [black box testing](https://www.javatpoint.com/black-box-testing) and verify the application along with the requirements and identify the bugs and sends it to the developer.

The developer fixes the bugs and does one round of white box testing and sends it to the testing team. Here, fixing the bugs implies that the bug is deleted, and the particular feature is working fine on the application.

Here, the test engineers will not include in fixing the defects for the following reasons:

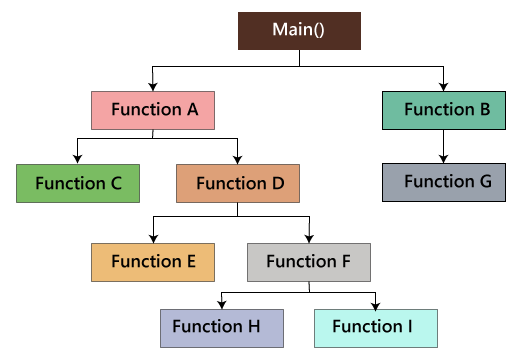
* Fixing the bug might interrupt the other features. Therefore, the test engineer should always find the bugs, and developers should still be doing the bug fixes.
* If the test engineers spend most of the time fixing the defects, then they may be unable to find the other bugs in the application.

The white box testing contains various tests, which are as follows:

* Path testing
* Loop testing
* Condition testing
* Testing based on the memory perspective
* Test performance of the program

Path testing

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:



And test all the independent paths implies that suppose a path from main() to function G, first set the parameters and test if the program is correct in that particular path, and in the same way test all other paths and fix the bugs.

Loop testing

In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

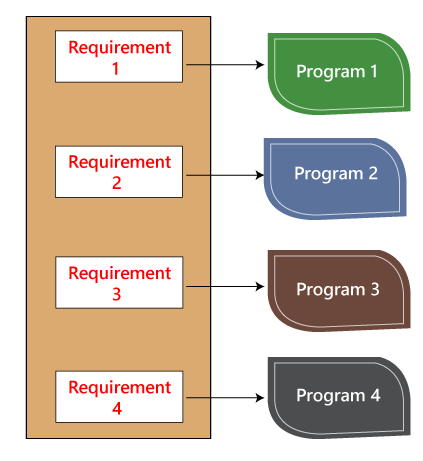
**For example**: we have one program where the developers have given about 50,000 loops.

1. {
2. while(50,000)
3. ……
4. ……
5. }

We cannot test this program manually for all the 50,000 loops cycle. So we write a small program that helps for all 50,000 cycles, as we can see in the below program, that test P is written in the similar language as the source code program, and this is known as a Unit test. And it is written by the developers only.

1. Test P
2. {
3. ……
4. …… }

As we can see in the below image that, we have various requirements such as 1, 2, 3, 4. And then, the developer writes the programs such as program 1,2,3,4 for the parallel conditions. Here the application contains the 100s line of codes.

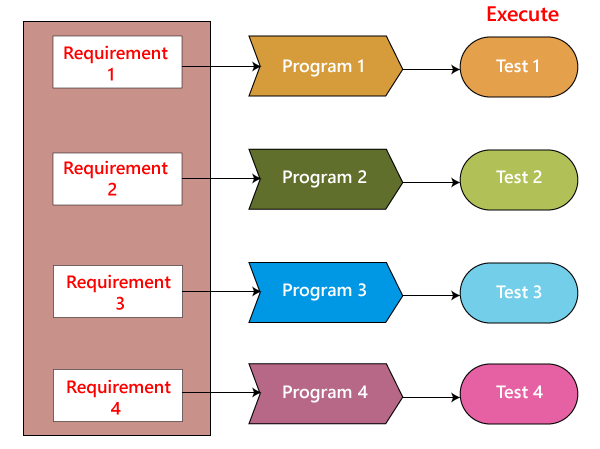


The developer will do the white box testing, and they will test all the five programs line by line of code to find the bug. If they found any bug in any of the programs, they will correct it. And they again have to test the system then this process contains lots of time and effort and slows down the product release time.

Now, suppose we have another case, where the clients want to modify the requirements, then the developer will do the required changes and test all four program again, which take lots of time and efforts.

**These issues can be resolved in the following ways:**

In this, we will write test for a similar program where the developer writes these test code in the related language as the source code. Then they execute these test code, which is also known as **unit test programs**. These test programs linked to the main program and implemented as programs.



Therefore, if there is any requirement of modification or bug in the code, then the developer makes the adjustment both in the main program and the test program and then executes the test program.

Condition testing

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and**else** condition.

**For example:**

1. if(condition) - true
2. {
3. …..
4. ……
5. ……
6. }
7. else - false
8. {
9. …..
10. ……
11. ……
12. }

The above program will work fine for both the conditions, which means that if the condition is accurate, and then else should be false and conversely.

Testing based on the memory (size) perspective

The size of the code is increasing for the following reasons:

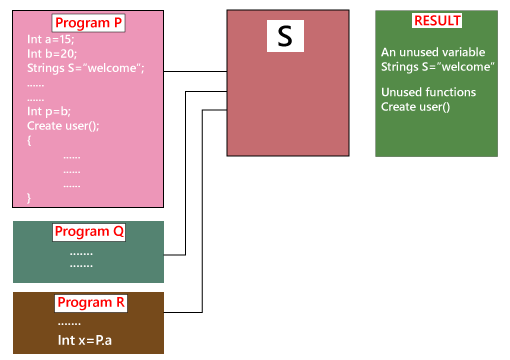
* **The reuse of code is not there**: let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.
* The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
* The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase.

**For example**,

1. Int a=15;
2. Int b=20;
3. String S= "Welcome";
4. ….
5. …..
6. …..
7. ….
8. …..
9. Int p=b;
10. Create user()
11. {
12. ……
13. ……
14. ….. 200's line of code
15. }

In the above code, we can see that the **integer a** has never been called anywhere in the program, and also the function **Create user** has never been called anywhere in the code. Therefore, it leads us to memory consumption.

We cannot remember this type of mistake manually by verifying the code because of the large code. So, we have a built-in tool, which helps us to test the needless variables and functions. And, here we have the tool called **Rational purify**.



Suppose we have three programs such as Program P, Q, and R, which provides the input to S. And S goes into the programs and verifies the unused variables and then gives the outcome. After that, the developers will click on several results and call or remove the unnecessary function and the variables.

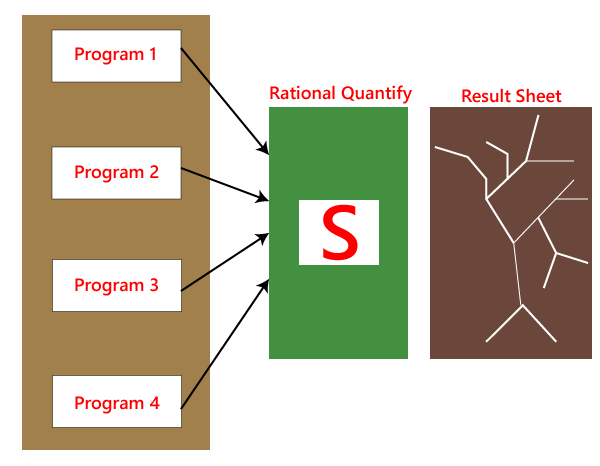
This tool is only used for the [C programming language](https://www.javatpoint.com/c-programming-language-tutorial) and [C++ programming language](https://www.javatpoint.com/cpp-tutorial); for another language, we have other related tools available in the market.

* The developer does not use the available in-built functions; instead they write the full features using their logic. Therefore, it leads us to waste of time and also postpone the product releases.

Test the performance (Speed, response time) of the program

The application could be slow for the following reasons:

* When logic is used.
* For the conditional cases, we will use **or** & **and** adequately.
* Switch case, which means we cannot use **nested if**, instead of using a switch case.

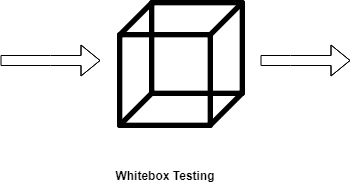


As we know that the developer is performing white box testing, they understand that the code is running slow, or the performance of the program is also getting deliberate. And the developer cannot go manually over the program and verify which line of the code is slowing the program.

To recover with this condition, we have a tool called **Rational Quantify**, which resolves these kinds of issues automatically. Once the entire code is ready, the rational quantify tool will go through the code and execute it. And we can see the outcome in the result sheet in the form of thick and thin lines.

Here, the thick line specifies which section of code is time-consuming. When we double-click on the thick line, the tool will take us to that line or piece of code automatically, which is also displayed in a different color. We can change that code and again and use this tool. When the order of lines is all thin, we know that the presentation of the program has enhanced. And the developers will perform the white box testing automatically because it saves time rather than performing manually.

Test cases for white box testing are derived from the design phase of the software development lifecycle. Data flow testing, control flow testing, path testing, branch testing, statement and decision coverage all these techniques used by white box testing as a guideline to create an error-free software.



White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Generic steps of white box testing

* Design all test scenarios, test cases and prioritize them according to high priority number.
* This step involves the study of code at runtime to examine the resource utilization, not accessed areas of the code, time taken by various methods and operations and so on.
* In this step testing of internal subroutines takes place. Internal subroutines such as nonpublic methods, interfaces are able to handle all types of data appropriately or not.
* This step focuses on testing of control statements like loops and conditional statements to check the efficiency and accuracy for different data inputs.
* In the last step white box testing includes security testing to check all possible security loopholes by looking at how the code handles security.

Reasons for white box testing

* It identifies internal security holes.
* To check the way of input inside the code.
* Check the functionality of conditional loops.
* To test function, object, and statement at an individual level.

Advantages of White box testing

* White box testing optimizes code so hidden errors can be identified.
* Test cases of white box testing can be easily automated.
* This testing is more thorough than other testing approaches as it covers all code paths.
* It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

* White box testing is too much time consuming when it comes to large-scale programming applications.
* White box testing is much expensive and complex.
* It can lead to production error because it is not detailed by the developers.
* White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation.

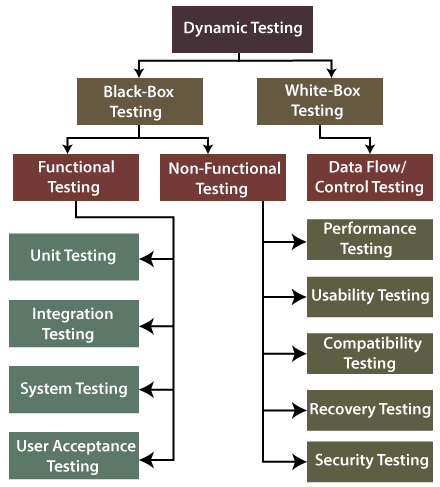
|  |  |
| --- | --- |
| [Data Flow Testing](https://www.javatpoint.com/data-flow-testing-in-white-box-testing) | Data flow testing is a group of testing strategies that examines the control flow of programs in order to explore the sequence of variables according to the sequence of events. |
| [Control Flow Testing](https://www.javatpoint.com/control-flow-testing-in-white-box-testing) | Control flow testing determines the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the tester to set the testing path. Test cases represented by the control graph of the program. |
| [Branch Testing](https://www.javatpoint.com/branch-coverage-testing-in-white-box-testing) | Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once. |
| [Statement Testing](https://www.javatpoint.com/statement-coverage-testing-in-white-box-testing) | Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code, out of total statements present in the source code. |
| [Decision Testing](https://www.javatpoint.com/decision-coverage-testing-in-white-box-testing) | This technique reports true and false outcomes of Boolean expressions. Whenever there is a possibility of two or more outcomes from the statements like do while statement, if statement and case statement (Control flow statements), it is considered as decision point because there are two outcomes either true or false. |

Techniques Used in White Box Testing

Difference between white-box testing and black-box testing

Following are the significant differences between white box testing and black box testing:

|  |  |
| --- | --- |
| **White-box testing** | **Black box testing** |
| The developers can perform white box testing. | The test engineers perform the black box testing. |
| To perform WBT, we should have an understanding of the programming languages. | To perform BBT, there is no need to have an understanding of the programming languages. |
| In this, we will look into the source code and test the logic of the code. | In this, we will verify the functionality of the application based on the requirement specification. |
| In this, the developer should know about the internal design of the code. | In this, there is no need to know about the internal design of the code. |



Unit Testing

Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

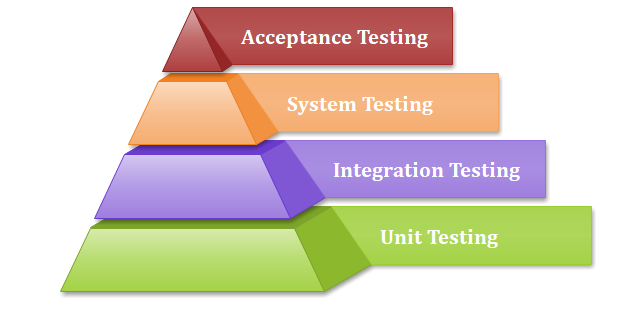
A unit is a single testable part of a software system and tested during the development phase of the application software.

The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers.

Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as **Unit testing** or **components testing**.

Why Unit Testing?

In a testing level hierarchy, unit testing is the first level of testing done before integration and other remaining levels of the testing. It uses modules for the testing process which reduces the dependency of waiting for Unit testing frameworks, stubs, drivers and mock objects are used for assistance in unit testing.



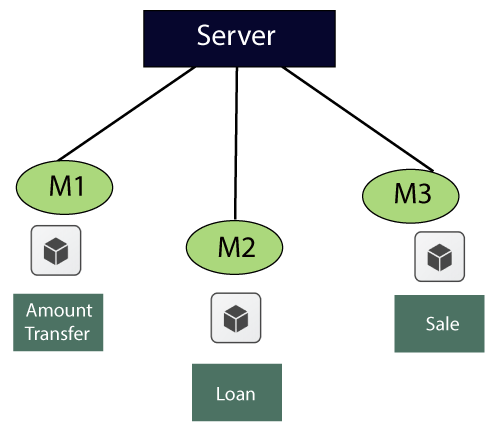
Generally, **the** software goes under four level of testing: Unit Testing, Integration Testing, System Testing, and Acceptance Testing but sometimes due to time consumption software testers does minimal unit testing but skipping of unit testing may lead to higher defects during Integration Testing, System Testing, and Acceptance Testing or even during Beta Testing which takes place after the completion of software application.

**Some crucial reasons are listed below:**

* Unit testing helps tester and developers to understand the base of code that makes them able to change defect causing code quickly.
* Unit testing helps in the documentation.
* Unit testing fixes defects very early in the development phase that's why there is a possibility to occur a smaller number of defects in upcoming testing levels.
* It helps with code reusability by migrating code and test cases.

Example of Unit testing

Let us see one sample example for a better understanding of the concept of unit testing:



For the **amount transfer,** requirements are as follows:

|  |  |
| --- | --- |
| 1. | Amount transfer |
| 1.1 | From account number (FAN)→ Text Box |
| 1.1.1 | FAN→ accept only 4 digit |
| 1.2 | To account no (TAN)→ Text Box |
| 1.2.1 | TAN→ Accept only 4 digit |
| 1.3 | Amount→ Text Box |
| 1.3.1 | Amount → Accept maximum 4 digit |
| 1.4 | Transfer→ Button |
| 1.4.1 | Transfer → Enabled |
| 1.5 | Cancel→ Button |
| 1.5.1 | Cancel→ Enabled |

Below are the application access details, which is given by the customer

* URL→ login Page
* Username/password/OK → home page
* To reach Amount transfer module follow the below

**Loans → sales → Amount transfer**

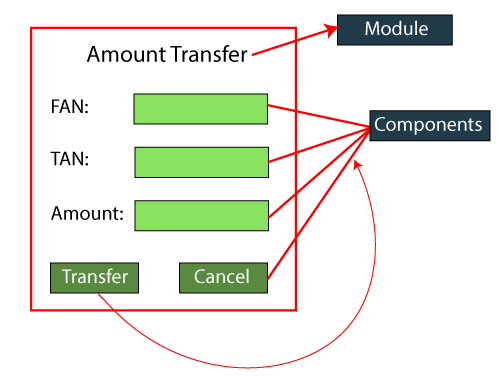
While performing unit testing, we should follow some rules, which are as follows:

* To start unit testing, at least we should have one module.
* Test for positive values
* Test for negative values
* No over testing
* No assumption required

When we feel that the **maximum test coverage** is achieved, we will stop the testing.

Now, we will start performing the unit testing on the different components such as

* **From account number(FAN)**
* **To account number(TAN)**
* **Amount**
* **Transfer**
* **Cancel**



**For the FAN components**

|  |  |
| --- | --- |
| **Values** | **Description** |
| 1234 | accept |
| 4311 | Error message→ account valid or not |
| blank | Error message→ enter some values |
| 5 digit/ 3 digit | Error message→ accept only 4 digit |
| Alphanumeric | Error message → accept only digit |
| Blocked account no | Error message |
| Copy and paste the value | Error message→ type the value |
| Same as FAN and TAN | Error message |

**For the TAN component**

* Provide the values just like we did in **From account number** (FAN) components

**For Amount component**

* Provide the values just like we did in FAN and TAN components.

**For Transfer component**

* Enter valid FAN value
* Enter valid TAN value
* Enter the correct value of Amount
* Click on the Transfer button→ amount transfer successfully( confirmation message)

**For Cancel Component**

* Enter the values of FAN, TAN, and amount.
* Click on the Cancel button → all data should be cleared.

Unit Testing Tools

We have various types of unit testing tools available in the market, which are as follows:

* NUnit
* JUnit
* PHPunit
* Parasoft Jtest
* EMMA

For more information about Unit testing tools, refers to the below link:

<https://www.javatpoint.com/unit-testing-tools>

Unit Testing Techniques:

Unit testing uses all white box testing techniques as it uses the code of software application:

* Data flow Testing
* Control Flow Testing
* Branch Coverage Testing
* Statement Coverage Testing
* Decision Coverage Testing

How to achieve the best result via Unit testing?

Unit testing can give best results without getting confused and increase complexity by following the steps listed below:

* Test cases must be independent because if there is any change or enhancement in requirement, the test cases will not be affected.
* Naming conventions for unit test cases must be clear and consistent.
* During unit testing, the identified bugs must be fixed before jump on next phase of the SDLC.
* Only one code should be tested at one time.
* Adopt test cases with the writing of the code, if not doing so, the number of execution paths will be increased.
* If there are changes in the code of any module, ensure the corresponding unit test is available or not for that module.

Advantages and disadvantages of unit testing

The pros and cons of unit testing are as follows:

Advantages

* Unit testing uses module approach due to that any part can be tested without waiting for completion of another parts testing.
* The developing team focuses on the provided functionality of the unit and how functionality should look in unit test suits to understand the unit API.
* Unit testing allows the developer to refactor code after a number of days and ensure the module still working without any defect.

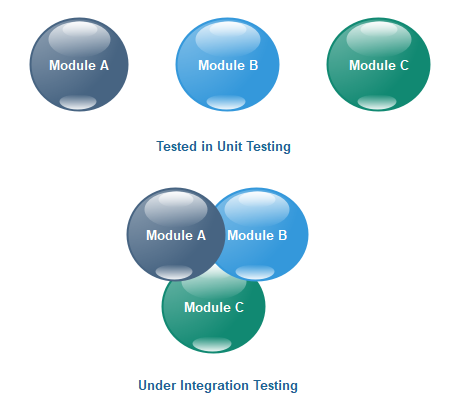
Disadvantages

* It cannot identify integration or broad level error as it works on units of the code.
* In the unit testing, evaluation of all execution paths is not possible, so unit testing is not able to catch each and every error in a program.
* It is best suitable for conjunction with other testing activities.

# Integration testing

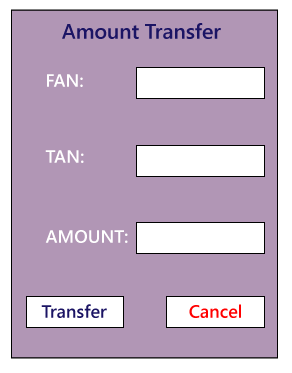
Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

[Unit testing](https://www.javatpoint.com/unit-testing) uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.



Once all the components or modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Let us see one sample example of a banking application, as we can see in the below image of amount transfer.



* First, we will login as a user **P** to amount transfer and send Rs200 amount, the confirmation message should be displayed on the screen as **amount transfer successfully**. Now logout as P and login as user **Q** and go to amount balance page and check for a balance in that account = Present balance + Received Balance. Therefore, the integration test is successful.
* Also, we check if the amount of balance has reduced by Rs200 in P user account.
* Click on the transaction, in P and Q, the message should be displayed regarding the data and time of the amount transfer.

## Guidelines for Integration Testing

* We go for the integration testing only after the functional testing is completed on each module of the application.
* We always do integration testing by picking module by module so that a proper sequence is followed, and also we don't miss out on any integration scenarios.
* First, determine the test case strategy through which executable test cases can be prepared according to test data.
* Examine the structure and architecture of the application and identify the crucial modules to test them first and also identify all possible scenarios.
* Design test cases to verify each interface in detail.
* Choose input data for test case execution. Input data plays a significant role in testing.
* If we find any bugs then communicate the bug reports to developers and fix defects and retest.
* Perform **positive and negative integration testing**.

Here **positive** testing implies that if the total balance is Rs15, 000 and we are transferring Rs1500 and checking if the amount transfer works fine. If it does, then the test would be a pass.

And **negative testing** means, if the total balance is Rs15, 000 and we are transferring Rs20, 000 and check if amount transfer occurs or not, if it does not occur, the test is a pass. If it happens, then there is a bug in the code, and we will send it to the development team for fixing that bug.

#### **Note: Any application in this world will do functional testing compulsory, whereas integration testing will be done only if the modules are dependent on each other. Each integration scenarios should compulsorily have source→ data→destination. Any scenarios can be called as integration scenario only if the data gets saved in the destination.**

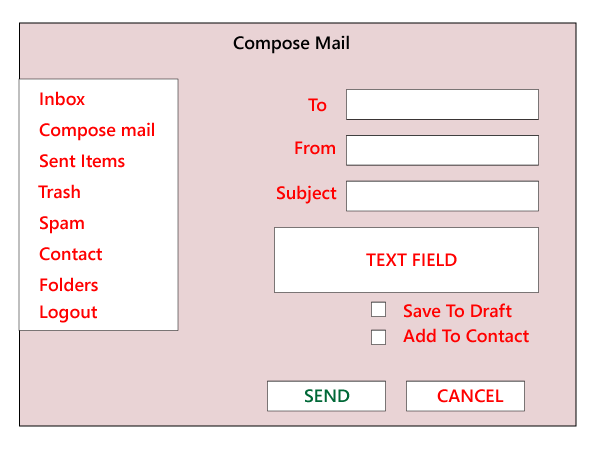
**For example**: In the Gmail application, the **Source** could be **Compose**, **Data** could be **Email** and the **Destination** could be the **Inbox**.

## Example of integration testing

Let us assume that we have a **Gmail** application where we perform the integration testing.

First, we will do **functional testing** on **the login page**, which includes the various components such as **username, password, submit, and cancel** button. Then only we can perform integration testing.

The different integration scenarios are as follows:



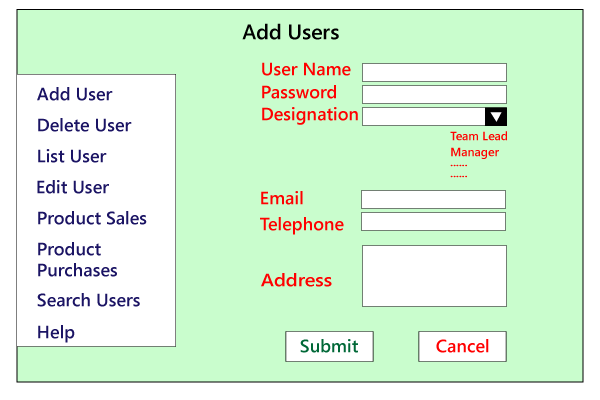
**Scenarios1:**

* First, we login as **P** users and click on the **Compose** mail and performing the functional testing for the specific components.
* Now we click on the **Send** and also check for **Save Drafts**.
* After that, we send a **mail** to **Q** and verify in the **Send Items** folder of P to check if the send mail is there.
* Now, we will **log out** as P and login as Q and move to the **Inbox** and verify that if the mail has reached.

**Secanrios2:** We also perform the integration testing on **Spam** folders. If the particular contact has been marked as spam, then any mail sent by that user should go to the spam folder and not in the inbox.

#### **Note: We will perform functional testing for all features, such as to send items, inbox, and so on.**

As we can see in the below image, we will perform the [**functional testing**](https://www.javatpoint.com/functional-testing) for all the **text fields and every feature**. Then we will perform **integration testing** for the related functions. We first test the **add user, list of users, delete user, edit user,** and then **search user**.



**Note:**

* There are some features, we might be performing only the **functional testing**, and there are some features where we are performing both **functional** and **integration testing** based on the feature's requirements.
* **Prioritizing is essential,** and we should perform it at all the phases, which means we will open the application and select which feature needs to be tested first. Then go to that feature and choose which component must be tested first. Go to those components and determine what values to be entered first.  
  And don't apply the same rule everywhere because testing logic varies from feature to feature.
* While performing testing, we should test one feature entirely and then only proceed to another function.
* Among the two features, we must be performing **only positive integrating testing** or both **positive and negative integration** testing, and this also depends on the features need.

## Reason Behind Integration Testing

Although all modules of software application already tested in unit testing, errors still exist due to the following reasons:

1. Each module is designed by individual software developer whose programming logic may differ from developers of other modules so; integration testing becomes essential to determine the working of software modules.
2. To check the interaction of software modules with the database whether it is an erroneous or not.
3. Requirements can be changed or enhanced at the time of module development. These new requirements may not be tested at the level of unit testing hence integration testing becomes mandatory.
4. Incompatibility between modules of software could create errors.
5. To test hardware's compatibility with software.
6. If exception handling is inadequate between modules, it can create bugs.

## Integration Testing Techniques

Any testing technique (Blackbox, Whitebox, and Greybox) can be used for Integration Testing; some are listed below:

### Black Box Testing

* State Transition technique
* Decision Table Technique
* Boundary Value Analysis
* All-pairs Testing
* Cause and Effect Graph
* Equivalence Partitioning
* Error Guessing

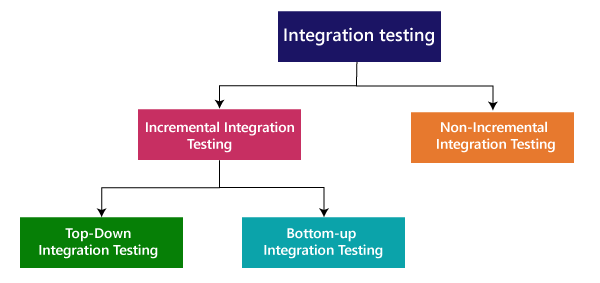
### White Box Testing

* Data flow testing
* Control Flow Testing
* Branch Coverage Testing
* Decision Coverage Testing

## Types of Integration Testing

Integration testing can be classified into two parts:

* **Incremental integration testing**
* **Non-incremental integration testing**

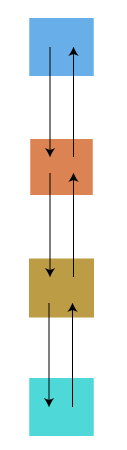


### Incremental Approach

In the Incremental Approach, modules are added in ascending order one by one or according to need. The selected modules must be logically related. Generally, two or more than two modules are added and tested to determine the correctness of functions. The process continues until the successful testing of all the modules.

**OR**

In this type of testing, there is a strong relationship between the dependent modules. Suppose we take two or more modules and verify that the data flow between them is working fine. If it is, then add more modules and test again.



**For example:** Suppose we have a Flipkart application, we will perform incremental integration testing, and the flow of the application would like this:

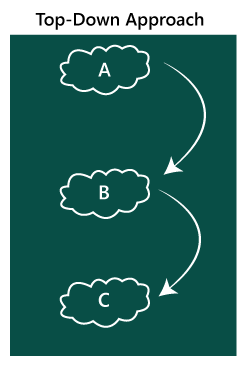
Flipkart→ Login→ Home → Search→ Add cart→Payment → Logout

Incremental integration testing is carried out by further methods:

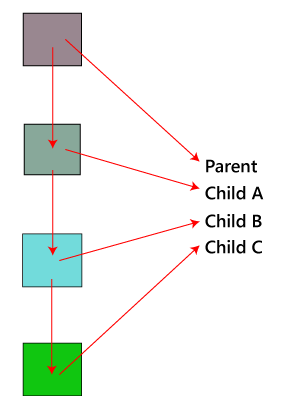
* Top-Down approach
* Bottom-Up approach

### Top-Down Approach

The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first. In this type of method, we will add the modules incrementally or one by one and check the data flow in the same order.



In the top-down approach, we will be ensuring that the module we are adding is the **child of the previous one like Child C is a child of Child B** and so on as we can see in the below image:



**Advantages:**

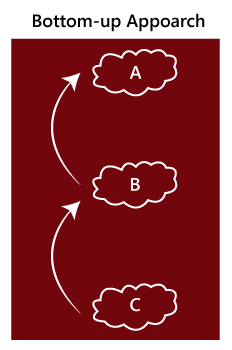
* Identification of defect is difficult.
* An early prototype is possible.

**Disadvantages:**

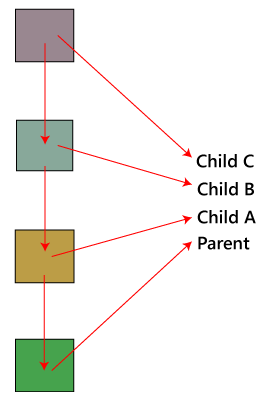
* Due to the high number of stubs, it gets quite complicated.
* Lower level modules are tested inadequately.
* Critical Modules are tested first so that fewer chances of defects.

### Bottom-Up Method

The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from **bottom to the top** and check the data flow in the same order.



In the bottom-up method, we will ensure that the modules we are adding **are the parent of the previous one** as we can see in the below image:



**Advantages**

* Identification of defect is easy.
* Do not need to wait for the development of all the modules as it saves time.

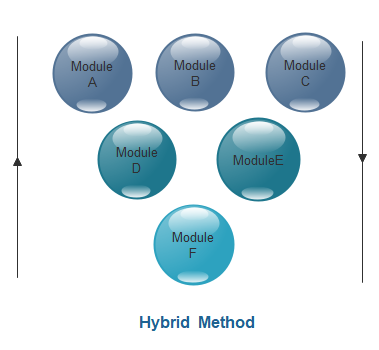
**Disadvantages**

* Critical modules are tested last due to which the defects can occur.
* There is no possibility of an early prototype.

In this, we have one addition approach which is known as **hybrid testing**.

### Hybrid Testing Method

In this approach, both **Top-Down** and **Bottom-Up** approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



**Advantages**

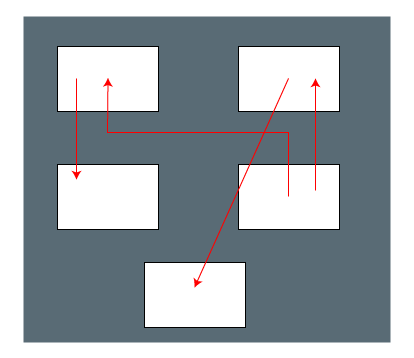
* The hybrid method provides features of both Bottom Up and Top Down methods.
* It is most time reducing method.
* It provides complete testing of all modules.

**Disadvantages**

* This method needs a higher level of concentration as the process carried out in both directions simultaneously.
* Complicated method.

### Non- incremental integration testing

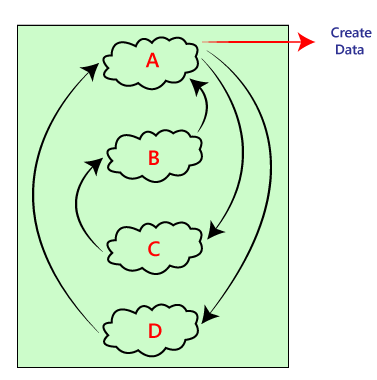
We will go for this method, when the data flow is very complex and when it is difficult to find who is a parent and who is a child. And in such case, we will create the data in any module bang on all other existing modules and check if the data is present. Hence, it is also known as the **Big bang method**.



### Big Bang Method

In this approach, testing is done via integration of all modules at once. It is convenient for small software systems, if used for large software systems identification of defects is difficult.

Since this testing can be done after completion of all modules due to that testing team has less time for execution of this process so that internally linked interfaces and high-risk critical modules can be missed easily.



**Advantages:**

* It is convenient for small size software systems.

**Disadvantages:**

* Identification of defects is difficult because finding the error where it came from is a problem, and we don't know the source of the bug.
* Small modules missed easily.
* Time provided for testing is very less.
* We may miss to test some of the interfaces.

Let us see examples for our better understanding of the non-incremental integrating testing or big bang method:

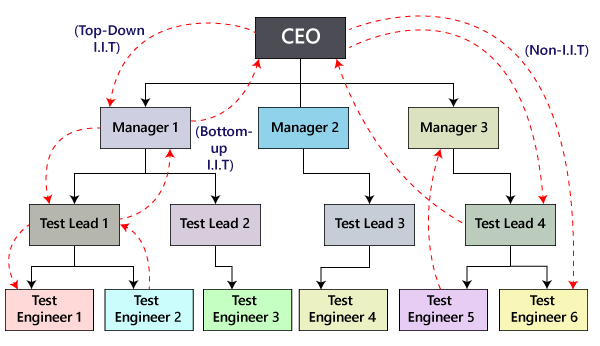
**Example1**

In the below example, the development team develops the application and sends it to the CEO of the testing team. Then the CEO will log in to the application and generate the username and password and send a mail to the manager. After that, the CEO will tell them to start testing the application.

Then the manager manages the username and the password and produces a username and password and sends it to the **test leads**. And the **test leads** will send it to the **test engineers** for further testing purposes. This order from the CEO to the test engineer is **top-down incremental integrating testing.**

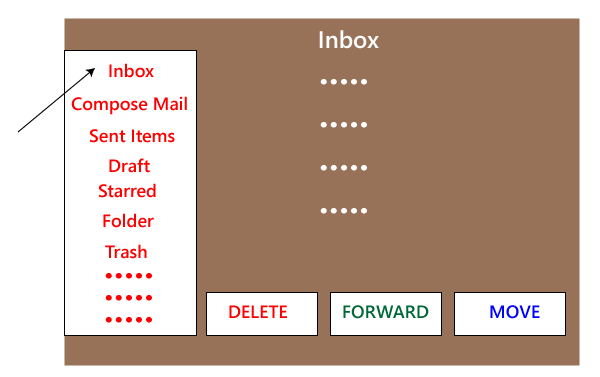
In the same way, when the test engineers are done with testing, they send a report to the **test leads**, who then submit a report to the **manager**, and the manager will send a report to the **CEO**. This process is known as **Bottom-up incremental integration testing** as we can see in the below image:

#### **Note: The combination incremental integration testing (I.I.T) and non-incremental integration testing is known as sandwich testing.**



**Example2**

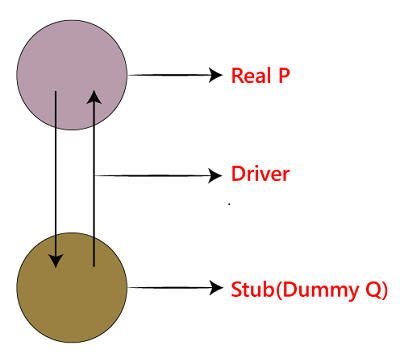
The below example demonstrates a home page of **Gmail's Inbox**, where we click on the **Inbox** link, and we are moved to the inbox page. Here we have to do **non- incremental integration testing** because there is no parent and child concept.



**Note**

**Stub and driver**

The **stub** is a dummy module that receives the data and creates lots of probable data, but it performs like a real module. When a data is sent from module P to Stub Q, it receives the data without confirming and validating it, and produce the estimated outcome for the given data.



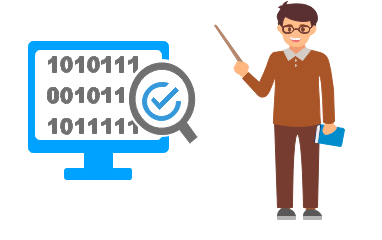
The function of a driver is used to verify the data from P and sends it to stub and also checks the expected data from the stub and sends it to P.

The **driver** is one that sets up the test environments and also takes care of the communication, evaluates results, and sends the reports. We never use the stub and driver in the testing process.

In [**White box testing**](https://www.javatpoint.com/white-box-testing)**, bottom-up integration testing** is ideal because writing drivers is accessible. And in [**black box testing**](https://www.javatpoint.com/black-box-testing), no preference is given to any testing as it depends on the application.

# Acceptance testing

Acceptance testing is formal testing based on user requirements and function processing. It determines whether the software is conforming specified requirements and user requirements or not. It is conducted as a kind of Black Box testing where the number of required users involved testing the acceptance level of the system. It is the fourth and last level of software testing.



User acceptance testing (UAT) is a type of testing, which is done by the customer before accepting the final product. Generally, UAT is done by the customer (domain expert) for their satisfaction, and check whether the application is working according to given business scenarios, real-time scenarios.

In this, we concentrate only on those features and scenarios which are regularly used by the customer or mostly user scenarios for the business or those scenarios which are used daily by the end-user or the customer.

However, the software has passed through three testing levels (Unit Testing, Integration Testing, System Testing) But still there are some minor errors which can be identified when the system is used by the end user in the actual scenario.

Acceptance testing is the squeezing of all the testing processes that have done previously.

**Note:**

It is done in the separate environment at the customer place, which is known as the UAT environment. The user acceptance testing is done by a different team called as domain expert who is known to the application.

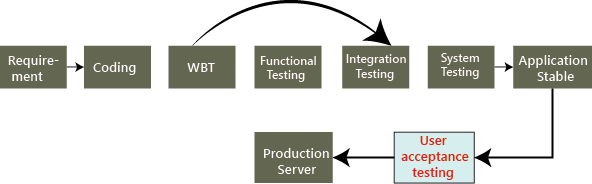
Generally, small companies do not have a domain expert because there is no frequent changes happen in the application.

## Reason behind Acceptance Testing

Once the software has undergone through Unit Testing, Integration Testing and System Testing so, Acceptance Testing may seem redundant, but it is required due to the following reasons.

* During the development of a project if there are changes in requirements and it may not be communicated effectively to the development team.
* Developers develop functions by examining the requirement document on their own understanding and may not understand the actual requirements of the client.
* There's maybe some minor errors which can be identified only when the system is used by the end user in the actual scenario so, to find out these minor errors, acceptance testing is essential.

#### **Note: Once we collect the requirement from the customer and done the coding process completely then the test engineer starts all different types of testing until the application becomes stable.**



Once the application is bug-free, we handover it to the customer, no customer accept the application blindly before using it. Hence, they do one round of testing for their satisfaction, which is known as user acceptance testing.

## Who performs user acceptance testing?

The acceptance testing can be performed by different persons in different cases.

For example, the blue-dart company gives the requirement to TCS for developing the application, and the TCS will accept the needs and agree to deliver the application in the two releases as we can see in the below image:

Acceptance Testing

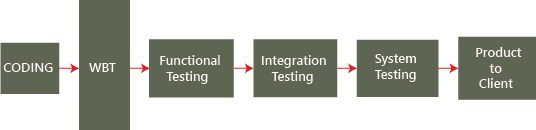
On August 10, the test manager tells the project manager that there is a critical bug in the application, and that will take another four days to fix it.

Acceptance Testing

But the project manager said we have to deliver the software within a given time. It takes another 30 days to fix the defect, or otherwise, we will have to pay the penalty (fine) for each day after the given release date. Is this the real situation? NO, let us see three different cases and understand who perform the acceptance testing.

**Case1**

In this, we will discuss how the acceptance testing is performed, and here the test engineer will do the acceptance testing.



Mostly, the actual flow for testing the application will be seen in the above image, but here it is little difference, as we know where the end-to-end testing or system testing ends and the acceptance testing will proceed. To understand this scenario, follow the below process:

The blue-dart provides the requirements, and TCS develops the application and performs all the testing and handover to the blue-dart company.

Now the question arises the blue-dart will use the application as soon they get it from TCS? NO, the blue dart company has a group of test engineers after they get the software, and this team will start testing the application, and this end-to-end testing is done at the customer environment, which is called the **User Acceptance Testing**.

Let us see the difference between **TCS test engineers** and **Blue-dart Engineers**:

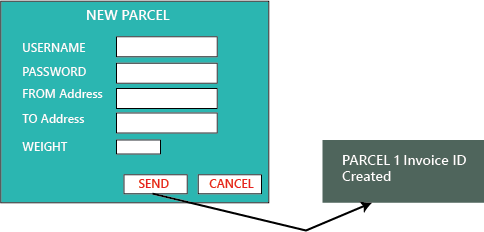
In **TCS**, the tester will perform the **functional testing, integration testing, and system testing**and whereas in **Blue-dart**, the tester will do only the **end-to-end or system testing, which is known as acceptance testing**.

The difference between end-to-end testing at TCS and blue-dart is as follows:

* The blue-dart test engineer is the one who gave the requirements
* The blue-dart engineer understands the product well
* The blue-dart engineer is a domain expert.
* They test the real-time data on the application.

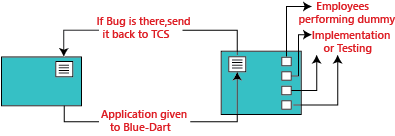
To understand this, we can see the below example, or if we have the application format is like this:

When the application is given to blue-dart test engineers, and they will perform testing and the application should generate a text message "**Parcel 1 invoice Id created**." It was not mentioned in the requirement, or it is there, and TCS does not fix it. Then fine(penalty) counts for TCS from that only, and whereas the test engineers at TCS will not knowing this, due to that, we can see the difference between the testing done at TCS and Blue-dart.



**Case2**

In this case, we will see how the Employee is becoming end-users and performing acceptance testing.



The application is developed and tested at the TCS environment and then sent to blue-dart. And in the Blue-dart, they have fewer test engineers, so they can't do acceptance testing. So for this, out of 300 employees of blue-dart, they will provide the application to the 30 employees and install the application to their systems and ask them to start using the application and find any defect or issues.

Now 30 employees will do the dummy implementation, which means they provide the data into the application and also written that data manually. And here, the employee becomes the end-user and also identify the bugs and issues while using the application.

These issues are verified against the requirements, and now the fine is charged for TCS (sometimes the penalty is charged on an hourly basis). If the identified bug is not as per requirement, then blue-dart can go for the **Request For Enhancement [REF] and Change Request [CR].**

Where **Request for enhancement** means that if the blue-dart feels that a particular module can be improved and developed in a better way, and then they can send the **Customer Requirement Specification [CRS]** as REF and TCS will follow the CRS and also make sure to do the necessary changes.

And the **Change Request** means, if the requirement has not been specified accurately, then blue-dart provides the exact needs and Request for changes.

Therefore, the acceptance testing can also be defined as end-to-end testing, which can be done by the engineers who are working in the client environment. Here, they take real-time scenarios and check whether the application is working fine or not, and also we can make real-time business scenarios because the end-user knows how the business flow works.

**Note:**

If we are getting more builds for acceptance testing, this means that:

* After receiving the application, the customer is getting more and more ideas, so they are asking for more and more changes.
* The quality of the software, which we delivered to customers, is not appropriate, and the development and testing both are not correctly done.
* The requirement which was given in the starting is not clear.

**Case3**

In this case, if the blue-dart customers become the end-users.

Here, the application is developed and tested and implemented at a blue-dart production server, and n-numbers of users start using the application, which is in the first release. While using the application, the blue-dart comes up with more number of features and enhancements, which is sent with the CRS to the TCS after that TCS will do the further changes in modules and sent it back to the blue-dart.

Hence, what is happing here, the application was developed when the requirement is collected by blue-dart from their end-users and customers.

The numbers of releases depend on the following facts:

* Difficulty of modules
* The number of modules.
* How the new module affects the old module.

**Note:**

**Hotfix:** In the production environment, whenever the customer identify the critical bug, we will do the following

* The developers fix the bugs.
* Small teams of test engineers will test the software.
* Re-install the application on the client environment.
* The client starts using the new software.

This entire process is known as a hotfix, and it can be done in a few hours or one day.

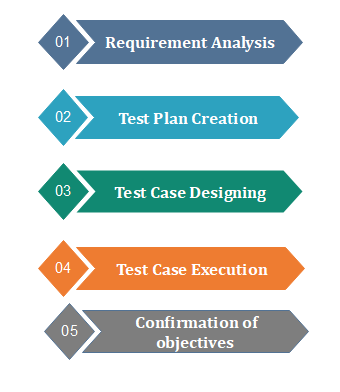
**For example:** If the significant module, suppose the Login module itself is not working at the production server, then the client will send it immediately for fixing it, and that has to be done as soon as possible.

**Short release**

Between two major releases, this is a short release of improvements, and it happens when the client needs some small features to change on an urgent basis.

**For example**, if we have 60 developers, where the ten developers will come out, and out of 40 test engineers, the 3 test engineers will come out, and they develop and test the application. And before adding it to the production server, the customer does one short round of acceptance testing.

## Steps to Perform Acceptance Testing



### Requirement Analysis:

In this step, the testing team analyzes requirement document to find out the objective of the developed software. Test planning accomplished by using requirement document, Process Flow Diagrams, System Requirements Specification, Business Use Cases, Business Requirements Document and Project Charter.

### Test Plan Creation:

Test Plan Creation outlines the whole strategy of the testing process. This strategy is used to ensure and verify whether the software is conforming specified requirements or not.

### Test Case Designing:

This step includes the creation of test cases based on test plan documents. Test cases should be designed in a way that can cover most of the acceptance testing scenario.

### Test Case Execution:

Test Case Execution includes execution of test cases by using appropriate input values. The testing team collects input values from the end user then all test cases are executed by both tester and end user to make sure software is working correctly in the actual scenario.

### Confirmation of objectives:

After successful completion of all testing processes, testing team confirms that the software application is bug-free and it can be delivered to the client.

## Tools used in Acceptance Testing

Acceptance Testing can be done by using several tools; some are given below:

done by using several tools; some are given below:

### Watir:

Acceptance testing uses this tool for the execution of automated browser-based test cases. It uses Ruby language for the inter-process communication.

### Fitness tool:

This tool is used to enter input values and generate test cases automatically. The user needs to input values, these values used by the tool to execute test cases and to produce output. It uses Java language for the inter-process communication. This tool makes it easy to create test cases as well as record them in the form of a table.

## Advantages of Acceptance Testing

* It increases the satisfaction of clients as they test application itself.
* The quality criteria of the software is defined in an early phase so that the tester has already decided the testing points. It gives a clear view to testing strategy.
* The information gathered through acceptance testing used by stakeholders to better understand the requirements of the targeted audience.
* It improves requirement definition as client tests requirement definition according to his needs.

## Disadvantages of Acceptance Testing

According to the testing plan, the customer has to write requirements in their own words and by themselves but

1. Customers are not willing to do that; it defeats the whole point of acceptance testing.
2. If test cases are written by someone else, the customer does not understand them, so tester has to perform the inspections by themselves only.

If the process is done in this manner, it destroys the existence of the Acceptance Testing.