K.T.S.P. MANDAL'S

**HUTUTMA RAJGURU MAHAVIDYALAYA,**

**RAJGURUNAGAR TAL-KHED, DIST-PUNE 410 505**

**DEPARTMENT OF COMPUTER SCIENCE**

**F.Y.Bsc (Computer Science)**

**Semester- II**

# Software Testing

**Subject – CS-362 -Software Testing**

**According to New Syllabus w.e.f. 2021-2022**

**Prof. Pallavi G. Darakhe**

**DEPARTMENT OF COMPUTER SCIENCE**

# Chapter 2

# Software Testing Strategies and Techniques

## Background

**1.Leave time for fixing:** Once problems are discovered, it is important to fix the time for the developers to resolve the issues. Also, the company needs time to retest the fixes as well.

**2. Discourage passing the buck:** If you want to minimize back and forth conversations between developers and testers, you need to develop a culture that will encourage them to hop on the phone or have desk-side chat to get to the bottom of things. Testing and fixing are all about collaboration.

**3. Manual testing has to be exploratory:** If it is possible to write down or script any issue in exact terms, it could be automated and belongs in the automated test suite. Real-world use of the software will not be scripted and the testers need to break things without a script.

**4. Encourage clarity:** You need to create a bug report that provides clarity rather than creating any confusion. However, it is also integral for a developer to go out of the way to effectively communicate as well.

**5. Test often:** This helps in preventing huge backlogs of problems from building up and crushing morale. Frequent testing is considered the best approach.

### Factors affecting Software Testing Strategies

- **Risks-** Risk management is very important during testing to figure out the risks and the risk level. For example, for an app that is well-established and slowly evolving, regression is a critical risk.

- **Objectives-** Testing should satisfy the requirements and needs of stakeholders to succeed. The objective is to look for as many defects as possible with less up-front time and effort invested.
- **Skills-** It is important to consider the skills of the testers since strategies should not only be chosen but executed as well. A standard-compliant strategy is a smart option when lacking skills and time in the team to create an approach.
- **Product-** Some products have specified requirements. This could lead to synergy with an analytical strategy that is requirements-based.
- **Business-** Business considerations and strategy are often important. If using a legacy system as a model for a new one, you could use a model-based strategy.
- **Regulations-** At some instances, one needs to satisfy the regulators along with the stakeholders. In this case, you would need a methodical strategy which satisfies these regulators.

## Testability

Software testability is measured with respect to the efficiency and effectiveness of testing. Efficient software architecture is very important for software testability. Software testing is a time-consuming, necessary activity in the software development lifecycle, and making this activity easier is one of the important tasks for software companies as it helps to reduce costs and increase the probability of finding bugs. There are certain metrics that could be used to measure testability in most of its aspects. Sometimes, testability is used to mean how adequately a particular set of tests will cover the product.

**Observability** is the ability to detect the software components' response to the inputs and monitor the alteration they cause to the inside state of the software. Careful observing is the basis for studying multiple behaviors and paths during testing.
**Controllability** is the capacity of a tester to control every module of software separately. The controllability level is directly proportional

to the testing efficiency. Moreover, control over software functionality is essential for performing any test automation.

## Best practices to follow to create a good test suite:

A good test suite doesn't take a long to execute. It ensures your software application works as intended. If it encounters a bug, it will automatically return feedback to help you identify the bug's source and help you fix it. Following are the properties which makes a test suite good fit to use for software developers:

**Fast:** If a test suite includes an extensive collection of integration tests and a few unit tests, it can take much longer to execute it, whereas a fast one will give feedback more quickly and make your development process even more efficient.

**Complete:** If your test suite covers 100% of your codebase, it will identify any errors arising from tweaks to your code application. Therefore, a complete test suite gives you confidence that your software applications are working fine as intended.

**Reliable:** It provides consistent feedback, irrespective of changes that can occur outside the test scope, whereas an unreliable one can have tests that fail intermittently, with no valuable feedback about changes you've done to your application.

**Isolated:** It runs test cases without hampering other tests in the suite. However, you may require cleaning up existing test data after running a test case in your suite.

**Maintainable:** A maintainable test suite that is organized is easy to manipulate. You easily add, change, or remove test cases. To maintain your test suite, you can follow best coding practices and develop a uniform process that suits you and your team.

**Expressive:** If your test suites are easy-to-read, they can be good for documentation. Always write test scripts that are descriptive of the features you are testing. Also, try to create a descriptive and understandable for a developer to read.

## Test Case Design for Desktop, Mobile, Web application using Excel

A **Test Case Template** is a well-designed document for developing and better understanding of the test case data for a particular test case scenario. A good Test Case template maintains test artifact consistency for the test team and makes it easy for all stakeholders to understand the test cases. Writing test case in a standard format lessen the test effort and the error rate. Test cases format are more desirable in case if you are reviewing test case from experts.

The template chosen for your project depends on your test policy. Many organizations create test cases in Microsoft Excel while some in Microsoft Word. Some even use test management tools like HP ALM to document their test cases.

| **How to write test cases: A step-by-step guide** |
|---|
| If I explain to you in just a two-line summary of how to write a manual test case, it would be: 1. Identify the feature or functionality you wish to test. 2. Create a list of test cases that define specific actions to validate the functionality. Let’s see the detailed steps for writing test cases. |
| Step 1 – Test Case ID: |
| In this step, the tester will assign a unique identifier to the test case. This allows the tester to recall and identify the test case in the future easily. |
| **Example:** TC-01: Verify Login Functionality for a User |
| Step 2 – Test Case Description: |
| The tester will describe the test case, outlining what it is designed to do. The tester may also provide a brief overview of the expected behavior. **An Example:** Test Case Description: Test for Logging Into the application Given: A valid username and password for the web |

| |
|---|
| application When: User enters the username and password in the login page Then: the user should be able to log in to the application successfully. The Home page for the application should be displayed. |
| Step 3 – Pre-Conditions: |
| The tester will document any pre-conditions that need to be in place for the test case to run properly. It may include initial configuration settings or manually executing some previous tests. A Pre-Condition example in testing could be that the test environment must be set up, to be very similar to the production environment, including the same hardware, operating system, and software. |
| Step 4 – Test Steps: |
| The tester will document the detailed steps necessary to execute the test case. This includes deciding which actions should be taken to perform the test and also possible data inputs. |
| **Example steps for our login test:** |
| 1. Launch the login application under test. |
| 2. Enter a valid username and password in the appropriate fields. |
| 3. Click the 'Login' button. |
| 4. Verify that the user has been successfully logged in. |
| 5. Log out and check if the user is logged out of the system. |
| Step 5 – Test Data: |
| The tester will define any necessary test data. For example, if the test case needs to test that login fails for incorrect credentials, then test data would be a set of incorrect usernames/passwords. |
| Step 6 – Expected Result: |
| The tester will provide the expected result of the test. This is the result the tester is looking to verify. **Examples of how to define expected results:** |
| 1. A user should be able to enter a valid username and password and click the login button. |
| 2. The application should authenticate the user's credentials and grant access to the application. |

| |
|---|
| 3. The invalid user should not be able to enter the valid username and password; click the login button. |
| 4. The application should reject the user’s credentials and display an appropriate error message. |
| Step 7 – Post Condition: |
| The tester will provide any cleanup that needs to be done after running the test case. This includes reverting settings or cleaning up files created during the test case. **Example:** 1. The user can successfully log in after providing valid credentials. 2. After providing invalid credentials, The user is shown the appropriate error message. 3. The user’s credentials are securely stored for future logins. 4. The user is taken to the correct page after successful login. 5. The user cannot access the page without logging in. 6. No unauthorized access to the user’s data. |
| Step 8 – Actual Result: |
| The tester will document the actual result of the test. This is the result the tester observed when running the test. **Example**: After entering the correct username and password, the user is successfully logged in and is presented with the welcome page. |
| Step 9 – Status: |
| The tester will report the status of the test. If the expected and actual results match, the test is said to have passed. If they do not match, the test is said to have failed. |
| **Example**: Tested the valid login functionality. Result: The user is able to log in with valid credentials. Overall Test Result: All the test steps were successfully executed, and the expected results were achieved. The login application is functioning as expected. Tested for Invalid Login functionality. Result: The user is unable to log in with invalid credentials. Overall Test Result: The invalid login functionality has been tested and verified to be working as expected |
| |

## **Manual Testing Test Case Examples**

Here are some examples that you can easily understand about Manual Testing:

1. Login Page: We can assume a login application like Gmail.

- **Test Case 1: Verify that the application allows users to input their username and password.**
- **Test Case 2: Verify that the application correctly validates the correct credentials.**
- **Test Case 3: Verify that the application displays an error message when the incorrect credentials are entered.**

2. Search Functionality:

We can assume Google searches for this.

- **Test Case 1: Verify that users can search for specific records in the database.**
- **Test Case 2: Verify that the application displays the query results correctly.**
- **Test Case 3: Verify that the application displays an error message when no matches are found.**

3. File Uploads:

We can assume a resume upload in any job portal like LinkedIn or Monster

- **Test Case 1: Verify that users are able to upload the correct type of file format.**
- **Test Case 2: Verify that the application does not allow users to upload malicious file formats.**
- **Test Case 3: Verify that the application displays an error message when the maximum file size is exceeded.**

The types of manual testing test cases are functional test cases, regression test cases, integration test cases, system test cases, GUI test

| |
|---|
| cases, security test cases, usability test cases, performance test cases, compatibility test cases, and acceptance test cases. |
| |

## Levels of Test writing process:

**Level 1:** In this level, you will write the **basic cases from the available specification** and user documentation.

**Level 2:** This is the **practical stage** in which writing cases depend on the actual functional and system flow of the application.
**Level 3:** This is the stage in which you will group some cases and **write a test procedure**. The test procedure is nothing but a group of small cases, maybe a maximum of 10.

**Level 4: Automation of the project.** This will minimize human interaction with the system and thus the QA can focus on the currently updated functionalities to test rather than remaining busy with Regression testing.

## Fields:

**1.Test case id**
**2.Unit to test:** What to be verified?
**3.Assumptions**
**4.Test data:** Variables and their values
**5.Steps to be executed**
**6.Expected Result**
**7.Actual result**
**8.Pass/Fail**
**9.Comments**

## Basic Format of Test Case Statement

***Verify***
***Using*** *[tool name, tag name, dialog, etc]*
***With*** *[conditions]*
***To*** *[what is returned, shown, demonstrated]*

**Verify:** Used as the first word of the test statement.
**Using:** To identify what is being tested. You can use 'entering' or 'selecting' here instead of using depending on the situation.

## For any application, you need to cover all types of tests as:

**Functional cases**
**Negative cases**
**Boundary value cases**
While writing these, all your **TC's should be simple and easy to understand**.

## Important Factors Involved in Writing Process:

### a) TCs are prone to regular revision and update:

We live in a continuously changing world and the same holds good for software as well. Software requirements change directly affects the cases. Whenever requirements are altered, TCs need to be updated.

Yet, it is not only the change in the requirement that may cause revision and update of TCs. During the execution of TCs, many ideas arise in the mind and many sub-conditions of a single TC may be identified. All this causes an update of TCs and sometimes it even leads to the addition of new TCs.

During regression testing, several fixes and/or ripples demand revised or new TCs.

### b) TCs are prone to distribution among the testers who will execute these:

Of course, there is hardly such a situation in which a single tester executes all the TCs. Normally, there are several testers who test different modules of a single application. So the TCs are divided among the testers according to their owned areas of the application under test.

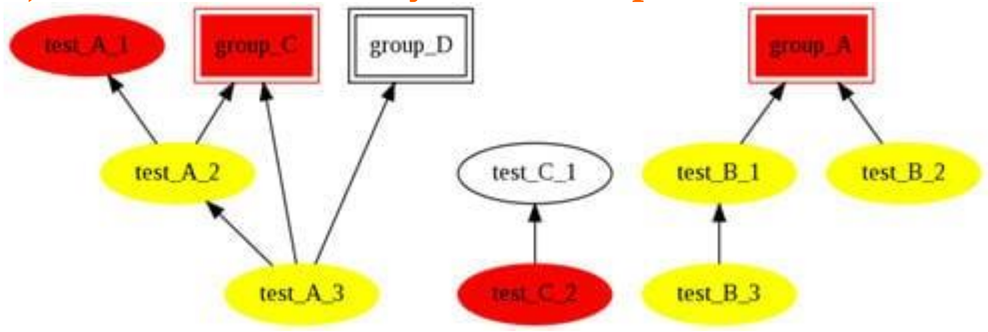
Some TCs which are related to the integration of application may be executed by multiple testers, while the other TCs may be executed only by a single tester.

## c) TCs are prone to Clustering and Batching:

It is normal and common that TCs belonging to a single test scenario usually demand their execution in some specific sequence or as a group. There may be certain pre-requisites of a TC that demand the execution of other TCs before running itself.

Similarly, as per the business logic of the AUT, a single TC may contribute to several test conditions and a single test condition may comprise multiple TCs.

## d) TCs have a tendency of inter-dependence:



This is also an interesting and important behavior of the TCs, denoting that they can be interdependent on each other. From medium to large applications with complex business logic, this tendency is more visible.

The clearest area of any application where this behavior can definitely be observed is the interoperability between different modules of the same or even different applications. Simply, wherever the different modules of a single application or multiple applications are interdependent, then the same behavior is reflected in the TCs as well.

## e) TCs are prone to distribution among the developers (especially in Test-driven development environment):

An important fact about TCs is that these are not only to be utilized by the testers. In the normal case, when a bug is under fix by the developers, they are indirectly using the TC to fix the issue.

Similarly, if the test-driven development is followed, then TCs are directly used by the developers in order to build their logic and cover all the scenarios in their code that are addressed by TCs.

## Tips to write effective Test Cases

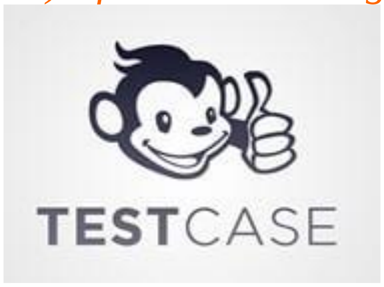
*#1) Keep it simple but not too simple; make it complex, but not too complex*

This statement seems a paradox. But, we promise it is not so. Keep all the steps of TCs atomic and precise. Mention the steps with the correct sequence and correct mapping to the expected results. The test case should be self-explanatory and easy to understand. This is what we mean to make it simple.

Now, making it complex means to make it integrated with the Test Plan and other TCs. Refer to the other TCs, relevant artifacts, GUIs, etc. where and when required. But, do this in a balanced way. Do not make a tester move back and forth in the pile of documents for completing a single test scenario.

Do not even let the tester to document these TCs compactly. While writing TCs, always remember that you or someone else will have to revise and update these.

*#2) After documenting the Test cases, review once as Tester*



Never think that the job is done once you have written the last TC of the test scenario. Go to the start and review all the TCs once, but not with the mindset of a TC writer or Testing Planner. Review all TCs with the mind of a tester. Think rationally and try to dry run your TCs.

Evaluate all the Steps and see if you have mentioned these clearly in an understandable way and the expected results are in harmony with those steps.

Ensure that the test data specified in TCs is feasible not only for actual testers but is according to the real-time environment too. Ensure that there is no dependency conflict among TCs and verify that all the references to other TCs/artifacts/GUIs are accurate. Otherwise, the Testers may be in great trouble.

### *#3) Bound as well as ease the Testers*

Do not leave the test data on testers. Give them a range of inputs especially where calculations are to be performed or the application's behavior depends on inputs. You can let them decide the test data item values but never give them the liberty to choose the test data items themselves.

Because, intentionally or unintentionally, they may use the same test data again & again and some important test data may be ignored during the execution of TCs.

Keep the testers at ease by organizing the TCs as per the testing categories and the related areas of an application. Clearly, instruct and mention which TCs are interdependent and/or batched. Likewise, explicitly indicate which TCs are independent and isolated so that the tester may manage his overall activity accordingly.

Now, you might be interested to read about boundary value analysis, which is a test case design strategy that is used in black-box testing. Click here to know more about it.

### *#4) Be a Contributor*

Never accept the FS or Design Document as it is. Your job is not just to go through the FS and identify the Test Scenarios. Being a QA resource, never hesitate to contribute to business and give suggestions if you feel that something can be improved in the application.

Suggest to developers too, especially in TC-driven development environment. Suggest the drop-down lists, calendar controls, selection-list, group radio buttons, more meaningful messages, cautions, prompts, improvements related to usability, etc.

Being a QA, don't just test but make a difference!

### *#5) Never Forget the End User*

The most important stakeholder is the 'End User' who will finally use the application. So, never forget him at any stage of TC's writing. In fact, the End User should not be ignored at any stage throughout the SDLC. Yet, our emphasis so far is just related to the topic.

So, during the identification of test scenarios, never overlook those cases which will be mostly used by the user or the cases which are business-critical even if they are less frequently used. Keep yourself in the shoes of the End User and then go through all the TCs and judge the practical value of executing all your documented TCs.

## Most Common Problems in Test Cases

1. Composite steps
2. Application behavior is taken as expected behavior

3. Multiple conditions in one case

These three have to be on my top 3 list of common problems in the test writing process.

What's interesting is that these happen with both new and experienced testers and we just keep following the same flawed processes without realizing that a few simple measures can fix things easily.

**Let's get to it and discuss each one:**

*#1) Composite Steps*

Firstly, what is a composite step?

For instance, you are giving directions from Point A to point B: if you say that "Go to XYZ place and then to ABC" this will not make sense, because here we ourselves think – "How do I get to XYZ in the first place"- instead of starting with "Turn left from here and go 1 mile, then turn right on Rd. no 11 to arrive at XYZ" might achieve better results.

The same rules apply to tests and their steps as well.

**For Example,** I am writing a test for Amazon.com – place an order for any product.

The following are my test steps (Note: We are only writing the steps and not all the other parts of the test like the expected result etc.)

**a**. Launch Amazon.com
**b**. Search for a product by entering the product keyword/name into the "Search" field on the top of the screen.
**c**. From the search results displayed, choose the first one.
**d**. Click on Add to Cart on the product details page.
**e**. Checkout and pay.
**f**. Check the order confirmation page.

Now, **can you identify which of these is a composite step?** Right- Step (e)

Remember, tests are always about “How” to test, so it is important to write the exact steps of “How to check out and pay” in your test.

**Therefore, the above case is more effective when written as below:**
**a**. Launch Amazon.com
**b**. Search for a product by entering the product keyword/name into the “Search” field on the top of the screen.
**c**. From the search results displayed, choose the first one.
**d**. Click on Add to Cart on the product details page.
**e**. Click on Checkout on the shopping cart page.
**f**. Enter the CC information, shipping, and billing information.
**g**. Click Checkout.
**h**. Check the order confirmation page.
Therefore, a composite step is one that can be broken down into several individual steps. Next time when we write tests, let’s all pay attention to this part and I am sure you will agree with me that we do this more often than we realize.

## *#2) Application behavior is taken as expected behavior*
More and more projects have to deal with this situation these days.

Lack of documentation, Extreme programming, rapid development cycles is few reasons that force us into relying on the application (an older version) to either write the tests or to base the testing itself on. As always, this is a proven bad practice- not always, really.

It is harmless as long as you keep an open mind and keep the expectation that the “AUT could be flawed”. It is only when you do not think that it is, things work badly. As always, we will let the examples do the talking.

**If the following is the page you are writing/designing the test steps for:**
**Case 1:**
**If my test case steps are as below:**

1. Launch the shopping site.

2. Click on Shipping and return- Expected result: The shipping and returns page is displayed with “Put your info here” and a “Continue” button.

Then, this is incorrect.

**Case 2:**

1. Launch the shopping site.
2. Click on Shipping and return.
3. In the ‘Enter the order no’ text box present on this screen, enter the order no.
4. Click Continue- Expected result: The details of the order related to shipping and returns are displayed.

Case 2 is a better test case because even though the reference application behaves incorrectly, we only take it as a guideline, do further research and write the expected behavior as per the expected correct functionality.

**Bottom line:** Application as a reference is a quick shortcut, but it comes with its own perils. As long as we are careful and critical, it produces amazing results.

*#3) Multiple Conditions in one case*

Once again, let’s learn from an **Example**.

**Look at the below test steps: The following are the test steps within one test for a login function.**

a. Enter valid details and click Submit.
b. Leave the Username field empty. Click Submit.
c. Leave the password field empty and click Submit.
d. Choose an already logged in username/password and click Submit.

What had to be 4 different cases is combined into one. You might think- What’s wrong with that? It is saving a lot of documentation and what I can do in 4; I am doing it in 1- isn’t that great? Well, not quite. Reasons?

**Read on:**

- What if one condition fails – we have to mark the entire test as ‘failed?’. If we mark the entire case ‘failed’, it means all 4 conditions are not working, which isn’t really true.
- Tests need to have a flow. From precondition to step 1 and throughout the steps. If I follow this case, in step (a), if it is successful, I will be logged onto the page, where the “login” option is no longer available. So when I get to step (b) – where is the tester going to enter the username? The flow is broken.

Document Collection for Test Writing

**#1) User Requirements Document**

It is a document that lists the business process, user profiles, user environment, interaction with other systems, replacement of existing systems, functional requirements, non-functional requirements, licensing and installation requirements, performance requirements, security requirements, usability, and concurrent requirements, etc.,

**#2) Business Use Case Document**

This document details the use case scenario of the functional requirements from the business perspective. This document covers the business actors (or system), goals, pre-conditions, post-conditions, basic flow, alternate flow, options, exceptions of each and every business flow of the system under requirements.

**#3) Functional Requirements Document**

This document details the functional requirements of each feature for the system under requirements.

Normally, functional requirements document serves as a common repository for both the development and testing team as well as to the project stakeholders including the customers for the committed (sometimes frozen) requirements, which should be treated as the most important document for any software development.

**#4) Software Project Plan (Optional)**

A document which describes the details of the project, objectives, priorities, milestones, activities, organization structure, strategy, progress monitoring, risk analysis, assumptions, dependencies, constraints, training requirements, client responsibilities, project schedule, etc.,

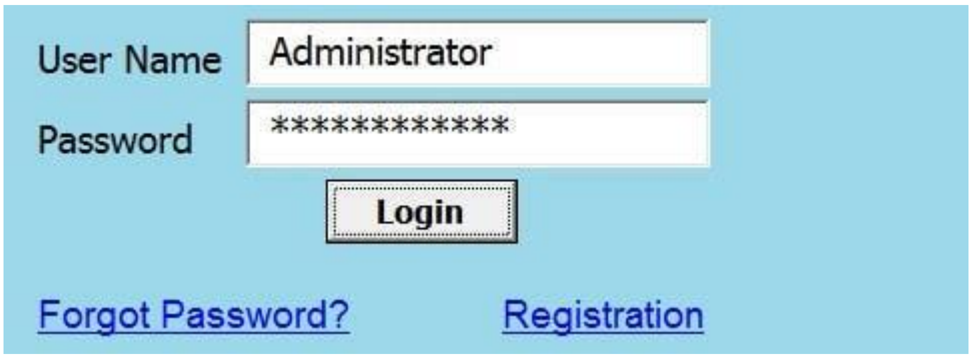
**#5) QA/Test Plan**

This document details the quality management system, documentation standards, change control mechanism, critical modules, and functionalities, configuration management system, testing plans, defect tracking, acceptance criteria, etc.

The test plan document is used to identify the features to be tested, features not to be tested, testing team allocations and their interface, resource requirements, testing schedule, test writing, test coverage, test deliverables, pre-requisite for test execution, bug reporting, and tracking mechanism, test metrics, etc.

Real Example

Let us see how to efficiently write test cases for a familiar ‘Login’ screen as per the below figure. The **approach of testing** will be almost the same even for complex screens with more information and critical features.



**#1)** The first approach for any test case process will be to get a screen prototype (or wire-frames) as above, if available. This may not be available for some functionalities and depends on the criticality of designing a prototype in the earlier stages of development.

But, if an SRS (Software Requirements Specification) document is available for the project, most of the screen prototypes are developed by the project team. This kind of screen simplifies the tester’s job and increases the efficiency of tests.

**#2)** Next, the **functional requirements specifications**. It depends on the organization process, it will be available in a suite of multiple documents.
So, decide the best document for writing cases, either it may be a user requirement document or a functional requirements specifications (or even an SRS document if it can be understandable comfortably by the testing team) which will give a complete functional flow of the selected feature to be tested.

**#3)** Once the screen prototype and functional specifications are in place, the tester should start writing the cases with the following approach and criteria.

- **UI Tests**: The controls/fields which are visible to the user. There are static control and dynamic controls available for the feature to be tested. **For Example,** in the login screen above, the 'User Name & Password' texts are static fields that require no user interaction, just for displaying the text only.
- **Functional Cases**: The login button and the Hyperlinks (Forgot Password? & Registration) are dynamic fields that require user interaction by clicking on the controls, which will do some action afterward.
- **Database Cases**: Once the user enters the username and password, the tests may be written to check the related database for, whether the username & password is checked in the right database & table, and also the user has the permission to log in to the application under test.
- **Process Tests**: This is related to the process (not the actions associated with the visible controls available on the screen) associated with the feature and the functionality. **For Example,** clicking Forgot Password link in the above sample screen may email the user. So, maybe an Email needs to be tested for the proper process and confirmation.

**4)** Finally, keep the "**BAOE mantra**", which means **i) Basic Flow ii) Alternate Flow iii) Options and iv) Exceptions** for the complete coverage of the functional flow and feature to be tested. Every concept should apply to positive and negative tests.

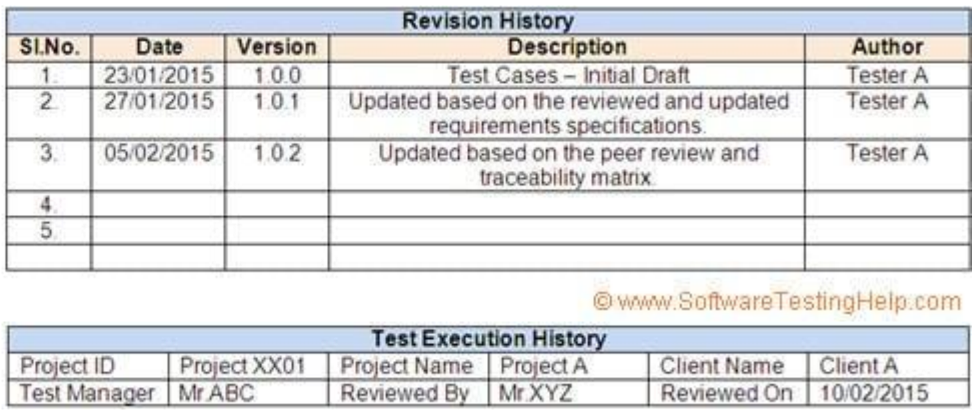
**For Example,** let us see the simple BAOE approach for the sample login screen above.

- **Basic Flow:** Enter the URL path of the Login in any browser and enter the information required and login to the application.
- **Alternate Flow:** Install the application on a mobile device and enter the information required and log in to the application.
- **Options:** What are the options that are available to come to the same login screen? **For Example,** after logging in to the application, clicking the 'Logout' may bring the same screen or if the session timeout or session expired, the user may come to the login screen.
- **Exceptions:** What are the exceptions if my tests are negative? **For Example,** if wrong credentials are entered in the Login screen, whether the user will get an error message or no action associated.

With all this information in hand, let us write the TCs for the login screen, in a format with complete coverage, traceability, and detailed information. The logical sequence and numbering of identifying the '**Test Case ID**' will be very useful for quick identification of the execution history of test cases.

**Also read** => 180+ sample ready to use test cases for web and desktop applications.

Test Case Document



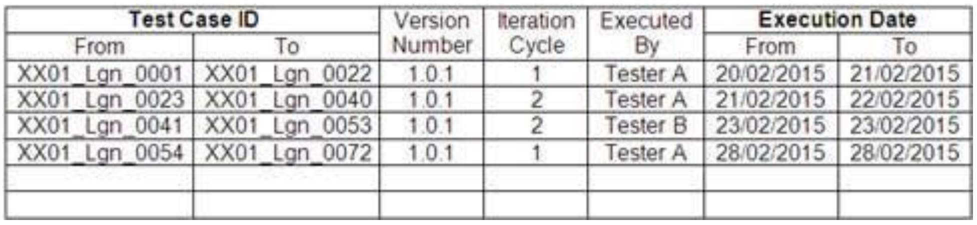
Revision History

| Sl.No. | Date | Version | Description | Author |
|---|---|---|---|---|
| 1 | 23/01/2015 | 1.0.0 | Test Cases – Initial Draft | Tester A |
| 2 | 27/01/2015 | 1.0.1 | Updated based on the reviewed and updated requirements specifications | Tester A |
| 3 | 05/02/2015 | 1.0.2 | Updated based on the peer review and traceability matrix | Tester A |
| 4 | | | | |
| 5 | | | | |
| | | | | |

© www.SoftwareTestingHelp.com

Test Execution History

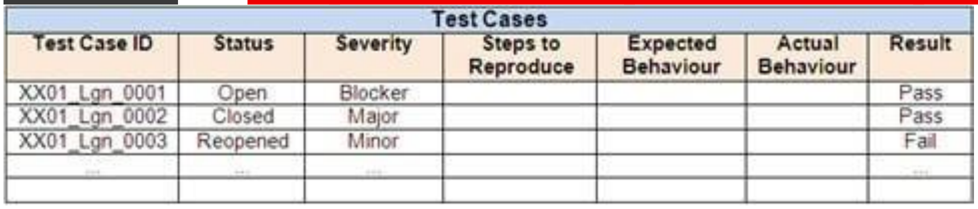
| | | | | | |
|---|---|---|---|---|---|
| Project ID | Project XX01 | Project Name | Project A | Client Name | Client A |
| Test Manager | Mr.ABC | Reviewed By | Mr.XYZ | Reviewed On | 10/02/2015 |



| Test Case ID | | Version Number | Iteration Cycle | Executed By | Execution Date | |
|---|---|---|---|---|---|---|
| From | To | | | | From | To |
| XX01_Lgn_0001 | XX01_Lgn_0022 | 1.0.1 | 1 | Tester A | 20/02/2015 | 21/02/2015 |
| XX01_Lgn_0023 | XX01_Lgn_0040 | 1.0.1 | 2 | Tester A | 21/02/2015 | 22/02/2015 |
| XX01_Lgn_0041 | XX01_Lgn_0053 | 1.0.1 | 2 | Tester B | 23/02/2015 | 23/02/2015 |
| XX01_Lgn_0054 | XX01_Lgn_0072 | 1.0.1 | 1 | Tester A | 28/02/2015 | 28/02/2015 |
| | | | | | | |
| | | | | | | |

**Note**: The test columns are not limited to the below sample test document, which can be maintained in an excel sheet to have as many columns as required for a complete traceability matrix viz., priority, purpose of testing, type of testing, error screenshot location, etc.

**Also read** => **Sample test case template with examples.**



| Test Cases | | | | | | |
|---|---|---|---|---|---|---|
| Test Case ID | Status | Severity | Steps to Reproduce | Expected Behaviour | Actual Behaviour | Result |
| XX01_Lgn_0001 | Open | Blocker | | | | Pass |
| XX01_Lgn_0002 | Closed | Major | | | | Pass |
| XX01_Lgn_0003 | Reopened | Minor | | | | Fail |
| ... | ... | ... | | | | ... |
| | | | | | | |

For the ease of simplicity and readability of this document, let us write the steps to reproduce, expected, and actual behavior of the tests for the login screen below

**Note**: Add the Actual Behavior column at the end of this template.
s

Test Data Collection
When the test case is being written, the most important task for any tester is to collect the test data. This activity is skipped and overlooked by many testers with the assumption that the test cases can be executed with some sample data or dummy data and can be fed when the data is really required.

This is a critical misconception that feeding sample data or input data from the mind memory at the time of executing test cases.

If the data is not collected and updated in the test document at the time of writing the tests, then the tester would spend abnormally more time collecting the data at the time of test execution. The test data should be collected for both positive and negative cases from all the perspectives of the functional flow of the feature. The business use case document is very much useful in this situation.

Find a sample test data document for the tests written above, which will be helpful in how effectively we can collect the data, which will ease our job at the time of test execution.

## White box Testing –

**White box testing** techniques analyze the internal structures the used data structures, internal design, code structure, and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing or structural testing. White Box Testing is also known as transparent testing or open box testing.

White box testing is a software testing technique that involves testing the internal structure and workings of a software application. The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level.

White box testing is also known as structural testing or code-based testing, and it is used to test the software's internal logic, flow, and structure. The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements.

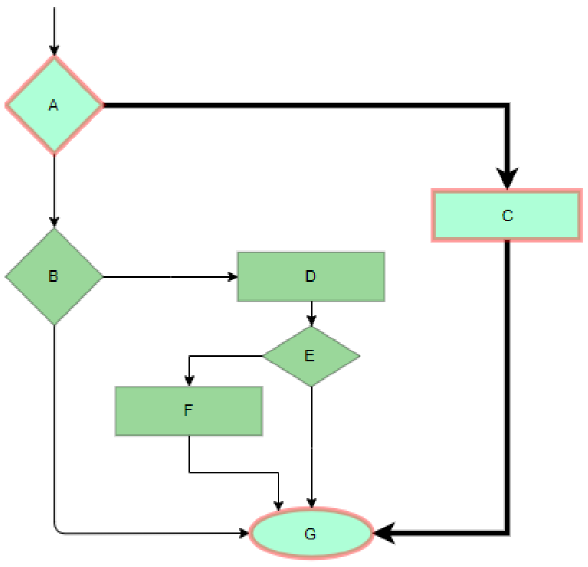
### Process of White Box Testing

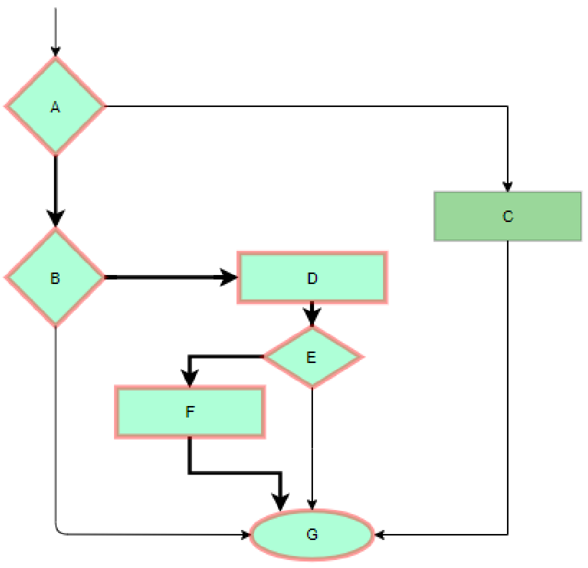
1. **Input:** Requirements, Functional specifications, design documents, source code.
2. **Processing:** Performing risk analysis to guide through the entire process.
3. **Proper test planning:** Designing test cases to cover the entire code. Execute rinse-repeat until error-free software is reached. Also, the results are communicated.
4. **Output:** Preparing final report of the entire testing process.

**Testing Techniques**

1. **Statement Coverage**

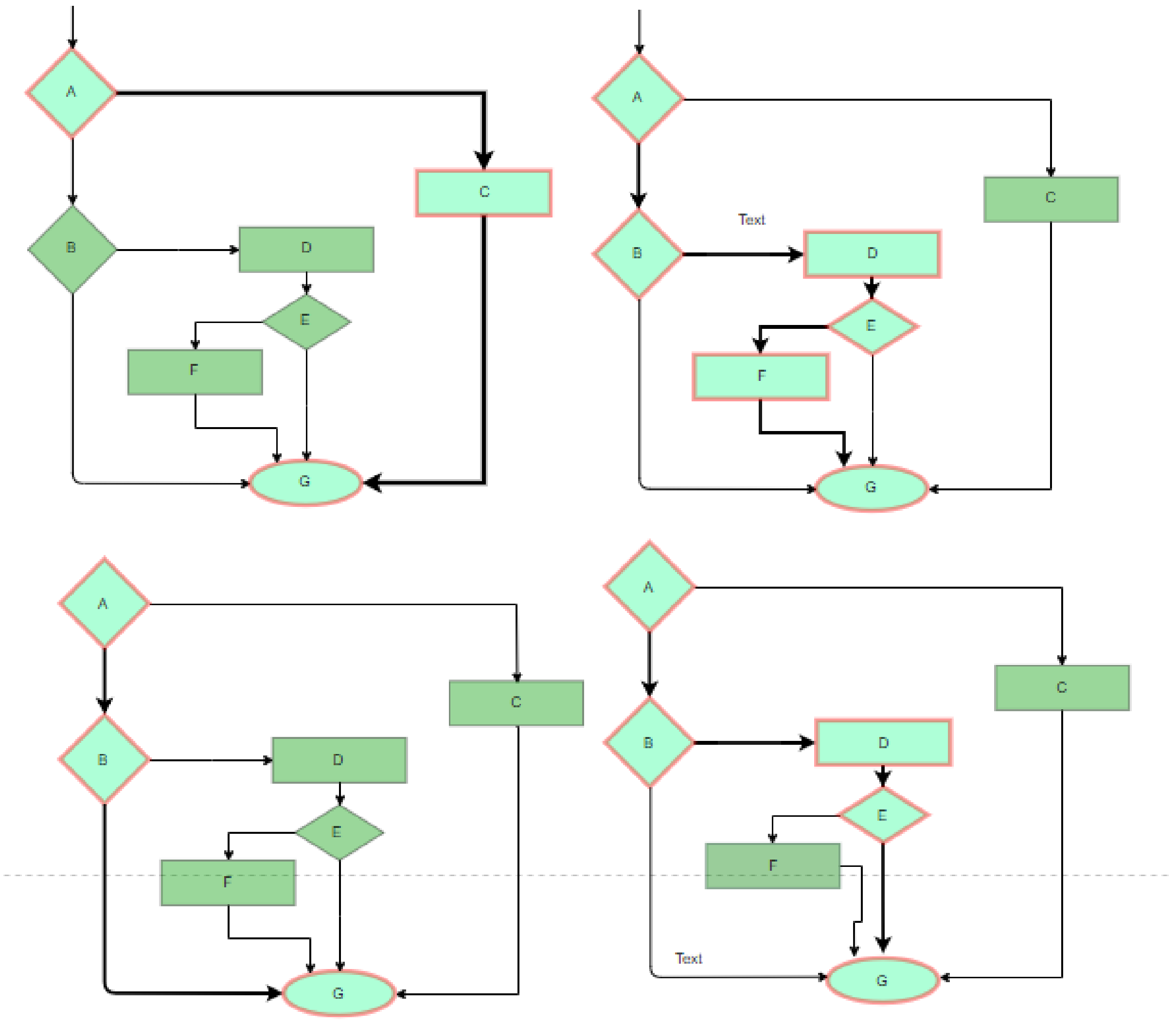
In this technique, the aim is to traverse all statements at least once. Hence, each line of code is tested. In the case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, it helps in pointing out faulty code.





*Statement Coverage Example*

2. **Branch Coverage:**
In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.



*4 test cases are required such that all branches of all decisions are covered,*

*i.e, all edges of the flowchart are covered*

3. Condition Coverage

In this technique, all individual conditions must be covered as shown in the following example:

- READ X, Y
- IF(X == 0 || Y == 0)
- PRINT ‘0’
- #TC1 – X = 0, Y = 55
- #TC2 – X = 5, Y = 0

4. Multiple Condition Coverage

In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let’s consider the following example:

- READ X, Y
- IF(X == 0 || Y == 0)
- PRINT ‘0’
- #TC1: X = 0, Y = 0
- #TC2: X = 0, Y = 5
- #TC3: X = 55, Y = 0
- #TC4: X = 55, Y = 5

5. Basis Path Testing

In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path. **Steps:**

- Make the corresponding control flow graph
- Calculate the cyclomatic complexity
- Find the independent paths
- Design test cases corresponding to each independent path
- V(G) = P + 1, where P is the number of predicate nodes in the flow graph
- V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes
- V(G) = Number of non-overlapping regions in the graph
- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

6. Loop Testing

Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

- **Simple loops:** For simple loops of size n, test cases are designed that:

1. Skip the loop entirely
2. Only one pass through the loop

3. 2 passes
4. m passes, where m < n
5. n-1 ans n+1 passes

- **Nested loops:** For nested loops, all the loops are set to their minimum count, and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.
- **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

**White Testing is performed in 2 Steps**

1. Tester should understand the code well
2. Tester should write some code for test cases and execute them

**Tools required for White box testing:**

- PyUnit
- Sqlmap
- Nmap
- Parasoft Jtest
- Nunit
- VeraUnit
- CppUnit
- Bugzilla
- Fiddler
- JSUnit.net
- OpenGrok
- Wireshark
- HP Fortify
- CSUnit

## Features of White box Testing

1. **Code coverage analysis:** White box testing helps to analyze the code coverage of an application, which helps to identify the areas of the code that are not being tested.
2. **Access to the source code:** White box testing requires access to the application's source code, which makes it possible to test individual functions, methods, and modules.

3. **Knowledge of programming languages:** Testers performing white box testing must have knowledge of programming languages like Java, C++, Python, and PHP to understand the code structure and write tests.
4. **Identifying logical errors:** White box testing helps to identify logical errors in the code, such as infinite loops or incorrect conditional statements.
5. **Integration testing:** White box testing is useful for integration testing, as it allows testers to verify that the different components of an application are working together as expected.
6. **Unit testing:** White box testing is also used for unit testing, which involves testing individual units of code to ensure that they are working correctly.
7. **Optimization of code:** White box testing can help to optimize the code by identifying any performance issues, redundant code, or other areas that can be improved.
8. **Security testing:** White box testing can also be used for security testing, as it allows testers to identify any vulnerabilities in the application's code.
9. **Verification of Design:** It verifies that the software's internal design is implemented in accordance with the designated design documents.
10. **C**
    **heck for Accurate Code:** It verifies that the code operates in accordance with the guidelines and specifications.
11. **I**
    **dentifying Coding Mistakes:** It finds and fix programming flaws in your code, including syntactic and logical errors.
12. **P**
    **ath Examination:** It ensures that each possible path of code execution is explored and test various iterations of the code.
13. **D**
    **etermining the Dead Code:** It finds and remove any code that isn't used when the programme is running normally (dead code).

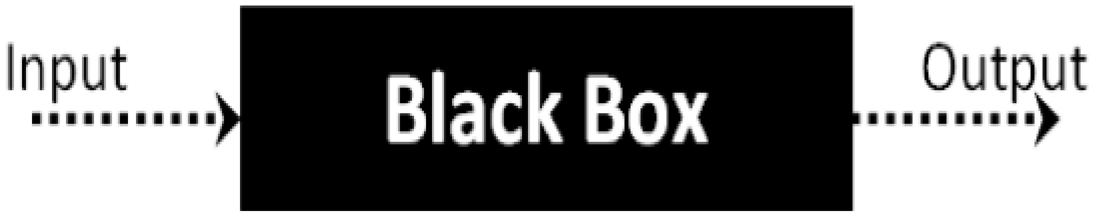
## Advantages of Whitebox Testing

1. **Thorough Testing**: White box testing is thorough as the entire code and structures are tested.
2. **Code Optimization:** It results in the optimization of code removing errors and helps in removing extra lines of code.
3. **Early Detection of Defects:** It can start at an earlier stage as it doesn't require any interface as in the case of black box testing.
4. **Integration with SDLC:** White box testing can be easily started in Software Development Life Cycle.
5. **Detection of Complex Defects:** Testers can identify defects that cannot be detected through other testing techniques.
6. **Comprehensive Test Cases:** Testers can create more comprehensive and effective test cases that cover all code paths.
7. Testers can ensure that the code meets coding standards and is optimized for performance.

## Disadvantages of White box Testing

1. **Programming Knowledge and Source Code Access:** Testers need to have programming knowledge and access to the source code to perform tests.
2. **Overemphasis on Internal Workings:** Testers may focus too much on the internal workings of the software and may miss external issues.
3. **Bias in Testing:** Testers may have a biased view of the software since they are familiar with its internal workings.
4. **Test Case Overhead:** Redesigning code and rewriting code needs test cases to be written again.
5. **Dependency on Tester Expertise:** Testers are required to have in-depth knowledge of the code and programming language as opposed to black-box testing.
6. **Inability to Detect Missing Functionalities:** Missing functionalities cannot be detected as the code that exists is tested.
7. **Increased Production Errors:** High chances of errors in production.

## Black Box Testing



**Features of Black Box Testing: The Basics**

At its core, Black Box Testing is a software testing method that scrutinizes the functionalities of software applications without any prior knowledge of their internal code structure, implementation details, or intricate internal pathways. Instead, it zeroes in on a simple principle – inputs and outputs. It’s like evaluating a vending machine; you don’t need to know how it dispenses snacks, just that it does so when you insert coins and make a selection.

**The features of black box testing are**

- **No Knowledge of Internal Code:**
  - Testers performing black box testing cannot access the software’s source code, internal algorithms, or implementation details. The focus is on understanding the system’s behavior based on inputs and outputs.
- **User’s Perspective**:
  - Black box testing simulates the perspective of an end-user or an external entity interacting with the software. It assesses how well the software meets user expectations and requirements.
- **Tests Multiple Inputs and Conditions:**
  - Test cases in black box testing cover a variety of inputs, including normal and boundary values, to evaluate how the software responds to different scenarios. It helps identify issues related to data handling and processing.
- **Test Design Independence:**

- Black box testing allows test design and execution to be carried out independently of the internal code. Testers can create test cases based on specifications, requirements, or functional specifications.

- **Focus on Outputs:**
  - The emphasis is on verifying the correctness of outputs generated by the software in response to given inputs. Testers check whether the actual outputs match the expected outputs.
- **System Integration Testing:**
  - Black box testing is commonly used for system integration testing, where the interactions between different components or subsystems are assessed without knowledge of their internal workings.
- **Test Case Reusability:**
  - Test cases developed for black box testing are often reusable, allowing for efficient testing of new releases or versions of the software. This helps in maintaining and improving test coverage over time.
- **Validation of Requirements:**
  - Black box testing helps validate that the software meets the specified requirements and adheres to the functional specifications. It ensures that the application behaves as intended by the stakeholders**.**
- **Non-Intrusive Testing:**
  - Black box testing is non-intrusive, meaning the testing process does not interfere with the system's internal logic. It evaluates the software's external behavior without making changes to its code.
- **Applicability at Different Testing Levels:**
  - Black box testing can be applied at various testing levels, including unit testing, integration testing, system testing, and acceptance testing. It provides a versatile approach for assessing different aspects of the software.

Black box testing is essential, complementing white and gray box testing methodologies to ensure comprehensive software quality assurance

## Boundary Value Analysis

Functional testing is a type of software testing in which the system is tested against the functional requirements of the system. It is conducted to ensure that the requirements are properly satisfied by the application. Functional testing verifies that each function of the software application works in conformance with the requirement and specification. Boundary Value Analysis(BVA) is one of the functional testings.

Boundary Value Analysis

Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.
It checks for the input values near the boundary that have a higher chance of error. Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition.

## Differences between Black Box Testing vs White Box Testing

| Black Box Testing | White Box Testing |
|---|---|
| It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it. | It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software. |
| Implementation of code is not needed for black box testing. | Code implementation is necessary for white box testing. |

| Black Box Testing | White Box Testing |
|---|---|
| It is mostly done by software testers. | It is mostly done by software developers. |
| No knowledge of implementation is needed. | Knowledge of implementation is required. |
| It can be referred to as outer or external software testing. | It is the inner or the internal software testing. |
| It is a functional test of the software. | It is a structural test of the software. |
| This testing can be initiated based on the requirement specifications document. | This type of testing of software is started after a detail design document. |
| No knowledge of programming is required. | It is mandatory to have knowledge of programming. |
| It is the behavior testing of the software. | It is the logic testing of the software. |
| It is applicable to the higher levels of testing of software. | It is generally applicable to the lower levels of software testing. |
| It is also called closed testing. | It is also called as clear box testing. |
| It is least time consuming. | It is most time consuming. |
| It is not suitable or preferred for algorithm testing. | It is suitable for algorithm testing. |
| Can be done by trial and error ways and methods. | Data domains along with inner or internal boundaries can be better tested. |
| **Example:** Search something on | **Example:** By input to check and |

| **Black Box Testing** | **White Box Testing** |
|---|---|
| google by using keywords | verify loops |
| **Black-box test design techniques-**<br>• Decision table testing<br>• All-pairs testing<br>• Equivalence partitioning<br>• Error guessing | **White-box test design techniques-**<br>• Control flow testing<br>• Data flow testing<br>• Branch testing |
| **Types of Black Box Testing:**<br>• Functional Testing<br>• Non-functional testing<br>• Regression Testing | **Types of White Box Testing:**<br>• Path Testing<br>• Loop Testing<br>• Condition testing |
| It is less exhaustive as compared to white box testing. | It is comparatively more exhaustive than black box testing. |