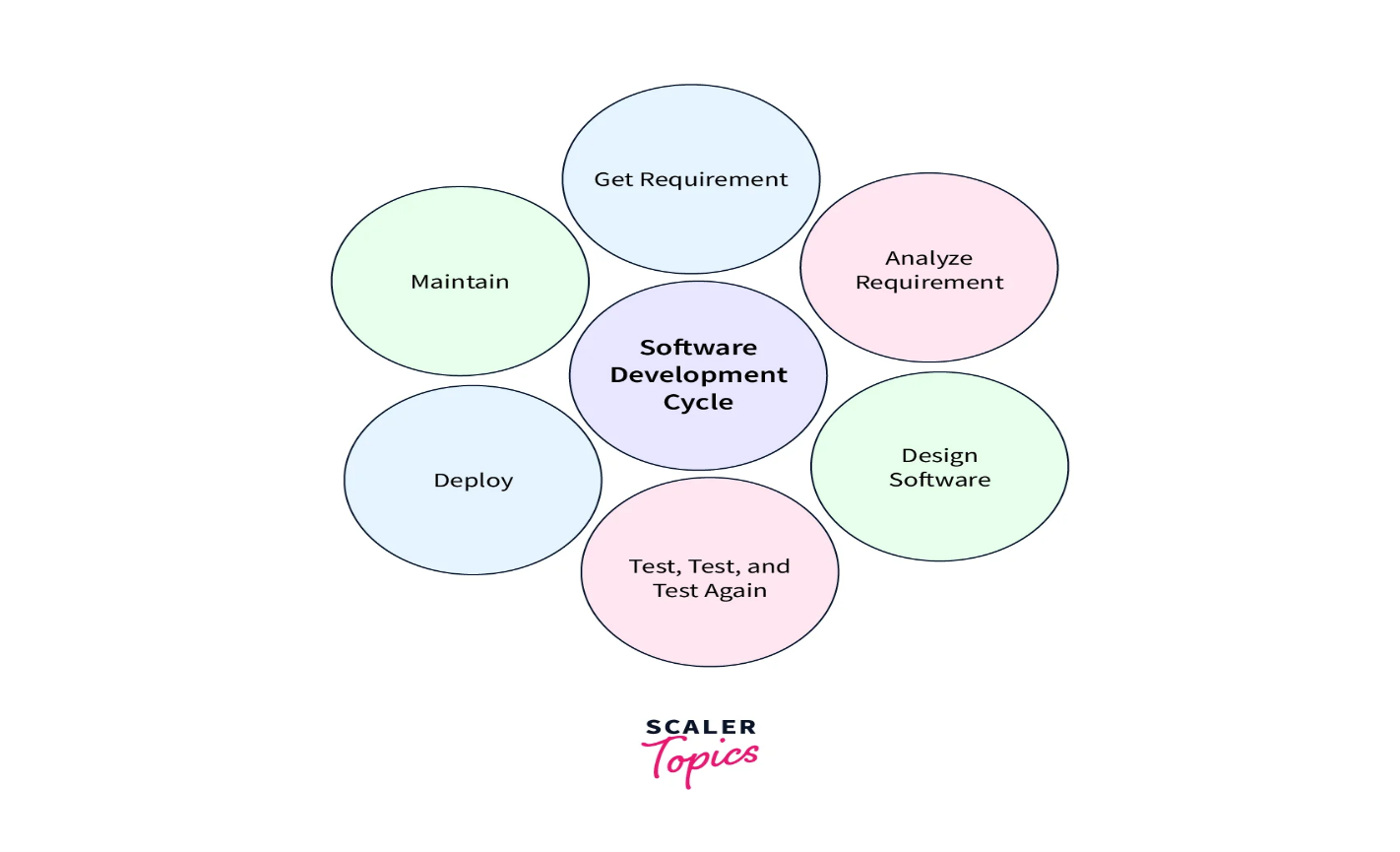
Module 3

Software Engineering

**Fundamental of testing**

Software testing is the process of determining software's accuracy by taking into account all of its properties (reliability, scalability, portability, re-usability, and usability) and analysing the execution of software components to uncover software bugs, mistakes, or flaws. The fundamentals of software testing entail examining code as well as executing code in multiple contexts and situations. In today's software development scenario, a testing team may be independent of the development team. Software testing aims to find mistakes, gaps, or missing requirements compared to the actual needs.



**Steps of Software Testing?**

Verification and Validation are two major steps in software testing. These steps implement the fundamentals of software testing.

* In software testing, **verification** is the process through which your team checks whether the program, system, or framework is consistent and aligned with the documentation requirements.
* **Validation** is the procedure by which your team verifies the system's accuracy. During this process, you will reflect on the product and the system; and think of what users desire and what has been developed.

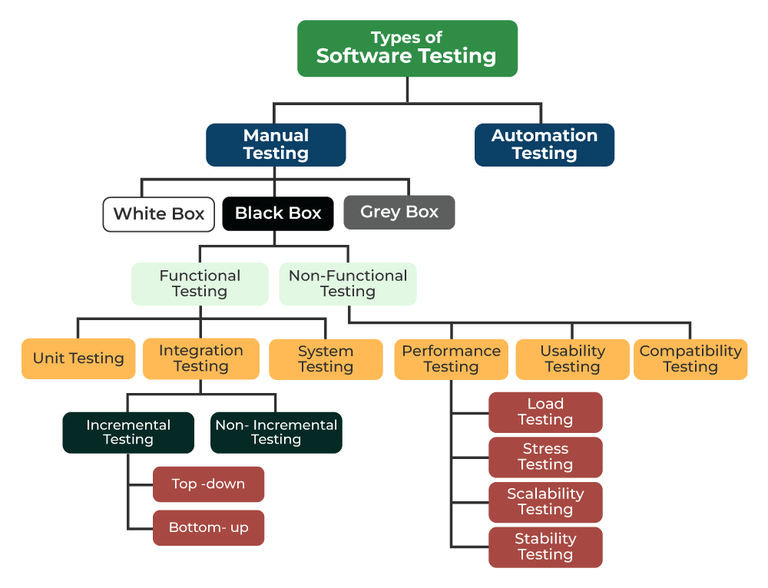
**Importance of Software Testing**

* **Defects can be identified early:**Software testing is important because if there are any bugs they can be identified early and can be fixed before the delivery of the software.
* **Improves quality of software:**Software Testing uncovers the defects in the software, and fixing them improves the quality of the software.
* **Increased customer satisfaction:**Software testing ensures reliability, security, and high performance which results in saving time, costs, and customer satisfaction.
* **Helps with scalability:**Software testing type non-functional testing helps to identify the scalability issues and the point where an application might stop working.
* **Saves time and money:**After the application is launched it will be very difficult to trace and resolve the issues, as performing this activity will incur more costs and time. Thus, it is better to conduct software testing at regular intervals during software development.

**Need for Software Testing**

Software bugs can cause potential monetary and human loss. There are many examples in history that clearly depicts that without the testing phase in software development lot of damage was incurred. Below are some examples:

* **1985:**Canada’s Therac-25 radiation therapy malfunctioned due to a software bug and resulted in lethal radiation doses to patients leaving 3 injured and 3 people dead.
* **1994:**China Airlines Airbus A300 crashed due to a software bug killing 264 people.
* **1996:**A software bug caused U.S. bank accounts of 823 customers to be credited with 920 million US dollars.
* **1999:**A software bug caused the failure of a $1.2 billion military satellite launch.
* **2015:**A software bug in fighter plan F-35 resulted in making it unable to detect targets correctly.
* **2015:**Bloomberg terminal in London crashed due to a software bug affecting 300,000 traders on the financial market and forcing the government to postpone the 3bn pound debt sale.
* Starbucks was forced to close more than 60% of its outlet in the U.S. and Canada due to a software failure in its POS system.
* Nissan cars were forced to recall 1 million cars from the market due to a software failure in the car’s airbag sensory detectors.

****

**Different Types of Software Testing**

**Automated Testing**

Automated Testing refers to the use of specialized software tools to execute predefined tests on the software application automatically. Unlike manual testing, which requires human intervention to perform the tests, automated testing runs the test cases using scripts or testing tools without manual input.

Advantages of Automated Testing:

* Speed: Automated tests run faster than manual tests and can be executed repeatedly without additional effort.
* Accuracy: Reduces the risk of human error, ensuring consistent and precise test execution.
* Reusability: Test scripts can be reused for different versions of the software, making it efficient for regression testing.
* Scalability: Automated testing can handle large test suites and run tests simultaneously on different platforms, environments, and configurations.
* Cost-effectiveness in the Long Term: Though the initial setup cost for automation may be high, it can save costs in the long run by reducing manual labor.

Disadvantages of Automated Testing:

1. High Initial Cost:
   * Setting up an automated testing environment requires investment in tools, infrastructure, and training. Writing and maintaining test scripts can be costly, especially for smaller projects with limited budgets.
2. Complex Setup and Maintenance:
   * Creating automated tests often requires programming skills and expertise. Additionally, as the software evolves, test scripts need constant updates to remain effective, which can be time-consuming and complex.
3. Limited Flexibility:
   * Automated tests are designed to perform specific tasks and may not adapt well to new or unforeseen changes in the software. Exploratory testing, where testers actively try to discover unexpected behavior, is difficult to automate.
4. Not Suitable for All Types of Testing:
   * Automated testing is not ideal for tests that require subjective evaluation, such as user experience, visual elements, or usability. Manual testing is better suited for such scenarios.
5. False Positives/Negatives:
   * Automated tests can sometimes produce incorrect results, such as marking a passing test as failed (false negative) or vice versa (false positive). This can lead to wasted time investigating non-issues or missing actual defects.
6. Initial Time Investment:
   * The time required to write and configure automated tests can be significant, particularly for large and complex applications. This investment may delay the immediate testing process, though it can save time in the long run.
7. Over-reliance on Automation:
   * Over-relying on automated testing can lead to neglecting manual testing efforts, potentially missing critical bugs that are best identified through human intuition and exploration.

Why Use Automated Testing Despite the Disadvantages:

1. Efficiency in Repetitive Testing:
   * Automated testing excels in scenarios where repetitive and time-consuming tests need to be executed multiple times, such as regression testing, which checks that new changes don’t break existing functionality.
2. Consistency and Reliability:
   * Automated tests provide consistent results, reducing the variability that comes with manual testing. This consistency is crucial for ensuring that tests produce reliable results across different environments and iterations.
3. Scalability for Large Projects:
   * As projects grow, the need for testing also increases. Automated testing can handle large test suites, run tests in parallel, and cover a wide range of scenarios, making it scalable for large and complex projects.
4. Faster Feedback Loop:
   * Automated tests provide quick feedback on code changes, enabling developers to identify and fix issues early in the development process, which is vital for continuous integration and continuous deployment (CI/CD) practices.
5. Long-Term Cost Savings:
   * While the initial cost and setup time for automated testing are high, it can save money in the long run by reducing the need for repetitive manual testing and catching defects earlier in the development cycle.
6. Support for Agile and DevOps Practices:
   * Automated testing is essential for Agile and DevOps environments, where rapid development cycles and frequent releases require continuous testing and integration. Automated tests help ensure that quality is maintained throughout the development process.

**Manual Testing**

Manual Testing is the process of manually executing test cases without the use of automated tools. The tester interacts with the software to identify defects or bugs by comparing the actual behavior of the application with the expected behavior.

Advantages of Manual Testing:

* Exploratory Testing: Allows testers to explore the application and find defects that automated tests might not catch.
* User-Centric: Testers can provide a better evaluation of the user experience and usability.
* Flexibility: Testers can adapt and change test cases on the fly, which is valuable in dynamic and rapidly changing environments.

Disadvantages of Manual Testing:

* Time-Consuming: Manual testing requires more time, especially for large and complex systems.
* Human Error: There is a higher risk of missing defects due to human error.
* Inconsistent: Results may vary depending on the tester's skills and knowledge.

**White Box Testing**

White Box Testing, also known as Clear Box Testing or Glass Box Testing, involves testing the internal structure, design, and coding of the software. Testers need to have knowledge of the code to design and execute test cases that validate the logic, code paths, and functions of the software.

Characteristics of White Box Testing:

* Code Coverage: Ensures that all paths, loops, and conditions in the code are tested.
* Focus on Internal Workings: Tests the logic and functionality of the code, rather than the user interface.
* Development-Oriented: Typically performed by developers or testers with coding knowledge.

Examples of White Box Testing:

* Unit Testing: Testing individual components or functions of the code.
* Integration Testing: Testing the interaction between different modules or components.

**Grey Box Testing**

Grey Box Testing is a hybrid approach that combines both White Box and Black Box Testing techniques. In Grey Box Testing, testers have partial knowledge of the internal workings of the application, which allows them to design more effective test cases.

Characteristics of Grey Box Testing:

* Limited Knowledge: Testers have some understanding of the code and architecture but do not need full access to the source code.
* Combination of Approaches: Leverages both internal code knowledge and external functional testing.
* Useful for Complex Systems: Effective for testing complex systems where both functionality and internal structure are important.

Examples of Grey Box Testing:

* Testing a web application by knowing the database schema to validate data integrity while also testing the user interface.

**Black Box Testing**

Black Box Testing focuses on testing the software's functionality without any knowledge of the internal code or structure. The tester interacts with the application as an end user would, providing inputs and validating outputs based on expected behavior.

Characteristics of Black Box Testing:

* Focus on Functionality: The goal is to validate that the software meets the specified requirements and behaves as expected.
* No Knowledge of Internal Code: Testers do not need to understand the code or architecture.
* User-Centric**:** Tests are designed from the user's perspective.

**Types of Black Box Testing (As per the given image):**

1. **Functional Testing:**
   * **Unit Testing:**
     + Tests individual components or units of the software in isolation to ensure they function correctly. In Black Box Testing, the focus is on inputs and expected outputs rather than the code itself.
   * **Integration Testing:**
     + Tests the interaction between integrated units or modules. The goal is to ensure that the integrated components work together as expected.
     + **Incremental Testing:**
       - Involves testing modules one at a time as they are integrated.
       - **Top-Down:** Begins testing from the top-level modules and integrates lower-level modules step by step.
       - **Bottom-Up:** Starts with lower-level modules and integrates higher-level modules step by step.
     + **Non-Incremental Testing:**
       - Involves testing all modules at once without incremental integration, often referred to as "Big Bang" testing.
2. **System Testing:**
   * Tests the complete and integrated system to evaluate if it meets the specified requirements. It tests the system as a whole, covering functional and non-functional requirements.
3. **Non-Functional Testing:**
   * **Performance Testing:**
     + Evaluates the system's performance under different workloads.
     + Load Testing: Tests how the system handles a large number of users or high volumes of data.
     + Stress Testing: Pushes the system beyond its normal operational limits to identify breaking points.
     + Scalability Testing: Ensures the system can scale up efficiently as demand increases.
     + Stability Testing: Tests the system's stability over time under different conditions.
   * **Usability Testing:**
     + Evaluates the ease of use and user-friendliness of the application. It focuses on the user experience, interface design, and overall interaction.
   * **Compatibility Testing:**
     + Ensures that the application works across different devices, browsers, operating systems, and platforms. Compatibility testing is crucial for applications that need to function on a wide range of environments.

**White box testing**

The developer performs the white box testing and checks every line of code before passing it on to the test engineer. Because the code is accessible to the developer during testing, it is also known as white box testing. The term **"white box" refers to the system's internal perspective**, i.e., the ability to see through the exterior shell of the software into its inner workings. White box testing focuses on the inner intricacies of an application and is centered on internal structure testing. Programming abilities are required to design test cases in this form of testing.

**Types of White Box Testing Techniques:**

1. **Unit Testing:**
   * Focuses on testing individual components or functions within the software.
   * Ensures that each function or class behaves as expected in isolation.
2. **Integration Testing:**
   * Examines how different modules or components interact with each other.
   * Verifies that integrated components work correctly when combined.
3. **Statement Coverage:**
   * Ensures every possible statement in the code is executed at least once.
4. **Branch Coverage:**
   * Ensures that all possible branches in the code (e.g., if-else statements) are tested.
5. **Path Coverage:**
   * Ensures that all possible execution paths in the code are tested.
6. **Control Flow Testing:**
   * Focuses on the logical flow of the program, ensuring all paths and conditions are covered.
7. **Data Flow Testing:**
   * Tracks the flow of data through the code, ensuring that variables are properly initialized, used, and cleaned up.
8. **Mutation Testing:**
   * Deliberately introduces small changes (mutations) into the code to test whether the test cases can detect errors.

**White Box Testing Process:**

1. **Understand the Code:** The first step is to understand the software’s internal structure and logic. This includes studying the source code, control flow, and data flow.
2. **Identify Test Cases:** Based on the code structure, identify specific test cases that need to be written. This might include testing specific branches, paths, or conditions in the code.
3. **Write and Execute Tests:** Write the test scripts or manually test the identified cases. These tests should be run on the software to validate the internal logic.
4. **Analyze Code Coverage:** After running the tests, analyze which parts of the code were executed and identify any gaps. Tools like coverage analyzers can help visualize this.
5. **Report Results:** Document the results of the testing, including any bugs or issues found, and the percentage of the code covered by the tests.
6. **Iterate:** Based on the test results, refine the test cases, fix any bugs, and continue testing until the software meets the quality criteria.

**Benefits of White Box Testing:**

1. **Thoroughness:** Since every part of the code is tested, it ensures a high level of quality and reliability.
2. **Early Bug Detection:** Bugs are often found early in the development process, reducing the cost and effort needed to fix them later.
3. **Optimized Code:** White box testing can help identify unnecessary or inefficient code, leading to better performance.
4. **Security:** By analyzing the internal code, potential security vulnerabilities can be detected and addressed.

**Challenges of White Box Testing:**

1. **Complexity:** Requires deep knowledge of the code, which can be challenging for large or complex systems.
2. **Time-Consuming:** Writing and maintaining test cases for every possible path in the code can be time-consuming.
3. **Limited to Known Issues:** Since white box testing relies on understanding the code, it may not detect issues that arise from unexpected inputs or conditions.

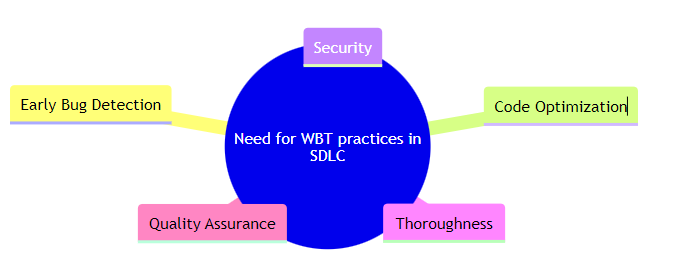
**Objective of White Box Testing**

White Box Testing serves a crucial role in software testing by allowing testers to inspect and verify the inner workings of a software system, including its code, infrastructure, and integrations.

**The key objective of White Box Testing include**:

* **Thoroughness**: It provides complete [code coverage](https://www.browserstack.com/guide/code-coverage-vs-test-coverage), ensuring every part of the software’s internal structure is tested.
* **Automation**: Test cases can be easily automated, saving time and resources.
* **Optimization**: It helps in code optimization by identifying hidden errors and redundancies.
* **Introspection**: It provides an in-depth understanding of the software, which can be invaluable for future development and maintenance.

**Why perform White Box Testing?**

  
White Box Testing practices are integral to the Software Development Life Cycle (SDLC) for several reasons:

* **Early Bug Detection:** White Box Testing allows for detecting bugs and errors early in development. This early detection can save time, effort, and resources, as fixing bugs later in the development process can be more complex and costly.
* **Code Optimization:** Identify redundant code and software areas that can be optimized. This leads to more efficient and streamlined software.
* **Security:** Uncover security vulnerabilities in the code. By examining the internal structure of the software, testers can identify potential security risks and ensure that security best practices have been followed.
* **Thoroughness:** It examines all the internal workings of the software. This thoroughness ensures that every part of the code is tested and validated, leading to robust and reliable software.
* **Quality Assurance:** White Box Testing is a critical part of ensuring software quality. By testing the software’s internal structure, White Box Testing ensures that the software functions as expected and meets the required standards.

White Box Testing practices are crucial to the SDLC, contributing to developing high-quality, secure, and efficient software.

**Types of White Box Testing**

Different types of White Box Testing are:

* **Unit Testing:** Imagine you’re building a bicycle. [Unit testing](https://www.browserstack.com/guide/unit-testing-a-detailed-guide) would be like checking each part separately – testing the brakes, the gears, the pedals, etc., to ensure they all work correctly before assembling the whole bicycle.
* **Static Analysis:** This is like proofreading a book before it’s published. You’re looking for errors in grammar, punctuation, and sentence structure. Still, you need to read the book as a whole to understand the story (which would be more like dynamic analysis).
* **Dynamic Analysis:** This would be like test-driving a car. You’re not just looking at the components (like in static analysis), but you’re driving the car to see how it performs on the road.
* **Statement Coverage:** Imagine you’re a teacher checking a student’s homework. Statement coverage would be like ensuring the student has answered every question on the assignment.
* **Branch Testing:** This is like exploring all possible routes on a GPS. If you’re at an intersection, branch testing involves going straight, turning left, and turning right to ensure all paths lead to valid destinations.
* **Path Testing:** This would be like a postman ensuring they can deliver mail to every house on their route. They need to make sure every possible path is covered.
* **Loop Testing:** This is like checking a playlist on repeat. You want to ensure it loops back to the first song correctly after the last song finishes.

**White Box Testing Example**

Here’s an example demonstrating how white box testing works

def Printme(a, b):

    result = a + b

    if result > 0:

        print("Positive", result)

    else:

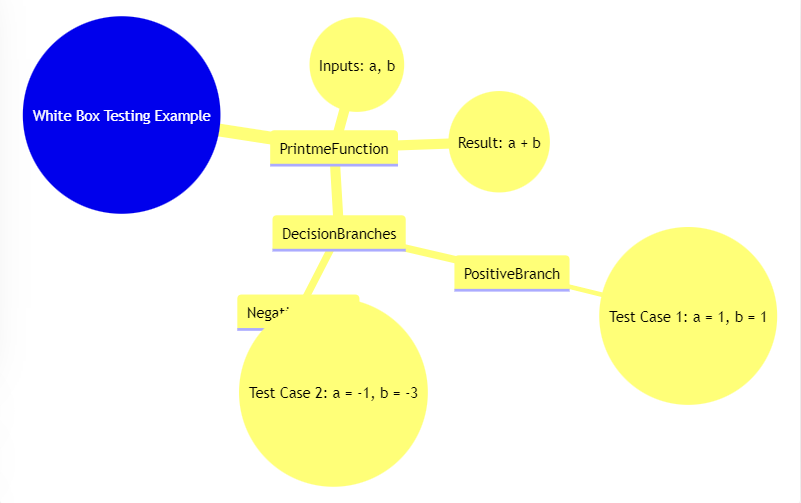
        print("Negative", result)

In this code, Printme is a function that takes two inputs, adds them, and checks whether the result is positive or negative. If the result is positive, it prints “Positive”. If the result is not positive (i.e., zero or negative), it prints “Negative” along with it.

**The goal of White Box Testing here is to verify all the decision branches (the if-else condition) in the code.**

**To exercise the statements in this code, we would create the following test cases**:

* Test Case 1: a = 1, b = 1
  + This would test the “Positive” branch of the if-else condition.
* Test Case 2: a = -1, b = -3
  + This would test the “Negative” branch of the if-else condition.



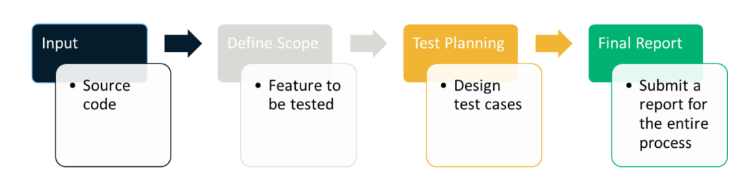
By running these test cases, we can ensure that both branches of the if-else condition in the code are tested, which is the goal of White Box Testing.

**Illustaration:** Imagine this code as a road with a fork, where you can either go left or right. The decision to go left or right is based on whether the result is positive.

The result is positive in the first test case, so we go left (the “Positive” branch). The result is not positive in the second test case, so we go right (the “Negative” branch). By running both test cases, we ensure that we have traveled both possible paths at the fork in the road, which is the goal of White Box Testing.

**What to Verify in White Box Testing?**

In White Box Testing, you’re like a book reviewer who must check every chapter.



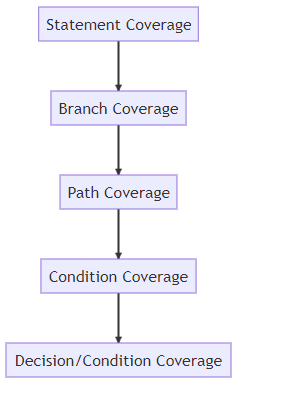
**Here’s what you need to verify in white box testing:**

* **Code Paths**: This is like checking every chapter in the book. You must ensure every part of the code is visited and works correctly.
* **Loops**: This is like checking if the book has any repeated chapters. In code, loops are parts that can run multiple times. You must ensure these loops work correctly and don’t repeat forever.
* **Conditions**: This is like checking if the book has twists and turns. In code, conditions can change what the software does. You need to ensure that every possible outcome of these conditions is tested.
* **Inputs and Outputs**: This is like checking the start and end of the book. You need to ensure that for every input (start), the software produces the correct output (end).
* **Individual Parts**: This is like checking every character in the book. In code, these are the individual functions or objects. You need to make sure each one works correctly on its own.

White Box Testing aims to ensure every part of the code works correctly.

**Different White Box Testing Techniques**

Here are different techniques used to perform white box testing:



* **Statement Coverage:** This is like making sure you read every sentence in a book. In code, it means ensuring every line or statement of the code is executed at least once during testing.
* **Branch Coverage:**This is like choosing every possible outcome in a “Choose Your Own Adventure” book. In code, it means testing every possible outcome of the code’s decision points (like if-else conditions).
* **Path Coverage:** This is like reading a book’s possible combination of chapters. In code, it means testing every possible path through the code from start to finish.
* **Condition Coverage:** This is like checking every possible answer to a question in a book. In code, it means testing every possible outcome of logical conditions in the code.
* **Decision/Condition Coverage:** This is like checking every possible combination of answers to multiple questions in a book. In code, it means testing every possible combination of outcomes in logical decisions (like a condition with and logic) in the code.

These techniques aim to ensure every part of the code is tested.

**Advantages and Limitations of White Box Testing**

White Box Testing has advantages and limitations that can help you decide when to use it.

|  |  |
| --- | --- |
| **Advantages** | **Limitations** |
| **Thorough Testing:** Like reading every chapter of a book, White Box Testing checks every part of the code, making it very thorough. | **Complexity:** Just like a book can be hard to understand if it’s written in a difficult language, White Box Testing can be complex because it requires understanding the code. |
| **Early Bug Detection:** Like spotting a typo in the first few pages of a book, White Box Testing can find bugs early in the development process, which makes them cheaper and easier to fix. | **Time-Consuming:** Just like reading a long book can take a lot of time, White Box Testing can be time-consuming because it’s so thorough. |
| **Improves Security:** Like a book review that warns about inappropriate content, White Box Testing can find security issues in the code, helping to make the software more secure. | **Requires Expertise:** Just like understanding a book written in an old or foreign language requires special knowledge, White Box Testing requires a deep understanding of coding and implementation. |
| **Optimizes Code:** Like a book review that suggests removing unnecessary chapters, White Box Testing can find unnecessary or redundant code, helping to make the software more efficient. | **Bias:** Just like a book reviewer might miss flaws in a book they love, developers who test their own code might miss bugs because they’re too familiar with it. |

White Box Testing is only one tool to help ensure software quality. It’s most effective when used in combination with other testing methods.

**Tools and Frameworks used to perform White Box Testing**

Here’s a list of different tools and frameworks that are used to perform White Box Testing:

Code Coverage Tools

* **JUnit:** A widely-used framework for unit testing in Java. It integrates with various code coverage tools to measure how much of the code is tested.
* **NUnit:** Similar to JUnit but for .NET applications. It supports various code coverage tools.
* **TestNG:** Another testing framework for Java that supports parallel test execution and integrates with code coverage tools.

Code Analysis Tool

* **Code Quality:** BrowserStack Code Quality helps check the code and ensure it follows all the required standards.

Unit Testing Frameworks

* **JUnit:** For Java applications, JUnit is a popular framework for writing and running tests.
* **NUnit:** For .NET applications, NUnit provides a framework for unit testing.
* **pytest:** A testing framework for Python that supports fixtures, parameterized testing, and other advanced features.
* **xUnit.net:** A testing framework for .NET languages that provides a wide range of features for unit testing.

IDE Integrations

* **Eclipse:** Provides plugins for JUnit, code coverage tools, and static analysis tools.
* **IntelliJ IDEA:** Supports various testing frameworks and integrates with code coverage and static analysis tools.
* **Visual Studio:** Offers support for NUnit, xUnit, and code coverage tools for .NET applications.

**Best Practices for White Box Testing**

Here are some of the best practices to follow for efficient white box testing:

* **Familiarize Yourself with the Code:** Before designing test cases, ensure you have a deep understanding of the codebase, including its architecture, design patterns, and algorithms.
* **Review Documentation:** Use design documents, code comments, and architectural diagrams to understand the code’s structure and functionality.
* **Path Testing:** Create test cases that cover all possible execution paths through the code. This helps ensure that all routes are tested.
* **Branch Testing:** Ensure that each decision point or branch in the code is tested for both true and false conditions.
* **Condition Testing:** Verify that each condition in decision statements is evaluated both to true and false.
* **Measure Coverage:** Use code coverage tools to track which parts of the code are being tested. Aim for high coverage.
* **Automate Unit Tests:** Use testing frameworks (e.g., JUnit, NUnit, pytest) to automate unit tests, ensuring consistent and repeatable testing.
* **Continuous Integration:** Integrate automated tests into your CI/CD pipeline to run tests automatically with each code change or deployment.
* [**Peer Reviews**](https://www.browserstack.com/guide/what-is-peer-testing)**:** Regularly conduct code reviews to ensure that the code is well-structured, follows best practices, and is easy to understand. It helps identify test cases and potential edge cases.
* **Maintain and Update Documentation:** Document test cases, test results, and any issues discovered. Keep documentation up-to-date with changes in the codebase to ensure that test cases remain relevant.
* **Test Edge Cases:** Design test cases for edge cases and boundary conditions. These often reveal issues that are not apparent in regular scenarios.
* **Exception Handling:** Verify that exceptions are handled correctly and that the code behaves as expected in error situations.
* **Continuous Improvement:** Regularly review and refine your testing strategy based on test results, coverage metrics, and feedback.

**Black box testing**

The Test Engineer performs the black box testing. The code is not visible while testing, which is why it is referred to as black-box testing. Black box testing is a software testing technique that checks program functionality without delving into its core structure or coding. The **primary source of black box testing** is a customer-specified requirement specification. In this method, the tester selects a function and provides an input value to examine its functionality, and then determines whether or not the function produces the intended result. If the function returns correct results, it passes testing; otherwise, it fails.

Black box testing is a software testing technique where the internal workings or code structure of the system being tested are not known to the tester.

In other words, the tester focuses solely on the external behaviour of the software, without having access to its internal source code. The name “black box” comes from the idea that the internal workings are hidden or “boxed” from the tester’s view.

Key characteristics of black box testing include:

* **Independent Testing**: Black box testing is typically performed by testers who are independent of the development team. This ensures a fresh perspective and helps identify issues that developers might overlook.
* **Requirements-Based Testing**: Testers design test cases based on the software’s requirements and specifications, without being concerned about how the code is implemented.
* [**Functional Testing**](https://www.browserstack.com/guide/functional-testing): The main goal of black box testing is to assess the functionality of the software, checking if it meets the expected behaviour and delivers the desired outputs for various inputs.
* **No Knowledge of Internal Code**: Testers do not have access to the source code, architecture, or design details of the software. They interact with the system through its user interfaces or APIs.

**Different Types of Black Box Testing**

Black box testing encompasses several types of testing techniques, each with a specific focus and objective. Some of the main types of black box testing include:

* [**Functional Testing**](https://www.browserstack.com/guide/functional-testing)**:**This type of black box testing verifies that the software’s functions and features work as expected and adhere to the specified requirements. Testers use functional test cases to validate the application’s inputs, outputs, and interactions, without being concerned about the internal code.
* [**Non-Functional Testing**](https://www.browserstack.com/guide/what-is-non-functional-testing): Unlike functional testing, non-functional testing evaluates aspects of the software that are not related to its specific functions. It includes tests for performance, usability, security, scalability, reliability, and other quality attributes.
* [**Regression Testing**](https://www.browserstack.com/guide/regression-testing)**:** Regression testing is performed to ensure that recent changes or updates to the software do not adversely affect existing functionality. Testers use a set of predefined test cases to verify that new features or bug fixes have not introduced new issues.
* **User Interface (UI) Testing:** [UI testing](https://www.browserstack.com/guide/ui-testing-guide) focuses on validating the user interface elements of the software, such as buttons, menus, forms, and layout. The goal is to ensure that the UI is [user-friendly](https://www.browserstack.com/guide/make-website-user-friendly), consistent, and functions correctly.

* [**Usability Testing**](https://www.browserstack.com/guide/what-is-usability-testing)**:** Usability testing assesses the software’s user-friendliness and how easily users can interact with it. Testers evaluate factors like navigation, visual appeal, ease of learning, and overall user experience.
* **Boundary Value Analysis (BVA):** BVA is a technique used to identify defects around the boundaries of input values. Test cases are designed with values at the edges of input ranges to assess how the software handles minimum and maximum limits.
* **Equivalence Partitioning:**In this technique, the input domain is divided into groups of data that are expected to behave similarly. Test cases are then derived from these partitions to minimise redundant testing.
* [**Ad-hoc Testing**](https://www.browserstack.com/guide/adhoc-testing): Ad-hoc testing is an informal and unstructured testing approach where testers explore the software freely, executing test scenarios based on their intuition and experience. It helps identify defects that might be missed by formal test cases.
* [**Compatibility Testing**](https://www.browserstack.com/guide/compatibility-testing): Compatibility testing assesses how well the software performs across different environments, such as various browsers, operating systems, devices, and network configurations.

* **Security Testing:**Security testing aims to identify vulnerabilities and weaknesses in the software’s security measures. Testers simulate attacks and check for potential security breaches.
* **Localization and Internationalization Testing:**These types of testing ensure that the software is adapted to different languages, cultures, and regional settings, and it functions correctly in various international environments.

**Different Black Box Testing Techniques**

Black box testing techniques focus on evaluating software from the user’s perspective, without delving into the internal code structure or logic.

**1. Equivalence Partitioning:** Divides the input data into equivalent partitions, with each partition being regarded the same by the program. Testing one representative from each partition is usually enough to cover all potential scenarios.

**Example:** For a form that accepts age input between 18 and 65, equivalence partitions might include:

* Valid partition: 18-65 (e.g., age 25)
* Invalid partition: Below 18 (e.g., age 15)
* Invalid partition: Above 65 (e.g., age 70)

**2. Boundary Value Analysis:** Tests the bounds of input ranges, as errors frequently arise on the edge of input limits.

**Example:** For an input field that accepts values from 1 to 100, boundary values would include:

* Lower boundary: 1
* Just below lower boundary: 0
* Just above upper boundary: 101
* Upper boundary: 100

**3. Decision Table Testing:** A decision table is used to represent and test different combinations of inputs and predicted outcomes. This method is effective for testing systems that involve several conditions and actions.

**4. State Transition Testing:** Tests the system’s behaviour in various states and transitions between them. It ensures that the system functions properly when transitioning from one state to another.

**Example:** For a user login system, states might include:

* Logged Out
* Logged In
* Suspended

**Transitions would be:**

* From Logged Out to Logged In (successful login)
* From Logged In to Suspended (suspend account)
* From Suspended to Logged Out (logout from suspended state)

**5. Use Case Testing:** Focuses on validating the functionality of the system based on user interactions described in use cases. It ensures that the system meets the requirements of each use case.

Example: For an online shopping application, a use case might be:

* **Use Case:** Purchase Item
* **Steps:** Select item, add to cart, proceed to checkout, enter payment details, confirm purchase
* **Expected Outcome:** Order confirmation is displayed, and order is recorded

**6. Error Guessing:** Relies on the tester’s experience and intuition to guess where errors might occur based on common mistakes, past experiences, and known problem areas.

**Example:** For a file upload feature, error guessing might include testing with:

* Files of various types (e.g., .exe, .jpg, .pdf)
* Files with very large sizes
* Files with invalid extensions

**7. All-pair Testing Technique:** All-pair testing, also known as pairwise testing, is a combinatorial testing technique that aims to cover all possible pairs of input parameters in a test set.

The purpose is to ensure that every combination of two input parameters is evaluated at least once, which aids in the detection of problems caused by interactions between parameter pairs.

**Consider a web application with three input parameters:**

* **Parameter 1:** Browser Type (Chrome, Firefox)
* **Parameter 2:** Operating System (Windows, macOS)
* **Parameter 3:** User Role (Admin, Guest)

With each parameter having two possible values, there are 2 x 2 x 2 = 8 possible combinations if tested exhaustively. However, using all-pair testing, you might only need a subset of combinations to cover all pairs of values.

**Possible Test Cases:**

* **Browser:** Chrome, OS: Windows, Role: Admin
* **Browser:** Chrome, OS: macOS, Role: Guest
* **Browser:** Firefox, OS: Windows, Role: Guest
* **Browser:** Firefox, OS: macOS, Role: Admin

**These test cases ensure that each pair of input values is tested, such as:**

* Browser Type and Operating System
* Browser Type and User Role
* Operating System and User Role

**8. Cause-Effect Technique:** The Cause-Effect approach, also known as Cause-Effect Graphing, is a black-box testing method that creates test cases based on the relationships between causes (inputs) and effects (outputs).

This technique aids in systematically determining the functional correctness of a system by visualising and analysing the logical linkages between various situations and actions.

**Some key concepts here are:**

* **Cause-Effect Graph:** A diagrammatic portrayal of the logical relationships between various input conditions (causes) and their anticipated outputs (effects). It assists in discovering and mapping how different inputs interact to produce diverse outcomes, allowing for more effective test case creation.
* **Cause:** An input condition or factor that determines system behaviour. Examples include user inputs, system settings, and configuration parameters.
* **Effect:** The result of the cause. It is the system’s response to the provided input. Examples include system outputs, status messages, and changes in system behaviour.

**Example:**

Consider an online account login system with the following input conditions (causes) and expected outputs (effects):

**Causes:**

* Correct username
* Correct password
* Incorrect username
* Incorrect password
* Account locked

**Effects:**

* **Login Success:** If the username and password are both correct and the account is not locked.
* **Login Failure:** If either the username or password is incorrect, or the account is locked.

**Cause-Effect Graph:**

* Cause 1 + Cause 2 → Effect 1 (Successful Login)
* Cause 3 + Cause 4 → Effect 2 (Login Failure)
* Cause 5 → Effect 3 (Account Locked)

**Derived Test Cases:**

* Correct username + Correct password (Expected: Login Success)
* Correct username + Incorrect password (Expected: Login Failure)
* Incorrect username + Correct password (Expected: Login Failure)
* Account locked + Correct username + Correct password (Expected: Account Locked)

**Example of Black Box Testing**

A simple black box testing example for a login functionality of a web application. In this scenario, we will test the login page without having access to the internal code or implementation details.

**Test Case Name**: Verify successful login with valid credentials.

**Test Steps**:

1. Open the web browser.
2. Enter the URL of the application’s login page.
3. Enter a valid username in the username field.
4. Enter a valid password in the password field.
5. Click on the “Login” button.
6. Wait for the application to process the login request.

**Expected Result**: The user should be successfully logged into the application’s dashboard/homepage.

**Test Case Status**: PASS (if the user is redirected to the dashboard/homepage)

**Test Case Name**: Verify unsuccessful login with invalid credentials.

**Test Steps**:

1. Open the web browser.
2. Enter the URL of the application’s login page.
3. Enter an invalid username (e.g., “invaliduser”) in the username field.
4. Enter an invalid password (e.g., “wrongpassword”) in the password field.
5. Click on the “Login” button.

Wait for the application to process the login request.

**Expected Resul**t: The login attempt should fail, and an appropriate error message (e.g., “Invalid username or password”) should be displayed on the login page.

**Test Case Status**: PASS (if the error message is displayed)

Features of Black Box Testing

Black box testing, as a software testing approach, offers several features and benefits that make it an essential part of the software development process. Some key features of black box testing include:

1. **Focus on External Behavior**: Black box testing emphasises evaluating the software’s functionality from an end-user perspective, focusing on how the system behaves with different inputs and usage scenarios.
2. **Independence from Internal Code**: Testers conducting black box testing do not require knowledge of the internal code or implementation details, making it suitable for testers who may not have programming expertise.
3. **Requirement-Based Testing**: Test cases in black box testing are designed based on the software’s requirements and specifications. This ensures that the application meets the intended functionality and business objectives.
4. **Real-World Scenario Testing**: Black box testing helps simulate [real-user conditions](https://www.browserstack.com/real-user-conditions-testing-on-browserstack), allowing testers to identify defects that might arise during actual usage of the software.
5. **Validation of Interfaces**: It is effective in verifying the accuracy of the software’s interfaces, ensuring that inputs and outputs are correctly handled.
6. **Identification of Interface-Level Bugs**: Black box testing is particularly useful for detecting interface-level bugs, such as incorrect error messages, incorrect data handling, or missing functionality.
7. **User-Centric Testing**: By focusing on the end-user perspective, black box testing ensures that the application meets user expectations and delivers a satisfactory user experience.
8. **Test Case Design Techniques**: Black box testing employs various test case design techniques, such as equivalence partitioning, boundary value analysis, decision table testing, and state transition testing, to ensure comprehensive test coverage.
9. **Compatibility Testing**: It helps assess the software’s compatibility with different environments, browsers, operating systems, and devices.
10. **Test Automation Support**: Many black box testing tools support test automation, enabling the execution of repetitive test cases efficiently and reducing the testing cycle time.

By leveraging these features, black box testing helps organizations deliver high-quality software that meets user expectations and complies with the specified requirements. When combined with other testing approaches, such as white box testing and gray box testing, it provides a comprehensive testing strategy for software development projects.

**Advantages and Limitations of Black Box Testing**

**Advantages of Black Box Testing**

* **Independence from Internal Implementation:**Testers do not need to have access to the source code or knowledge of the internal implementation, making it suitable for non-technical team members.
* **User-Centric Testing:** Black box testing focuses on the software’s external behavior, ensuring that it meets user requirements and expectations.
* **Testing from End-User Perspective:** It simulates real user scenarios, helping to identify usability issues and ensuring the software meets user needs.
* **Early Detection of Interface Issues:** Black box testing can uncover interface-related defects, such as input validation errors and output discrepancies.
* **Effective at Integration Testing:** It verifies the interactions between different system components, making it valuable for integration testing.
* **Test Case Design Flexibility:** Various test case design techniques, such as equivalence partitioning and boundary value analysis, allow for effective test coverage.
* **Effective for Requirement Validation:**Black box testing helps validate that the software meets the specified requirements.
* **Suitable for Large Projects:**It can be applied at different testing levels, from unit testing to acceptance testing, making it scalable for large projects.

**Limitations of Black Box Testing**

* **Limited Code Coverage:**Black box testing may not explore all possible code paths or internal logic, potentially leaving certain defects undetected.
* **Inability to Test Complex Algorithms:** It may not be effective at validating complex algorithms or intricate business logic that requires knowledge of the internal code.
* **Redundant Testing:** Some test cases may overlap, leading to redundant testing efforts and less optimal test coverage.
* **Dependency on Requirements:**Test cases are heavily dependent on the accuracy and completeness of the provided requirements. Incomplete or ambiguous requirements can result in incomplete testing.
* **Inefficiency with Repetitive Tasks:** Manual black box testing can be time-consuming and inefficient for repetitive tasks, making test automation essential for large-scale projects.
* **Inability to Assess Performance and Scalability:** Performance-related issues and scalability problems may not be effectively identified through black box testing alone.
* **Difficulty in Error Localization:** Identifying the root cause of defects detected in black box testing can be challenging, as testers lack access to internal code.
* **Limited Security Testing:**While black box testing can identify certain security vulnerabilities, it may not comprehensively address all potential security issues.

To overcome some of these limitations, organisations often use a combination of black box testing with other testing approaches like white box testing (to assess internal code and logic) and gray box testing (to combine elements of both black and white box testing).

This mixed approach allows for better test coverage and increased software quality assurance.

**Tools and Frameworks used to perform Black Box Testing**

There are several black box testing tools available that can assist testers in automating and managing the testing process for software applications. These tools help with creating and executing test cases, capturing test results, and generating reports.

Some popular black box testing tools include:

**1. Selenium**[Selenium](https://www.browserstack.com/selenium) is commonly used for black box testing, particularly for web applications. Selenium is an open-source testing framework that allows testers to automate the testing of web browsers, making it a valuable tool for performing black box testing on web-based systems. It interacts with web elements on the user interface, simulating real user interactions and validating the functionality of the application without accessing its internal code.

**2. Appium**[Appium](https://www.browserstack.com/app-automate/appium) is another popular tool that is often used for black box testing, particularly for mobile applications. Appium is an open-source test automation framework that allows testers to automate the testing of native, hybrid, and mobile web applications on both Android and iOS devices. It enables black box testing of mobile apps without accessing the internal code.

**Pro Tip** :It is advised to perform Selenium and Appium Tests on real device cloud to obtain more accurate test results.

The [Cloud Selenium Grid](https://www.browserstack.com/cloud-selenium-grid?utm_source=google&utm_medium=cpc&utm_platform=paidads&utm_content=602353912717&utm_campaign=Search-DSA-NB-Europe&utm_campaigncode=Guide-Pages+1007850&utm_term=+) of BrowserStack provides access to over 3000 browser device combinations, enabling QAs to test under actual user conditions for improved performance.

It is simple to test your native and hybrid mobile applications with [BrowserStack App Automate and the Appium automation framework](https://www.browserstack.com/docs/app-automate/appium" \o "BrowserStack App Automate and the Appium automation framework" \t "_blank). Test on a large number of actual Android and iOS devices.

**3. Cypress**  
[Cypress](https://browserstack.com/automate/cypress) is a powerful test automation framework primarily used for front-end testing, including [end-to-end (E2E) testing](https://www.browserstack.com/guide/cypress-end-to-end-testing) and user interface (UI) testing. While Cypress is more commonly associated with white box testing due to its ability to access and control the application’s internal code, it can also be used for black box testing to some extent.

While Cypress may provide some black box testing capabilities, its real strength lies in the combination of white box and black box testing. For instance, testers can use Cypress to conduct E2E tests and then complement it with other black box testing techniques like [exploratory testing](https://www.browserstack.com/guide/exploratory-testing) or [usability testing](https://www.browserstack.com/guide/what-is-usability-testing).

**4. Load Runner**LoadRunner is primarily known as a performance testing tool, and its core focus is on testing the performance, scalability, and reliability of applications under different load conditions. While LoadRunner is not typically used as a dedicated black box testing tool, it can still be employed to perform some aspects of black box testing in specific scenarios, for e.g.

* **Load Testing with Real User Scenarios:**LoadRunner can simulate real user scenarios and interactions with the application. In this sense, it acts as a black box, not having direct access to the application’s internal code.
* **User Experience Testing:** By conducting load tests with multiple virtual users, LoadRunner can help assess the overall user experience. It measures the application’s response times, resource utilization, and other performance metrics, simulating real-world scenarios from the end-user perspective.

**5. SoapUI**SoapUI is primarily known as an API testing tool, and its main focus is on testing the functionality and behavior of APIs (Web services). As such, SoapUI is well-suited for black box testing of APIs, ensuring that they meet the specified requirements without needing access to the underlying code.

Here’s how SoapUI can be used for black box testing of APIs:

* **Functional Testing:** SoapUI allows testers to create test cases that simulate API requests and responses. Testers can validate if the API functions correctly based on the expected results without knowing the internal implementation.
* **Input Validation:** Testers can use SoapUI to check how the API handles different types of inputs and whether it provides the appropriate responses, such as error messages for invalid data.
* **Boundary Value Analysis:** SoapUI allows testers to test API responses with boundary values to verify if the API behaves correctly at the edges of the input range.

For comprehensive black box testing, especially when dealing with end-to-end testing of web applications, it is recommended to use dedicated black box testing tools like Selenium or Cypress in combination with SoapUI for API testing. This combination allows for a more complete testing approach, covering both the functionality of the APIs and the user interface interactions of the application.

**Best Practices for Black Box Testing**

Effective black box testing requires careful planning, thorough test case design, and meticulous execution. Here are some best practices to ensure successful black box testing:

* **Requirement Analysis:** Start by thoroughly understanding the software’s requirements and specifications. Clear and well-defined requirements will guide the creation of meaningful test cases.
* **Test Planning:** Develop a comprehensive test plan that outlines the testing scope, objectives, testing levels, resources, and timelines. This will serve as a roadmap for the testing process.
* **Test Case Design Techniques:** Utilize various test case design techniques like equivalence partitioning, boundary value analysis, decision tables, and state transition testing to ensure comprehensive test coverage.
* **Test Data Management:**Prepare relevant and diverse test data to cover various scenarios. Validate both valid and invalid inputs to assess the software’s response.
* **Positive and Negative Testing:**Include test cases for both positive scenarios (valid inputs with expected outcomes) and negative scenarios (invalid inputs with appropriate error handling).
* **Usability Testing:** Focus on testing the user interface and overall user experience. Verify that the application is user-friendly, consistent, and easy to navigate.
* **Regression Testing:**As changes are made to the software, perform regression testing to ensure that new updates or fixes do not introduce new defects or impact existing functionality.
* **Boundary Value Analysis:**Test the software’s behavior around the boundaries of input ranges to identify potential issues with boundary conditions.
* **Error Localization and Reporting:**Clearly document and report any defects or issues discovered during testing, including detailed steps to reproduce the problem and information on the test environment.
* **Test Automation:**Automate repetitive and time-consuming test cases to improve testing efficiency and repeatability. Automation helps in running tests more frequently and consistently.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Black Box Testing** | **White Box Testing** |
| Meaning | It is a software testing method in which the program or the internal structure stays hidden, and the tester has no knowledge about it. | It is a software testing method in which the tester knows about the code or the internal structure and the program involved. |
| Type of Testers | Mostly, software testers do this. | Mostly, hardware developers do this. |
| Knowledge Required | The user doesn’t require knowledge regarding implementation. | A user requires knowledge for implementation. |
| Type of Testing | It is a form of external or outer software testing. | It’s the inner or internal way of software testing. |
| Source Code Access | The tester cannot access the source code of the software. | The tester is aware of the source code and internal workings of the software. |
| Interface of Operation | You can conduct Black box testing at the software interface. It requires no concern with the software’s internal logical structure. | You can conduct White box testing by ensuring that all the internal operations take place according to the specifications. |
| Purpose of Testing | It tests the functions of the software. | It tests the structure of the software. |
| Basis of Initiation | You can initiate this test on the basis of the requirement specifications document. | You can only start this type of testing software after a detailed design document. |
| Programming Knowledge | Testers don’t require a knowledge of programming. | Testers mandatorily require a knowledge of programming. |
| Assessment | This testing assesses software behavior. | This testing assesses software logic. |
| Level of Testing | Higher levels of software testing generally involve Black Box testing. | Lower levels of software testing usually involve White Box testing. |
| Other Names | You can also call it closed testing. | You can also call it clear box testing. |
| Time Consumed | It consumes less time. | It consumes more time. |
| Algorithm Testing | It does not work well for algorithm testing. | It is completely suitable and preferable for algorithm testing. |
| Methods of Testing | One can perform it using various trial and error methods and ways. | You can better test data domains and internal or inner boundaries. |
| Types of Testing | **Black Box Testing types:**   * Regression Testing * Functional Testing * Non-Functional Testing | **White Box Testing types:**   * Condition Testing * Path Testing * Loop Testing |
| Examples | **Example:** A user searching something on a search engine like Google using certain keywords. | **Example:** When a user inputs to check and verify the loops. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Black Box** | **White Box** | **Grey Box** |
| Definition | Testing without knowledge of the internal code structure; focuses on inputs and outputs. | Testing with knowledge of the internal code structure; focuses on logic, paths, and structures. | A combination of black box and white box testing; involves some knowledge of the internal workings but focuses on functionality. |
| Focus | Functionality and user requirements. | Internal logic, code structure, and pathways. | Functionality and internal design, focusing on both the end-user perspective and internal processes. |
| Knowledge Required | No knowledge of the internal code or system architecture. | Detailed knowledge of the internal code and system architecture. | Partial knowledge of the internal code or architecture. |
| Test Basis | Requirements, specifications, and user scenarios. | Source code, algorithms, and internal logic. | Requirements, specifications, and a partial understanding of the code. |
| Types of Tests | Functional testing, system testing, acceptance testing. | Unit testing, integration testing, and code coverage analysis. | Integration testing, security testing, and system testing. |
| Testing Scope | Broad, focusing on overall system functionality. | Narrower, focusing on specific code paths and logic. | Mid-level, focusing on both functional and structural aspects. |
| Test Design | Test cases derived from functional requirements and user stories. | Test cases derived from code and design documents. | Test cases based on functional requirements and some knowledge of the system’s internal workings. |
| Tools Used | Test management tools, and functional testing tools. | Debuggers, code analysis tools, and unit testing frameworks. | Functional and security testing tools, along with some code analysis tools. |