1. **Black Box Testing**

Black box testing is a software testing approach where the tester evaluates the functionality of an application without knowing its internal structure, code, or implementation details.

The key objective of Blackbox testing are

It Validate the application’s functionality against the requirements.

It Ensure the software works as expected for valid inputs and gracefully handles invalid inputs.

It Detect issues in boundary values and edge cases.

It Focuses on inputs and outputs ieTest scenarios, inputs, and expected outcomes.

Identify discrepancies between the expected and actual outputs.

Focus on user experience, usability, and error handling.

Real-time Example:

If we Consider an e-commerce website's login feature. The tester provides various combinations of valid and invalid credentials to verify the login functionality, such as:

Test case: Enter a valid username and password.
Expected result: The user successfully logs in and is redirected to the homepage.
Test case: Enter an invalid password three times.
Expected result: The account is locked, and the user sees an error message.
In black box testing, testers don’t need to know how the system verifies credentials; they only care about the input-output behavior.

White Box Testing

White box testing evaluates the internal structure, logic, and code paths of an application. The tester requires knowledge of the underlying code.

Key Objectives of white box testing are:

It Validate internal operations, such as code correctness and logic.

It Test all possible execution paths and loops for completeness.

It Ensure the software handles edge cases and boundary conditions properly.

It Optimize the performance and efficiency of the code.

Identify code vulnerabilities like SQL injection or buffer overflow

Real-time Example:

For the same e-commerce website, a tester reviews the login module code to ensure:

The tester analyzes the authentication algorithm.

Scenario: Test whether the hash function for encrypting passwords works as expected.

Expected result: A user's plain-text password gets hashed correctly and matches the stored hash for login.

The tester might also check edge cases, such as:

Empty inputs leading to code execution errors.

SQL injection attempts to ensure query sanitization.

2.**Comparison of Weak Normal Equivalence Class Testing with Other Types**

Equivalence Class Testing

Equivalence class testing divides input data into partitions (equivalence classes) that are expected to exhibit similar behavior. Tests are then designed for representative values from each class.

Weak Normal Equivalence Class Testing

Definition: Tests only one valid value from each equivalence class at a time, while keeping other variables constant.

Key Characteristic: Focuses on covering valid input values.

Advantage: Provides a basic level of assurance that the system works for normal, valid inputs without redundancy.

Comparison with Other Types:

Strong Normal Equivalence Class Testing:

Description: Tests combinations of valid values from all equivalence classes together.

Difference: Weak normal tests one representative value per class, whereas strong normal tests combinations, making it more exhaustive.

Weak Robust Equivalence Class Testing:

Description: Tests one valid value at a time along with boundary and invalid values.

Difference: Weak normal focuses only on valid inputs, while weak robust includes invalid inputs to validate error handling.

Strong Robust Equivalence Class Testing:

Description: Tests multiple combinations of valid and invalid values together.

Difference: Strong robust is the most comprehensive and ensures robust error handling for all possible input combinations.

Real-time Example for Equivalence Class Testing

Consider a system requiring:

Username: 5-10 characters

Password: Must include a special character

Age: 18-60

Weak Normal Testing: A form with **Username (5-10 chars), Age (18-60), and Password (special char required).** Testing username with a valid length while keeping valid values for age and password.

Username = "User123" (Valid), while keeping a valid password and age.

Password = "Pa$$word" (Valid), while keeping a valid username and age.

Age = 25 (Valid), while keeping a valid username and password.

Strong Normal Testing: Testing **valid username, valid age, and valid password** at the same time.

Username = "User123", Password = "Pa$$word", Age = 25 (All valid together).

Weak Robust Testing: Testing username with a valid length while also checking an **invalid age (e.g., 17 or 61).**

Username = "Us" (Invalid, too short), while keeping valid password and age.

Password = "password" (Invalid, missing special character), while keeping valid username and age

Strong Robust Testing: Testing **invalid username, invalid password, and invalid age** in a single test case.

Username = "Us" (Too short), Password = "password" (No special char), Age = 17 (Too young)