White Box Testing

**✅ Definition**

**White box testing is a strategy in which testing is based on the internal paths, structure, and implementation of the software under test (SUT)**

**📌 Explanation:**

White box testing (also called **clear box**, **glass box**, or **structural testing**) is a software testing technique that tests the internal logic and structure of the code. Unlike black box testing, which only considers the software's inputs and outputs, white box testing **requires knowledge of the source code**.

**📌 Key Terms:**

* **Internal paths**: Sequences of execution through the code.
* **Structure**: Program control structures like loops, branches, functions.
* **SUT**: Software Under Test.

**📌 Why it’s important:**

* Detects **logical errors**, **design errors**, and **implementation bugs** early.
* Ensures **code coverage** (all statements and branches are tested).
* Helps in **optimization** and **security validation**.

**Applicability of White Box Testing**

**✅ White box testing can be applied at all levels of system development**

* **Unit Testing** 🧱
* **Integration Testing** 🧩
* **System Testing** 🖥️

**Generally white box testing is equated with unit testing performed by developers**

**📌 Explanation:**

* **Unit testing** involves testing individual modules or components. Developers use white box testing techniques to ensure that each function or method behaves correctly.
* Although applicable to higher levels, **unit testing is where white box is most effective** because of direct access to code internals.

**White Box Testing - Applicability**

**✅ White Box Testing Techniques:**

* **Control flow testing**
* **Data flow testing**

**📌 Control Flow Testing:**

Analyzes the logical paths the program might take during execution.

**📌 Data Flow Testing:**

Focuses on the **variables**, how they are defined, used, and killed (i.e., their lifecycle in memory).

**Flow of Control-Based Testing**

**✅ Based on the flow of control in the program:**

* **Logical decisions** (e.g., if, switch)
* **Loops** (e.g., for, while)
* **Execution paths**
* **Coverage metrics**: Quantitative measure of testing completeness.

**Coverage metrics measure how complete the test cases are — not how good they are!**

**📌 Example:**

A test case might execute all code lines (100% coverage), but it might not cover **all edge cases**, or **combinations of decisions**.

**Control Flow Testing**

**✅ Focuses on:**

* **Boolean logic**: AND (&&), OR (||), XOR, etc.
* Analyzes **control flow** through conditions and decisions.
* Determines **which test cases** are "good" by measuring **coverage**.

**Control Flow Graphs (CFG)**

**A program graph is a directed graph where nodes are statement fragments and edges represent the flow of control**

**📌 Graph Definitions:**

* **Directed Graph**: Arrows (edges) show direction of flow.
* **Undirected Graph**: Edges do not have direction (not used in control flow).

**📌 Example:**

For the code:

cpp

CopyEdit

if (x > 5)

y = 10;

z = 20;

CFG would be:

* Node 1: if (x > 5)
* Node 2: y = 10
* Node 3: z = 20
* Edge from Node 1 to Node 2 (if true)
* Edge from Node 1 to Node 3 (if false)
* Edge from Node 2 to Node 3

**Levels of Code Coverage**

**✅ Types of Coverage:**

| **Type** | **Description** |
| --- | --- |
| **Statement/Line/Block Coverage** | Executes every line at least once. |
| **Decision (Branch) Coverage** | Executes each branch (true/false) of every decision. |
| **Condition Coverage** | Executes each individual boolean condition in a decision both as true and false. |
| **Multiple Condition Coverage** | Executes **all combinations** of boolean condition values. |
| **Decision/Condition Coverage** | Ensures both the overall decision and individual conditions are tested. |
| **Loop Coverage** | Ensures loops execute for 0, 1, many iterations. |
| **Path Coverage** | Covers all possible unique execution paths. |

**Statement Coverage**

**✅ Goal:**

**Execute each statement at least once**

**📌 Example:**

cpp

CopyEdit

Begin

if (y >= 0)

y = 0;

abs = y;

End

**CFG:**

* Node 1: if (y >= 0)
* Node 2: y = 0
* Node 3: abs = y

**Test Case 1 (Yes path):**

* **Input**: y = 0
* **Expected/Actual Result**: abs = 0
* ✅ All statements executed → **100% statement coverage**

**⚠️ Limitations:**

* Might miss errors if the **false branch** is never executed (e.g., y < 0 not tested).

**Line Coverage: Pros and Cons**

**✅ Used because:**

* **Simple** to measure.
* Available via many testing tools (e.g., JUnit, gcov).

**❌ Problems:**

* **Weakest** form of coverage.
* May **miss bugs** in conditions (e.g., if not evaluated as false).
* **False confidence** — appears well-tested, but edge cases might be untested.

Experts recommend using it **only when other coverage types are unavailable**.

**Decision (Branch) Coverage**

**Execute each edge in the CFG at least once**

**📌 Goal:**

* Test both **true and false** outcomes of each decision.

**📌 Code:**

cpp

CopyEdit

Begin

if (y >= 0)

y = 0;

abs = y;

End

**✅ Test Cases:**

1. **Input**: y = 0 → executes **true branch**
2. **Input**: y = -5 → executes **false branch**

✅ Now, both branches of if are tested → **100% branch coverage**

**Condition Coverage**

**Each condition in a decision must be evaluated both true and false**

**📌 Example:**

cpp

CopyEdit

if (x < 10 && y > 20)

z = foo(x, y);

else

z = fie(x, y);

**✅ Test Cases:**

| **Case** | **x** | **y** | **x < 10** | **y > 20** |
| --- | --- | --- | --- | --- |
| 1 | -4 | 12 | T | F |
| 2 | 12 | 30 | F | T |

✅ Each condition (x < 10 and y > 20) is true in one case and false in the other.

**Multiple Condition Coverage**

**All combinations of conditions must be executed**

**📌 For the same code:**

| **Case** | **x < 10** | **y > 20** |
| --- | --- | --- |
| 1 | T | T |
| 2 | T | F |
| 3 | F | T |
| 4 | F | F |

✅ Covers **every possible combo** of truth values → thorough condition testing.

**Decision/Condition Coverage**

**Satisfies both decision and condition coverage**

**📌 Example:**

* Test Case 1: x = -4, y = 30 → T, T → Decision is True
* Test Case 2: x = 12, y = 12 → F, F → Decision is False

✅ Covers:

* Each condition is T and F at least once.
* Decision is both True and False.

**Loop Coverage**

**✅ Types of Loops:**

* **Simple**: Single for/while
* **Nested**: One loop inside another
* **Concatenated**: Multiple loops, one after another
* **Unstructured**: Irregular loop flows (e.g., goto)

**✅ Testing Strategy:**

**Simple Loop:**

* **0 iterations** (skipped)
* **1 iteration**
* **2 iterations**
* **m iterations** (typical value)
* **n-1, n, n+1 iterations** (boundary testing)

**Nested Loops:**

* Start with **innermost loop**.
* Keep outer loop values fixed.
* Add more complex cases gradually.

**Concatenated Loops:**

* If **independent**, test each as simple.
* If **dependent**, test like nested.

**Unstructured Loops:**

* Avoid if possible!
* Refactor into structured programming form.

**Path Coverage**

**A path is a sequence of decisions/conditions in code**

**✅ Goal:**

* Each **unique path** through the code is tested.
* Emphasizes **branches and conditions** — where most bugs occur.

**Cyclomatic Complexity**

**✅ Definition:**

Measures the number of **linearly independent paths** in a program using the **Control Flow Graph**.

**📌 Formula:**

V(G)=E−N+2PV(G) = E - N + 2PV(G)=E−N+2P

Where:

* **E** = number of edges in CFG
* **N** = number of nodes
* **P** = number of connected components (usually 1 for a single program)

**✅ Interpretation:**

* Tells you **how many test cases** are needed to cover all paths.
* Higher complexity → more risk, harder to test and maintain.

**✅ Example:**

A function with 1 if and 1 loop might have 3 independent paths → Cyclomatic Complexity = 3

**✅ Next Step**

Would you like me to now continue with the next chapter from your syllabus — **Project Planning & Estimation Techniques** — or stay on this topic and expand further (e.g., data flow testing or additional examples)?

**✅ Loop Coverage – Explained in Detail**

Loop coverage is a white-box testing strategy that ensures loops are thoroughly tested for correctness, especially at **boundaries** and **edge cases**. This is critical since many bugs occur due to incorrect loop conditions, infinite loops, or unexpected exit points.

**🔸 1. Simple Loops**

**✅ Definition:**

A **simple loop** is a basic loop construct like for, while, or do-while, that iterates a fixed number of times with **one entry and one exit point**.

**✅ Example:**

cpp

CopyEdit

for (int i = 0; i < n; i++) {

process(i);

}

**✅ Test Cases for Simple Loops:**

| **Case** | **Description** | **Example (n = 10)** |
| --- | --- | --- |
| 0 iterations | Loop skipped | n = 0 |
| 1 iteration | Loop runs once | n = 1 |
| 2 iterations | Loop runs twice | n = 2 |
| m iterations | Loop runs a small typical number | m = 5, n = 10 |
| n-1, n, n+1 iterations | Boundary values | n = 9, 10, 11 |

**✅ Why Important?**

These tests ensure the loop:

* Is **entered and exited correctly**
* **Handles boundaries** safely
* **Does not go infinite**
* **Executes logic as expected** at start, middle, and end

**🔸 2. Nested Loops**

**✅ Definition:**

One loop is **inside another**. Inner loop executes completely for each iteration of the outer loop.

**✅ Example:**

cpp

CopyEdit

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

process(i, j);

}

}

**✅ Test Strategy:**

1. **Start with the innermost loop**: Test it using **simple loop strategy** (0, 1, 2, m, n-1, n, n+1).
2. Keep **outer loop values fixed at minimal valid range**, e.g., n = 1
3. Gradually increase outer loop complexity while ensuring inner loop is thoroughly tested.
4. Test for **boundary conditions** at each nesting level.

**✅ Example Test Cases:**

| **Outer Loop (i)** | **Inner Loop (j)** | **Description** |
| --- | --- | --- |
| 1 iteration | 0 iteration | Test skip inner loop |
| 1 iteration | 1 iteration | Test minimum nested loop |
| 2 iterations | 3 iterations | Mid-range test |
| n iterations | m iterations | Full nested coverage |
| n-1/n/n+1 | m-1/m/m+1 | Boundary loop values |

**🔸 3. Concatenated Loops**

**✅ Definition:**

**Two or more loops**, one **after another**, not nested.

**✅ Example:**

cpp

CopyEdit

for (int i = 0; i < x; i++) {

doSomething(i);

}

for (int j = 0; j < y; j++) {

doSomethingElse(j);

}

**✅ Types:**

1. **Independent Concatenated Loops**  
   → Each loop runs on its own without dependency.

🔹 **Strategy**: Test **each loop separately** using **Simple Loop** approach.

1. **Dependent Concatenated Loops**  
   → One loop’s counter or result **affects** the other.

🔹 **Strategy**: Treat like **Nested Loops**. Apply combined testing.

**✅ Example:**

**🔸 Independent:**

cpp

CopyEdit

for (int i = 0; i < 3; i++) {}

for (int j = 0; j < 2; j++) {}

* Test each with: 0, 1, 2, n-1, n, n+1 iterations.

**🔸 Dependent:**

cpp

CopyEdit

for (int i = 0; i < x; i++) {}

for (int j = i; j < x + 5; j++) {} // j depends on i

* Must test **combinations** of i and j values like in **Nested Loop strategy**.

**🔸 4. Unstructured Loops**

**✅ Definition:**

Loops that do not follow structured constructs. Often use **goto, break, continue, recursion**, or **multiple entry/exit points**. These are **hard to test and maintain**.

**✅ Example:**

cpp

CopyEdit

start:

if (condition) {

process();

goto start;

}

OR

cpp

CopyEdit

while (true) {

if (exit\_condition) break;

}

**✅ Testing Strategy:**

* **Avoid these** wherever possible.
* Refactor into **structured loops** (for, while, do-while).
* If refactoring not possible:
  + **Trace each path** manually.
  + Ensure **entry, exit, and all branches** are tested.
  + Test for **infinite loop protection**.

**✅ Summary Table of Loop Types and Their Testing**

| **Loop Type** | **Description** | **Testing Approach** |
| --- | --- | --- |
| **Simple** | Single loop | 0, 1, 2, m, n-1, n, n+1 iterations |
| **Nested** | Loop inside loop | Start from innermost, apply simple strategy, expand outwards |
| **Concatenated (Independent)** | Loops one after another | Test each loop separately using simple strategy |
| **Concatenated (Dependent)** | Second loop depends on first | Treat as nested loops |
| **Unstructured** | Irregular control flow | Refactor; else test every possible path manually |

**✅ Path Coverage – Explained in Detail**

**🔸 What is a Path?**

A **path** in code refers to a **specific sequence of execution** that follows a set of **branches (if, while, switch, etc.) and decisions** from the start of a program/module to the end.

**🔸 Why Focus on Paths?**

In **white-box testing**, **branches and conditions** are more error-prone than straight-line code because:

* Bugs usually **depend on specific decision logic**
* Small changes in conditions can **break logic**
* Some paths may **never be tested** unless explicitly designed

**🔸 What is Path Coverage?**

Path coverage is a testing method that ensures **every possible path** in a control flow graph is **executed at least once** through test cases.

**✅ Goal:**

Test **all unique paths** through a program.

This helps in:

* Detecting logic errors
* Covering edge cases missed by block/branch testing
* Ensuring decisions are working correctly

**🔸 Example: Code with Branches**

cpp

CopyEdit

void check(int x) {

if (x > 0)

cout << "Positive";

else

cout << "Non-positive";

if (x % 2 == 0)

cout << " Even";

else

cout << " Odd";

}

**✅ Flow Graph:**

* **Node 1**: Entry
* **Node 2**: x > 0
* **Node 3**: Positive
* **Node 4**: Non-positive
* **Node 5**: x % 2 == 0
* **Node 6**: Even
* **Node 7**: Odd
* **Node 8**: Exit

**✅ Paths:**

| **Path #** | **Description** | **Path** |
| --- | --- | --- |
| P1 | x > 0 and x % 2 == 0 | 1 → 2 → 3 → 5 → 6 → 8 |
| P2 | x > 0 and x % 2 != 0 | 1 → 2 → 3 → 5 → 7 → 8 |
| P3 | x <= 0 and x % 2 == 0 | 1 → 2 → 4 → 5 → 6 → 8 |
| P4 | x <= 0 and x % 2 != 0 | 1 → 2 → 4 → 5 → 7 → 8 |

🔹 **Total paths = 4**  
🔹 Test cases must be created to execute **each path once**

**🔸 Independent Paths**

**✅ Definition:**

An **independent path** is a path that:

* **Introduces at least one new edge or node** in the control flow graph that no previous path included.
* Reflects a **unique decision sequence** in the logic.

🔹 **Important**: Even if the start and end are the same, if a condition changes, it's a new path.

**✅ Example:**

Given this snippet:

cpp

CopyEdit

if (A)

if (B)

doSomething();

You have 3 independent paths:

1. A=false
2. A=true, B=false
3. A=true, B=true

Each explores a **unique condition path**.

**✅ Summary Table:**

| **Concept** | **Explanation** |
| --- | --- |
| **Path** | A unique decision route through a program |
| **Path Coverage** | Ensure all execution paths are tested |
| **Independent Path** | A path with at least one **new edge** in control flow |
| **Why Important** | Conditions and branches are where most bugs hide |
| **Test Case Creation** | Each independent path = at least one test case |

**✅ Benefits of Path Coverage**

* Ensures **maximum decision testing**
* Uncovers **hidden logic bugs**
* Provides better **test effectiveness** than just statement or branch coverage

**✅ Cyclomatic Complexity**

**📌 Definition:**

Cyclomatic Complexity is a **quantitative metric** in software engineering that measures the **logical complexity** of a program. It indicates the number of **linearly independent paths** through the source code.

**📈 Why it's Important:**

* Measures **code complexity**
* Helps determine **test case design**
* More complexity → **more errors** likely
* Lower complexity → **easier to understand, test, and maintain**

**📐 How It's Calculated:**

**1. Formula from Graph Theory:**

V(G)=E−N+2P\boxed{V(G) = E - N + 2P}V(G)=E−N+2P​

Where:

* EEE = Number of **edges**
* NNN = Number of **nodes**
* PPP = Number of **connected components** (usually 1 for a single program)

**2. Alternative Formula:**

V(G)=b+1\boxed{V(G) = b + 1}V(G)=b+1​

Where:

* bbb = Number of **binary decisions** (e.g., if, while, for, etc.)

**📊 Example:**

Given a control flow graph with:

* E=16E = 16E=16, N=13N = 13N=13, P=1P = 1P=1

V(G)=16−13+2(1)=5V(G) = 16 - 13 + 2(1) = \boxed{5}V(G)=16−13+2(1)=5​

Or if we count decisions:

* b=4⇒V(G)=4+1=5b = 4 \Rightarrow V(G) = 4 + 1 = \boxed{5}b=4⇒V(G)=4+1=5​

**✅ Interpretation of Cyclomatic Complexity:**

| **V(G) Value** | **Interpretation** |
| --- | --- |
| 1–10 | Simple, low risk |
| 11–20 | Moderate complexity |
| 21–50 | High complexity, more testing needed |
| >50 | Very complex, **should be redesigned** |

**✅ Basis Path Testing**

**📌 Definition:**

A **white-box testing method** that uses Cyclomatic Complexity to:

* Identify a **basis set** of linearly independent paths
* Create **test cases** for each path

**✅ Steps in Basis Path Testing:**

1. **Construct Control Flow Graph** (CFG)
2. **Compute Cyclomatic Complexity** → Number of independent paths
3. **Determine Basis Paths**
4. **Create Test Cases** for each path
5. **Execute Test Cases**

**✅ Independent Path:**

A path that introduces **at least one new edge** not included in other paths.

**✅ McCabe’s Baseline Method:**

**📌 Purpose:**

Efficiently generate **independent paths** for testing.

**📌 Steps:**

1. Pick a **baseline path** (normal case)
2. Flip **first decision** in baseline to create a new path
3. Go back to baseline, flip **second decision**, and repeat
4. Continue until all decision points are flipped
5. Repeat process for other paths to get all basis paths

**📊 Example:**

If:

* E=19E = 19E=19
* N=14N = 14N=14
* P=1P = 1P=1

Then:

V(G)=19−14+2(1)=7V(G) = 19 - 14 + 2(1) = \boxed{7}V(G)=19−14+2(1)=7​

So we need **7 independent paths**.

Example of independent paths:

yaml

CopyEdit

Path 1: 1 - 2 - 3 - 5 - 7 - 9 - 11 - 13 - 14

Path 2: 1 - 3 - 4 - 14

Path 3: 1 - 3 - 5 - 6 - 14

Path 4: 1 - 3 - 5 - 7 - 8 - 14

Path 5: 1 - 3 - 5 - 7 - 9 - 10 - 14

Path 6: 1 - 3 - 5 - 7 - 9 - 11 - 12 - 14

Path 7: 1 - 3 - 5 - 7 - 9 - 11 - 13 - 14

**✅ Key Observations:**

* Cyclomatic Complexity gives the **upper bound** on the number of test cases required.
* Basis Path Testing ensures **full statement and branch coverage**.
* Multiple **valid sets** of basis paths may exist.
* Basis path testing is effective but **time-consuming** and requires **understanding of control flow**.

**✅ Applicability & Limitations:**

**✔ Applicability:**

* Useful for **unit testing**
* Helps detect **logic errors**
* Complements **code reviews**

**❌ Limitations:**

* Requires **code-level understanding**
* Can be **time-consuming**
* Doesn’t guarantee **absence of bugs**, only thorough coverage