

What is White Box Testing?

If we go by the definition, “White box testing” (also known as clear, glass box or structural testing) is a testing technique which evaluates the code and the internal structure of a program.

White box testing involves looking at the structure of the code. When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised.

Why we perform WBT?

To ensure:

- That all independent paths within a module have been exercised at least once.
- All logical decisions verified on their true and false values.
- All loops executed at their boundaries and within their operational bounds internal data structures validity.

To discover the following types of bugs:

- Logical error tend to creep into our work when we design and implement functions, conditions or controls that are out of the program
- The design errors due to difference between logical flow of the program and the actual implementation
- Typographical errors and syntax checking

Steps to Perform WBT

Step #1 – Understand the functionality of an application through its source code. Which means that a tester must be well versed with the programming language and the other tools as well techniques used to develop the software.

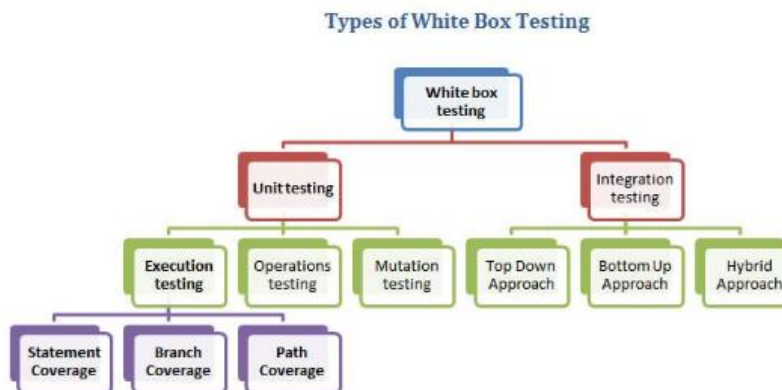
Step #2– Create the tests and execute them.

When we discuss the concept of testing, “coverage” is considered to be the most important factor. Here I will explain how to have maximum coverage from the context of White box testing.

Types and Techniques of White Box Testing

There are several types and different methods for each white box testing type.

See the below image for your reference.



3 Main White Box Testing Techniques:

1. Statement Coverage
2. Branch Coverage
3. Path Coverage

Note that the statement, branch or path coverage does not identify any bug or defect that needs to be fixed. It only identifies those lines of code which are either never executed or remains untouched. Based on this further testing can be focused on.

Let's understand these techniques one by one with a simple example.

#1) Statement coverage:

In a programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in a running mode.

Hence “*Statement Coverage*”, as the name itself suggests, it is the method of validating whether each and every line of the code is executed at least once.

#2) Branch Coverage:

“Branch” in a programming language is like the “IF statements”. An IF statement has two branches: **True and False.**

So in Branch coverage (also called Decision coverage), we validate whether each branch is executed at least once.

In case of an “IF statement”, there will be two test conditions:

- One to validate the true branch and,
- Other to validate the false branch.

Hence, in theory, Branch Coverage is a testing method which is when executed ensures that each and every branch from each decision point is executed.

#3) Path Coverage

Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

White Box Testing Example

Consider the below simple pseudocode:

INPUT A & B

$C = A + B$

IF $C > 100$

PRINT “ITS DONE”

For ***Statement Coverage*** – we would only need one test case to check all the lines of the code.

That means:

If I consider *TestCase_01* to be ($A=40$ and $B=70$), then all the lines of code will be executed.

Now the question arises:

1. Is that sufficient?
2. What if I consider my Test case as $A=33$ and $B=45$?

Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well.

Hence for maximum coverage, we need to consider “***Branch Coverage***”, which will evaluate the “FALSE” conditions.

In the real world, you may add appropriate statements when the condition fails.

So now the pseudocode becomes:

INPUT A & B

C = A + B

IF C>100

PRINT "ITS DONE"

ELSE

PRINT "ITS PENDING"

Since Statement coverage is not sufficient to test the entire pseudo code, we would require Branch coverage to ensure maximum coverage.

So for Branch coverage, we would require two test cases to complete the testing of this pseudo code.

TestCase_01: A=33, B=45

TestCase_02: A=25, B=30

With this, we can see that each and every line of the code is executed at least once.

Here are the Conclusions that are derived so far:

- Branch Coverage ensures more coverage than Statement coverage.
- Branch coverage is more powerful than Statement coverage.
- 100% Branch coverage itself means 100% statement coverage.
- But 100 % statement coverage does not guarantee 100% branch coverage.

Now let's move on to **Path Coverage**:

As said earlier, Path coverage is used to test the complex code snippets, which basically involve loop statements or combination of loops and decision statements.

Consider this pseudocode:

INPUT A & B

C = A + B

IF C>100

PRINT "ITS DONE"

END IF

IF A>50

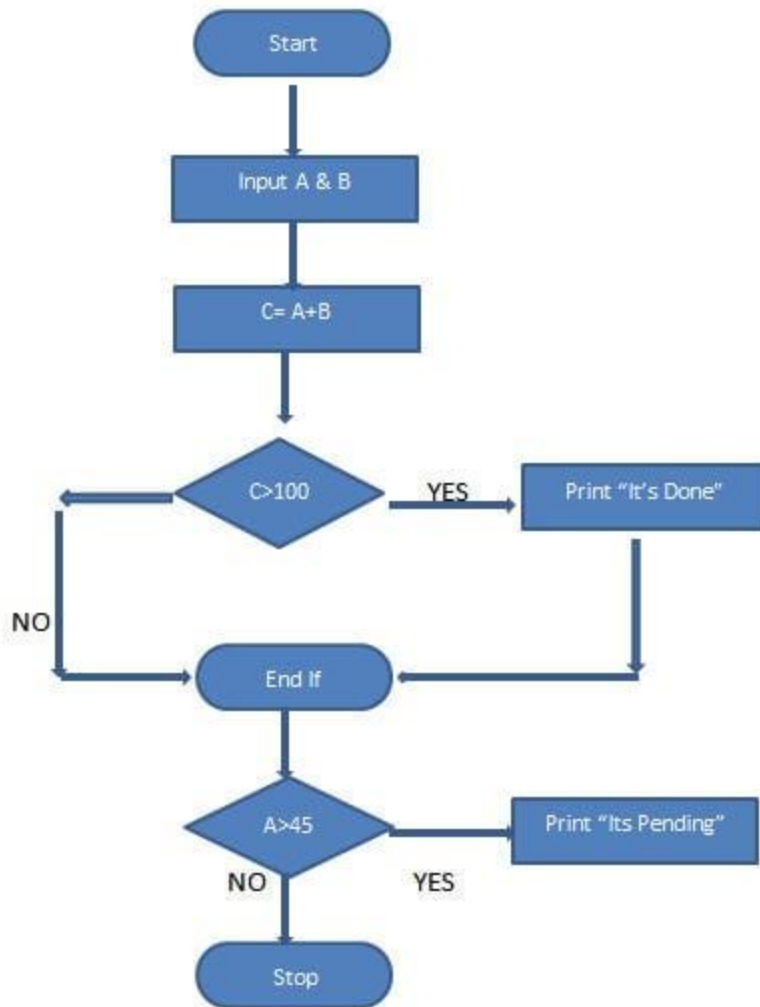
PRINT "ITS PENDING"

END IF

Now to ensure maximum coverage, we would require 4 test cases.

How? Simply – there are 2 decision statements, so for each decision statement, we would need two branches to test. One for true and the other for the false condition. So for 2 decision statements, we would require 2 test cases to test the true side and 2 test cases to test the false side, which makes a total of 4 test cases.

To simplify these let's consider below flowchart of the pseudo code we have:



Path Coverage

*Further Reading => **How To Make A Flowchart In MS Word***

In order to have the full coverage, we would need following test cases:

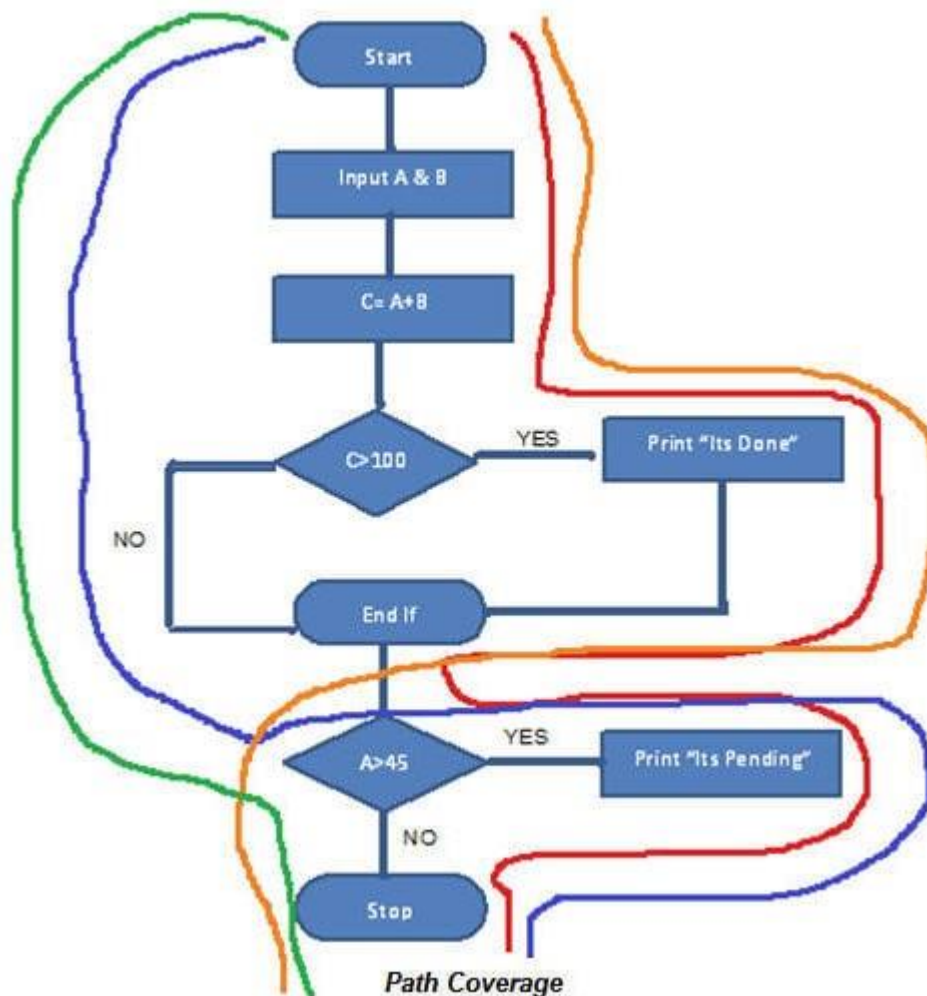
TestCase_01: A=50, B=60

TestCase_02: A=55, B=40

TestCase_03: A=40, B=65

TestCase_04: A=30, B=30

So the path covered will be:



Red Line – TestCase_01 = (A=50, B=60)

Blue Line = TestCase_02 = (A=55, B=40)

Orange Line = TestCase_03 = (A=40, B=65)

Green Line = TestCase_04 = (A=30, B=30)

What Is Code Coverage?

This is an important unit testing metric. It comes handy in knowing the effectiveness of unit tests. It is a measure that indicates what percentage of source code would get executed while testing.

In simple terms, **the extent to which the source code of a software program or an application will get executed during testing is what is termed as Code Coverage.**

If the tests execute the entire piece of code including all branches, conditions, or loops, then we would say that there is complete coverage of all the possible scenarios and thus the Code Coverage is 100%. To understand this even better, let's take up an example.

Given below is a simple code that is used to add two numbers and display the result depending on the value of the result.

Input a, b

Let $c = a + b$

If $c < 10$, print c

Else, print 'Sorry'

The above program takes in two inputs i.e. 'a' & 'b'. The sum of both is stored in variable c . If the value of c is less than 10, then the value of 'c' is printed else 'Sorry' is printed.

Now, if we have some tests to validate the above program with the values of a & b such that the sum is always less than 10, then the else part of the code never gets executed. In such a scenario, we would say that the coverage is not complete.

This was just a small example to clarify the meaning of Code Coverage. As we explore more, you will gain better clarity on it.

Why We Need Code Coverage?

Various reasons make Code Coverage essential and some of those are listed below:

- It helps to ascertain that the software has lesser bugs when compared to the software that does not have a good Code Coverage.
- By aiding in improving the code quality, it indirectly helps in delivering a better 'quality' software.
- It is a measure that can be used to know the test effectiveness (effectiveness of the unit tests that are written to test the code).
- Helps to identify those parts of the source code that would go untested.
- It helps to determine if the current testing (unit testing) is sufficient or not and if some more tests are needed in place as well.

What Is Mutation Testing

Mutation testing is a fault-based testing technique where variations of a software program are subjected to the test dataset. This is done to determine the effectiveness of the test set in isolating the deviations.

Mutation Testing (MT) goes a long way, back to the 70s where it was first proposed as a school project. It was written off as it was very resource-intensive. However, as humans continued to develop more advanced computers, it slowly made a comeback and is now one of the most popular testing techniques.

Mutation testing definition

MT is also known as **fault-based testing, program mutation, error-based testing, or mutation analysis.**

As the name suggests, mutation testing is a software testing type that is based on changes or mutations. Miniscule changes are introduced into the source code to check whether the defined test cases can detect errors in the code.

The ideal case is that none of the test cases should pass. If the test passes, then it means that there is an error in the code. We say that the mutant (the modified version of our code) lived. If the test fails, then there is no error in the code, and the mutant was killed. Our goal is to kill all mutants.

Mutation testing also helps to test the quality of the defined test cases or the test suites with a bid to write more effective test cases. The more mutants we can kill, the higher the quality of our tests.

Mutation Testing Concepts

Before we discuss mutation testing further, let us explore some concepts that we will come across.

#1) Mutants: It is simply the mutated version of the source code. It is the code that contains minute changes. When the test data is run through the mutant, it should ideally give us different results from the original source code.

Mutants are also called **mutant programs**.

There are different types of mutants. These are as follows:

- **Survived Mutants:** As we have mentioned, these are the mutants that are still alive after running test data through the original and mutated variants of the source code. These must be killed. They are also known as live mutants.
- **Killed Mutants:** These are mutants that are killed after mutation testing. We get these when we get different results from the original and mutated versions of the source code.
- **Equivalent Mutants:** These are closely related to live mutants, in that, they are 'alive' even after running test data through them. What differentiates them from others is that they have the same meaning as the original source code, even though they may have different syntax.

#2) Mutators/mutation operators: These are what makes mutations possible, they are on the 'driver's seat'. They basically define the kind of alteration or change to make to the source code to have a mutant version. They can be referred to as **faults or mutation rules**.

#3) Mutation score: This is a score based on the number of mutants.

It is calculated using the below formula:

$$\text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants (survived and killed)}} * 100\%$$

Note that, equivalent mutants are not considered when calculating the mutation score. Mutation score is also known as **mutation adequacy**. Our aim should be to achieve a high mutation score.

Data Flow Testing

Data flow testing can be defined as a software testing technique that is based on data values and their usage in a program. It verifies that data values have been properly used and that they generate the correct results. Data flow testing helps to trace the dependencies between data values on a particular execution path.

Data Flow Anomalies

Data flow anomalies are simply errors in a software program. They are classified into types 1, 2, and 3 respectively.

Let's delve into them below:

Type 1: A variable is defined and a value is assigned to it twice.

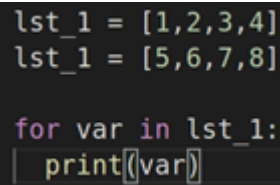
Example code: Python

```
lst_1 = [1,2,3,4]
```

```
lst_1 = [5,6,7,8]
```

```
for var in lst_1:
```

```
    print(var)
```



```
lst_1 = [1,2,3,4]
lst_1 = [5,6,7,8]

for var in lst_1:
    print(var)
```

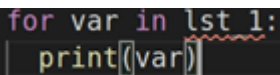
lst_1 is defined, and two different values are assigned to it. The first value is simply ignored. Type 1 anomalies do not cause the program to fail.

Type 2: The value of a variable is used or referenced before it is defined.

Example code: Python

```
for var in lst_1:
```

```
    print(var)
```



```
for var in lst_1:
    print(var)
```

The loop above has no values to iterate over. Type 2 anomalies cause the program to fail.

Type 3: A data value is generated, but is it never used.

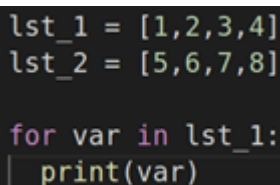
Example code: Python

```
lst_1 = [1,2,3,4]
```

```
lst_2 = [5,6,7,8]
```

```
for var in lst_1:
```

```
    print(var)
```



```
lst_1 = [1,2,3,4]
lst_2 = [5,6,7,8]

for var in lst_1:
    print(var)
```

The variable **lst_2** has not been referenced. Type 3 anomalies may not cause program failure.

Data Flow Testing Process

To define the dependencies between data values, we need to define the different paths that can be followed in a program. To effectively do this, we need to borrow from another structural testing type known as control-flow testing.

Step #1) Draw a control flow graph

We need to draw a control flow graph, which is a graphical representation of the paths that we could follow in our program.

