Chapter 3: Testing Strategies

Static Techniques

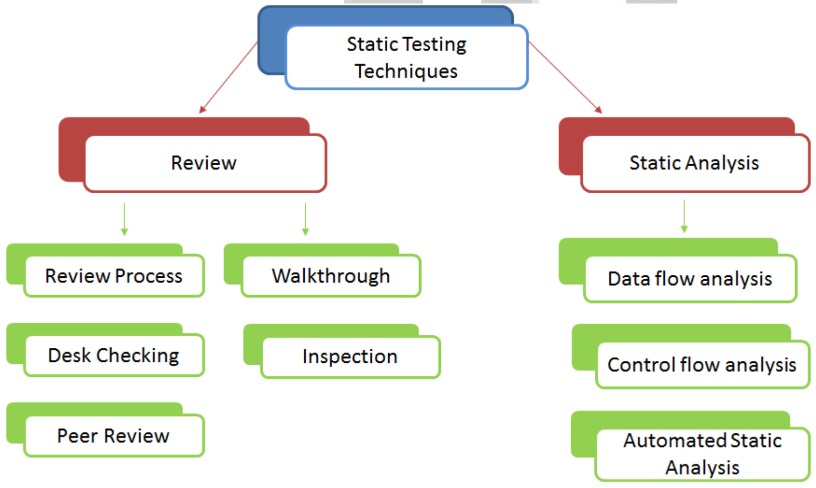
STATIC TESTING is a software testing technique by which we can check the defects in software without actually executing it.

Static testing is done to avoid errors at an early stage of development as it is easier to find sources of failures then failures themselves.

Static testing helps to find errors that may not be found by Dynamic Testing.

It can be done manually or by a set of tools. This type of testing checks the code, requirement documents and design documents and puts review comments on the work document.

Static testing can be done on work documents like requirement specifications, design documents, source code, test plans, test scripts and test cases, web page content.



Peer Review

Peer reviews are the informal reviews where team members conduct reviews amongst themselves. They are also known as buddy reviews.

Sometimes called buddy reviews, this method is really more of an “I’ll show you mine if you show me yours” type discussion.

Work products, such as requirements, design, and code, are reviewed by peers to find defects at an early stage.

Peer reviews are often held with just the programmer who wrote the code and one or two other programmers or testers acting as reviewers.

That small group simply reviews the code together and looks for problems and oversights. To assure that the review is highly effective (and does not turn into a coffee break) all the participants need to make sure that the four key elements of a formal review are in place: Look for problems, follow rules, prepare for the review, and write a report.

Because peer reviews are informal, these elements are often scaled back. Still, just getting together to discuss the code can find bugs.

Technical review

It is less formal review

It is led by the trained moderator but can also be led by a technical expert

It is often performed as a peer review without management participation

Defects are found by the experts (such as architects, designers, key users) who focus on the content of the document.

In practice, technical reviews vary from quite informal to very formal

A technical review is a discussion meeting that focuses on technical content of a document. it is led by a trained moderator, but also can be led by a technical expert.

Compared to inspections, technical reviews are less formal and there is little or no focus on defect identification on the basis of referenced documents.

The experts that are needed to be present for a technical review can be architects, chief designers and key users. It is often performed as a peer review without management participation.

The specific goals of a technical review are:

• evaluate the value of technical concepts and alternatives in the product and project environment.

• establish consistency in the use and representation of technical concepts.

• ensuring at an early stage, that technical concepts are used correctly;

• inform participants of the technical content of the document.

Walkthroughs

Walkthroughs are the next step up in formality from peer reviews.

The objective of the walkthrough is to ensure a high quality and also to find problems and not to correct them.

It is an informal meeting but with the purpose.

In a walkthrough, an author means the programmer who wrote the code, formally presents (walks through) it to a small group of five or so other programmers and testers.

The reviewers should receive copies of the software in advance of the review so they can examine it and write comments and questions that they want to ask at the review. Having at least one senior programmer as a reviewer is very important.

The author reads through the code line by line, or function by function, explaining what the code does and why.

The reviewers listen and question anything that looks suspicious. Because of the larger number of participants involved in a walkthrough compared to a peer review, it's much more important for them to prepare for the review and to follow the rules.

It's also very important that after the review the presenter write a report telling what was found and how he plans to address any bugs discovered.

Author:

- Selects the participants in a walkthrough. No specific roles are assigned.

- Selects a walkthrough review approach for the product being reviewed.

- Select review participants, obtain their agreement to participate, and schedule a walkthrough meeting.

- Distribute work product to reviewers prior to the meeting.

- States his objectives for the review.

- - Modifies/refines the work product.

- Makes appropriate changes in the work product based on reviewer's comments and suggestions.

Reviewers:

- Presents comments, possible defects, and improvement suggestions to the author.

Inspection:

It is a highly structured step-step review of the deliverables produced by each phase of the Systems Development Life Cycle in order to identify potential defects.

An inspection is a formal meeting, more formalized than a walk-through and typically consists of 3-10 people including a moderator, reader (the author of whatever is being reviewed) and a recorder (to make notes in the document).

The subject of the inspection is typically a document, such as a requirements document or a test plan.

The purpose of an inspection is to find problems and see what is missing, not to fix anything. The result of the meeting should be documented in a written report. Attendees should prepare for this type of meeting by reading through the document, before the meeting starts; most problems are found during this preparation.

Preparation for inspections is difficult, but is one of the most cost-effective methods of ensuring quality, since bug prevention is more cost effective than bug detection.

An inspection is one of the most common sorts of review practices found in software projects.

The goal of the inspection is for all of the inspectors to reach consensus on a work product and approve it for use in the project.

Commonly inspected work products include software requirements specifications and test plans. In an inspection, a work product is selected for review and a team is gathered for an inspection meeting to review the work product. A moderator is chosen to moderate the meeting.

Each inspector prepares for the meeting by reading the work product and noting each defect. The goal of the inspection is to identify defects. In an inspection, a defect is any part of the work product that will keep an inspector from approving it.

For example, if the team is inspecting a software requirements specification, each defect will be text in the document which an inspector disagrees with.

The process

The inspection process was developed by Michael Fagan in the mid-1970s and it has later been extended and modified.

The process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process. The entry criteria might be a checklist including items such as "The document has been spell-checked".

The stages in the inspections process are: Planning, Overview meeting, Preparation, Inspection meeting, Rework and Follow-up. The Preparation, Inspection meeting and Rework stages might be iterated.

* Planning: The inspection is planned by the moderator.

* Overview meeting: The author describes the background of the work product.

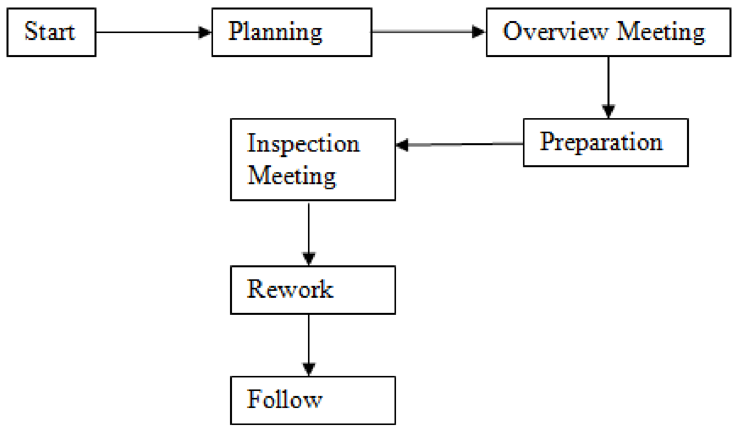
* Preparation: Each inspector examines the work product to identify possible defects.

* Inspection meeting: During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.

* Rework: The author makes changes to the work product according to the action plans from the inspection meeting.

* Follow-up: The changes by the author are checked to make sure everything is correct.

The process is ended by the moderator when it satisfies some predefined exit criteria.



Formal Inspection Roles:

1. Inspection Moderator:

- In software engineering (inspection), the individual who is trained to coordinate, lead, and control the inspection process and who is assigned to monitor the inspection.

- It is recommended that the moderator not be a member of the project team.

- The moderator ensures that the participants are properly prepared and that the inspection is efficiently and thoroughly conducted.

- The moderator schedules the inspection, distributes the material being inspected, facilitates the inspection process, and ensures that the errors in the material are recorded and corrected.

- The moderator is responsible for recording inspection data, making sure that the actions resulting from the inspection are completed, and for conducting re-inspections where appropriate.

- Ensures the author completes the follow-up tasks.

- Select inspectors and assign the roles of reader and recorder.

- Estimates inspection preparation time and schedule the inspection meeting. He sends inspection meeting notices to participants.

2. Reader:

- Responsible for setting the pace of the inspection.

- He/she has thorough familiarity with the material to be inspected.

- He is not the moderator or author.

3. Recorder:

- Responsible for listing defects and summarizing the inspection results.

- Records every defect found.

- May also be the moderator, but cannot be the reader or the author.

4. Author:

- He is the originator of the product being inspected.

- Initiates the inspection process by informing the moderator.

- He may also act as an inspector during the inspection meeting.

- Determines when the product is ready for the inspection.

- Assists the moderator in selecting the participants for the inspection team.

- Corrects the defects and presents finished rework to the moderator.

5. Inspector:

- The inspectors should be trained staff who can effectively contribute to meeting objectives of the inspection.

- The moderator, reader or recorder may also be inspector.

- He must prepare for the inspection by carefully reviewing and understating the material.

- Maintains the objectivity towards the product.

- Review the product.

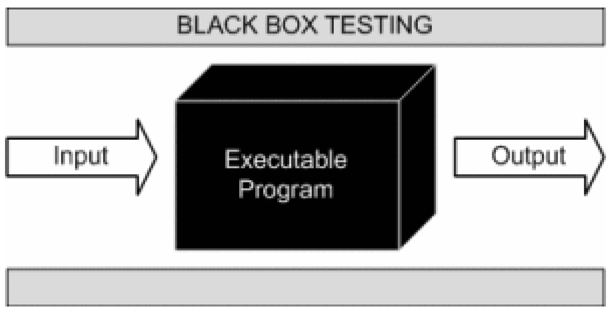
- Record all preparation time.

- Presents potential defects and problems encountered before and during the inspection meeting.

Black Box Testing Techniques

It is also known as Behavioral Testing, is software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.

Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications.



It focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box methods.

Black box testing is defined as a testing technique in which functionality of the Application Under Test (AUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on software requirements and specifications.

In Black Box Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing

Definition by ISTQB

- Black box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

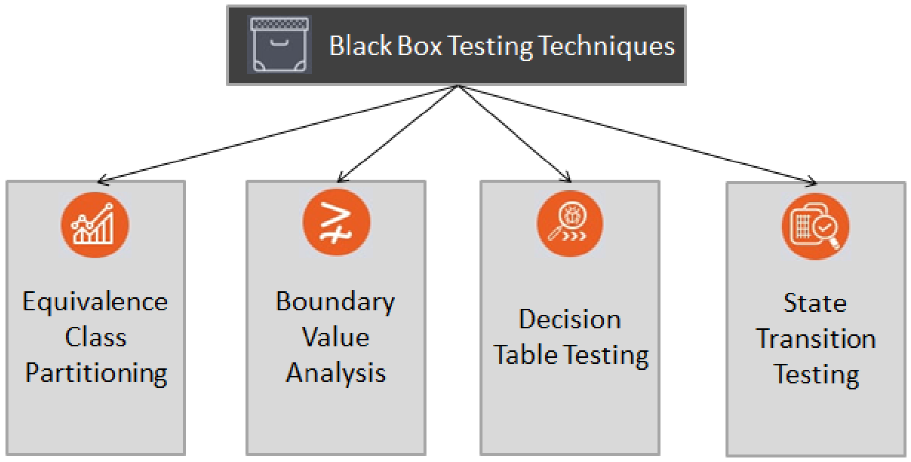
Techniques:

We know that exhaustive testing is not possible. So we need techniques to identify test cases with the most likelihood of finding a defect out of the possible many.

There are many test case designing techniques available.

In order to systematically test a set of functions, it is necessary to design test cases. Testers can create test cases from the requirement specification document using the various Black Box Testing techniques.

ISTQB Definition: Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

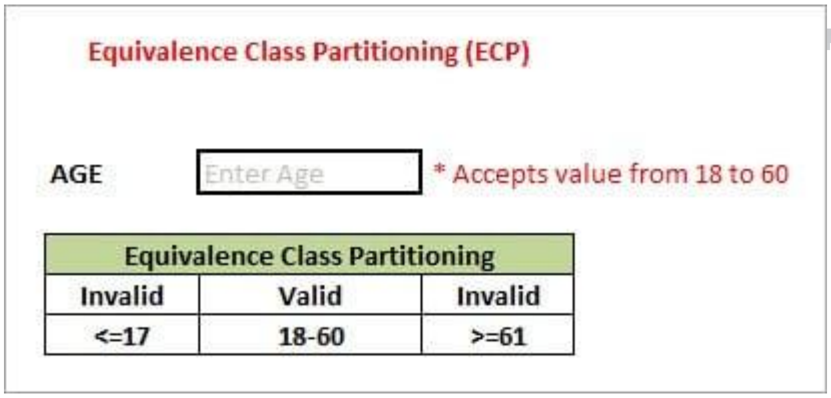


1. Equivalence Class Partitioning:

This technique is also known as Equivalence Class Partitioning (ECP). In this technique, input values to the system or application are divided into different classes or groups based on its similarity in the outcome.

Hence, instead of using each and every input value we can now use any one value from the group/class to test the outcome. In this way, we can maintain the test coverage while we can reduce a lot of rework and most importantly the time spent.

For Example:


Equivalence Class Partitioning (ECP)

AGE Enter Age * Accepts value from 18 to 60

Equivalence Class Partitioning		
Invalid	Valid	Invalid
<=17	18-60	>=61

As present in the above image, an "AGE" text field accepts only the numbers from 18 to 60. There will be three sets of classes or groups.

Two invalid classes will be:

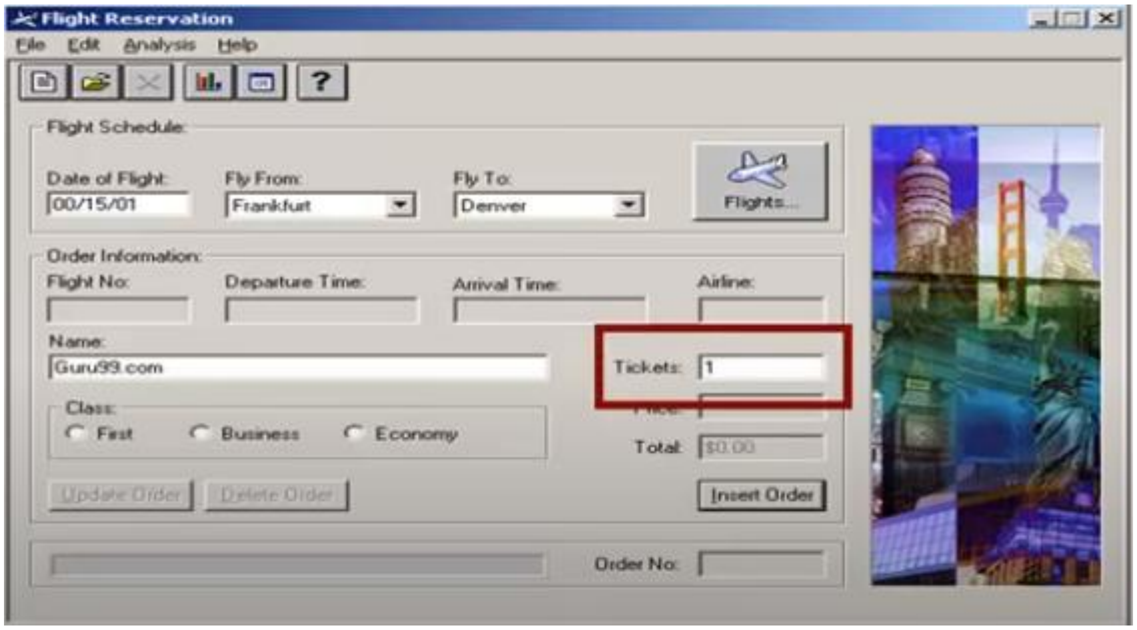
a) Less than or equal to 17.

b) Greater than or equal to 61.

One valid class will be anything between 18 and 60

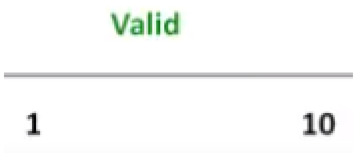
We have thus reduced the test cases to only 3 test cases based on the formed classes thereby covering all the possibilities. So, testing with anyone value from each set of the class is sufficient to test the above scenario.

Another Example:

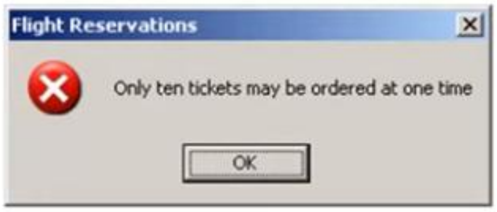


Date of Flight: 00/15/01
Fly From: Frankfurt
Fly To: Denver
Name: Guru99.com
Tickets: 1
Total: $0.00

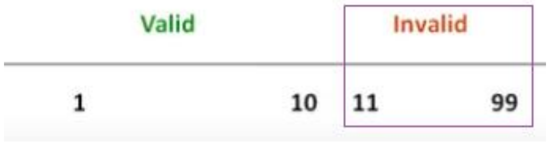
Valid values are only from 1 to 10



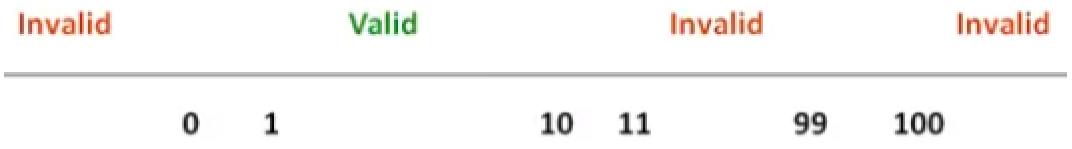
If user tries to enter values between 11 and 99, error message occurs.



Only ten tickets may be ordered at one time



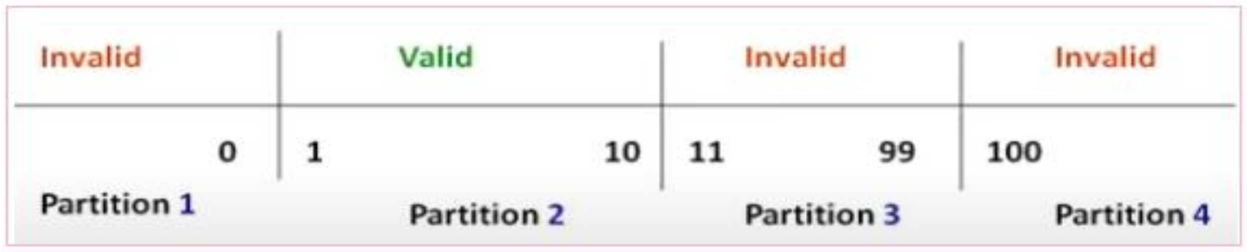
Even all values below 1 and above 99 are also invalid



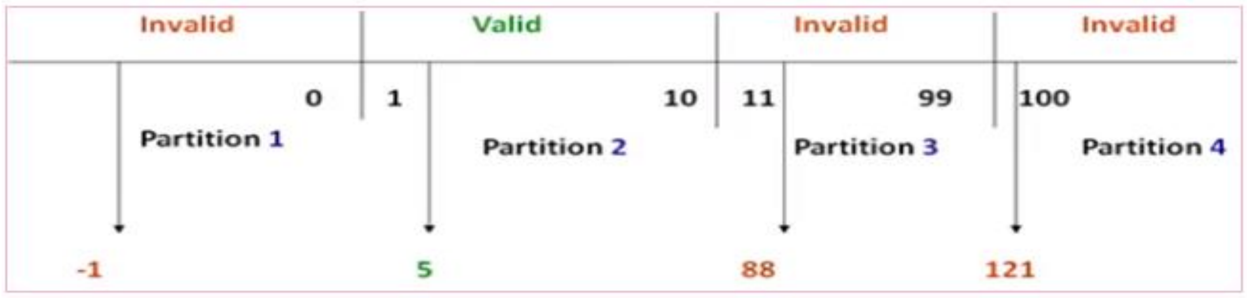
We cannot test all the possible values, because if done, number of test cases will be more than 100

We divide the possible values of tickets into groups or sets where the system behavior can be considered the same.

These sets are called **Equivalence Partitions or Equivalence Class**



Then we pick **only one value** from each partitioning for testing.



The hypothesis behind this technique is that if **one condition/ value in a partition pass**, **all others will also pass.**

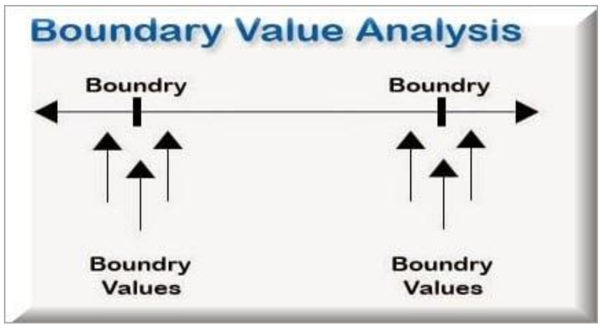
Likewise, if **one condition/ value in a partition fail, all other conditions in that partition will fail.**

2. Boundary Value Analysis:

From the name itself, we can understand that in this technique we focus on the values at boundaries as it is found that many applications have a high number of issues on the boundaries.

Boundary means the values near the limit where the behavior of the system changes. In boundary value analysis both the valid inputs and invalid inputs are being tested to verify the issues.

For Example:



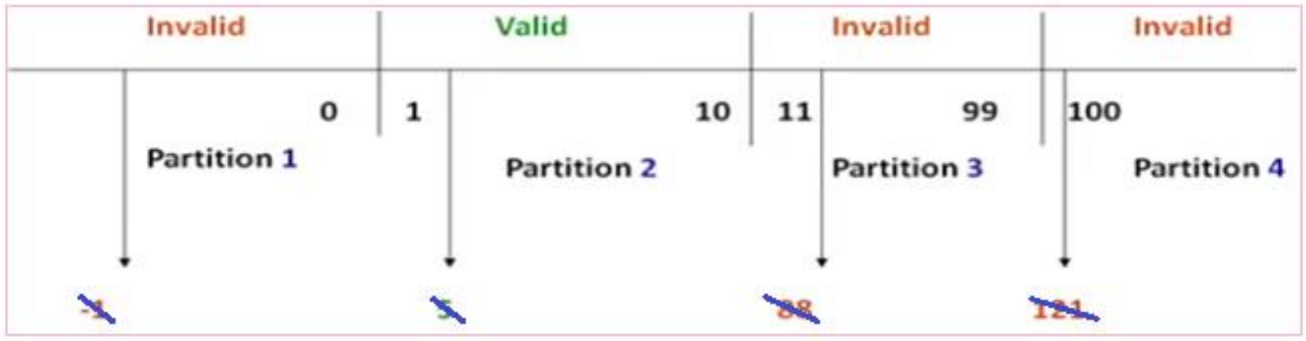
If we want to test a field where values from 1 to 100 should be accepted then we choose the boundary values: 1-1, 1, 1+1, 100-1, 100, and 100+1. Instead of using all the values from 1 to 100, we just use 0, 1, 2, 99, 100, and 101.

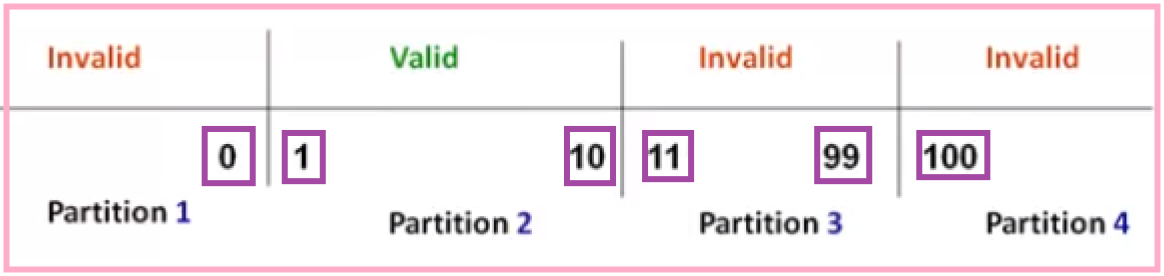
So in this, we test boundaries between partitions. It is also called range checking.

Equivalence Class Partitioning and Boundary Value Analysis are closely related and are generally used together at all levels of testing.

Consider earlier example of flight reservation system

Instead of checking one value from each partition, we will check the values at the boundaries.





3. Decision Table Testing:

Whenever there are logical conditions or decision-making steps then this technique is to be used.

It's a good way to deal with combinations of inputs which produce different results.

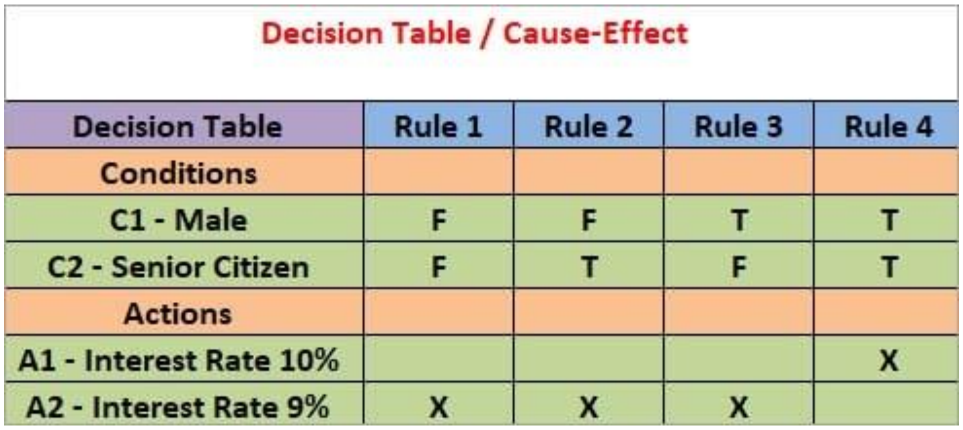
These can be like if a particular condition is not satisfied then Action A should be performed, else Action B is to be performed.

The tester needs to identify the input and actions which are to be performed based on the conditions.

Then a tester will identify two outputs (action1 and action2) for two conditions (True and False). A decision table is created based on these.

For Example:

Take an example of XYZ bank that provides interest rate for the Male senior citizen as 10% and for the rest of the people 9%.



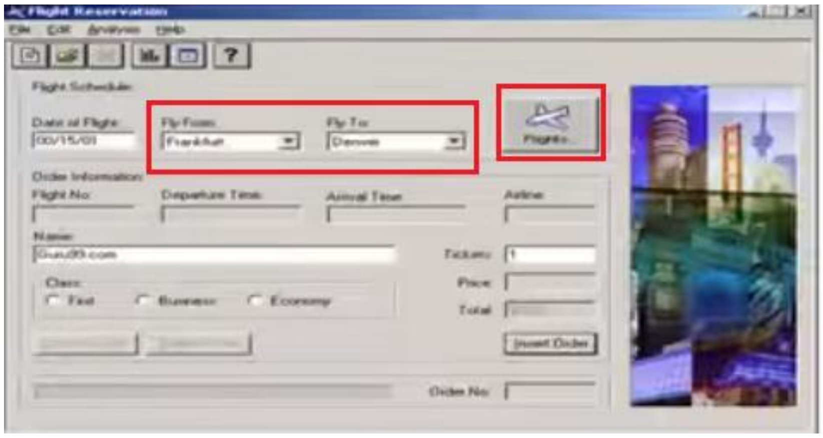
Decision Table / Cause-Effect

Decision Table	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
C1 - Male	F	F	T	T
C2 - Senior Citizen	F	T	F	T
Actions				
A1 - Interest Rate 10%				X
A2 - Interest Rate 9%	X	X	X	

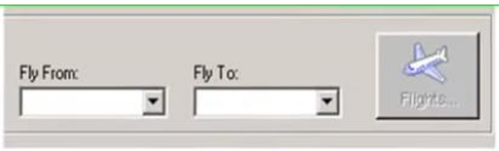
In this example condition, C1 has two values as true and false, condition C2 also has two values as true and false. The number of total possible combinations would then be four. This way we can derive test cases using a decision table.

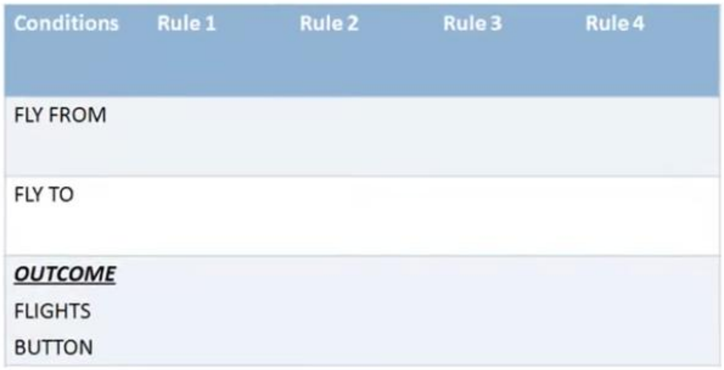
Possible combinations = 2^ n where n= no. of inputs

Consider earlier example of flight reservation system

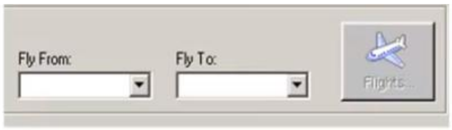


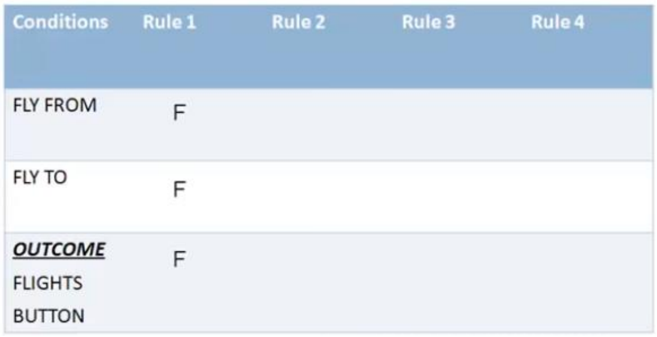
Date of Flight: 00/15/01
Fly From: Frankfurt
Fly To: Denver
Name: Guru99.com
Tickets: 1



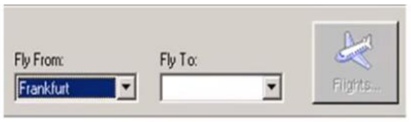


Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM				
FLY TO				
OUTCOME FLIGHTS BUTTON				

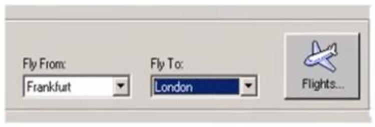




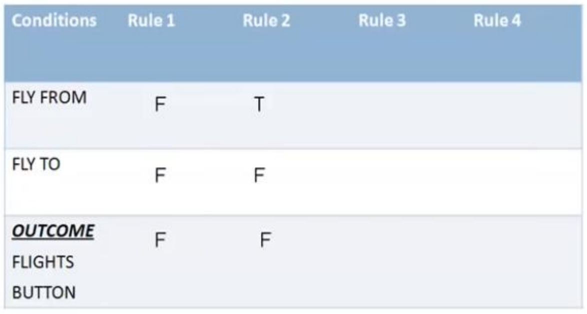
Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F			
FLY TO	F			
OUTCOME FLIGHTS BUTTON	F			


Fly From: Frankfurt

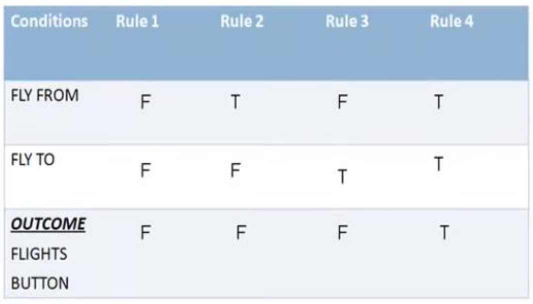
Fly To:


Fly From: Frankfurt

Fly To: London

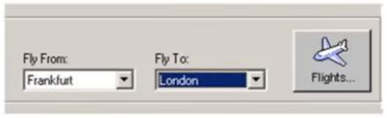


Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	T		
FLY TO	F	F		
OUTCOME FLIGHTS BUTTON	F	F		

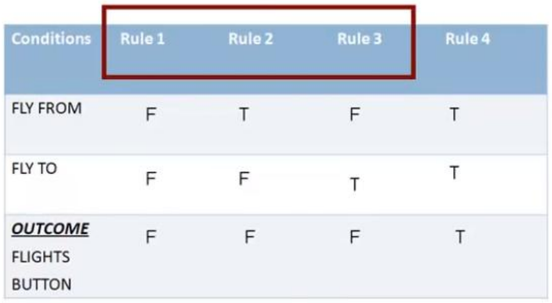


Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	T	F	T
FLY TO	F	F	T	T
OUTCOME FLIGHTS BUTTON	F	F	F	T

So output for rule 1, 2 and 3 remains same i.e. flight button remains disabled. So we can select any one of them and rule 4 for testing.


Fly From: Frankfurt

Fly To: London



Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	T	F	T
FLY TO	F	F	T	T
OUTCOME FLIGHTS BUTTON	F	F	F	T

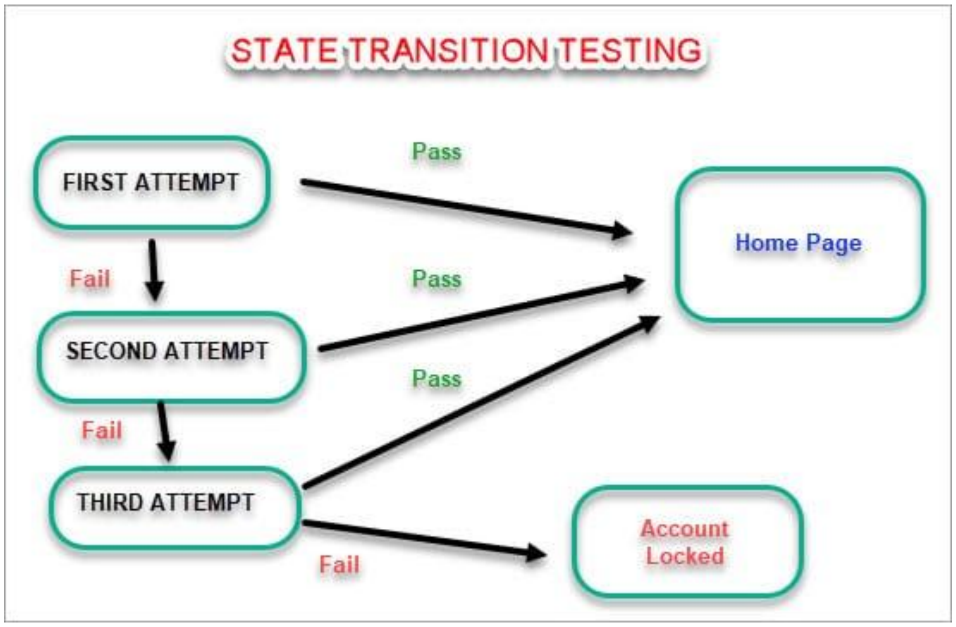
4. State Transition Testing:

State Transition Testing is a technique that is used to test the different states of the system under test. The state of the system changes depending upon the conditions or events. The events trigger states which become scenarios and a tester needs to test them.

A systematic state transition diagram gives a clear view of the state changes but it is effective for simpler applications. More complex projects may lead to more complex transition diagrams thus making it less effective.

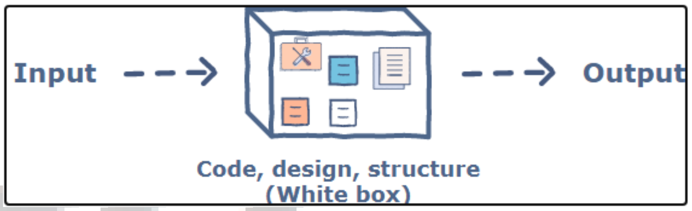
This technique is helpful where you need to test different system transitions.

For Example:



In this technique, the test case tries to test the system under different states. This state can change depending upon different conditions or events. When a particular event occurs then these scenarios can be tested.

White Box Testing Techniques



White-box testing is used to test software's architecture, design, and programming techniques. It mainly focuses on verifying the flow of inputs and outputs within the application.

It is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

White-box testing, also known as open box and glass box testing is usually performed by developers.

ISTQB Definition

White-box testing: Testing based on an analysis of the internal structure of the component or system.

In this technique, internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.

What do you verify in White Box Testing?

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output

- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

Levels

White Box Testing method is applicable to the following levels of software testing:

Unit Testing: For testing paths within a unit.

Integration Testing: For testing paths between units.

System Testing: For testing paths between subsystems.

However, it is mainly applied to Unit Testing.

Advantages

Code optimization by finding hidden errors

White box tests cases can be easily automated.

Testing is more thorough as all code paths are usually covered.

Testing can start early in SDLC even if GUI is not available.

Disadvantages

White box testing can be quite complex and expensive.

The white box testing by developers is not detailed can lead to production errors.

White box testing requires professional resources, with a detailed understanding of programming and implementation.

White-box testing is time-consuming, bigger programming applications take the time to test fully.

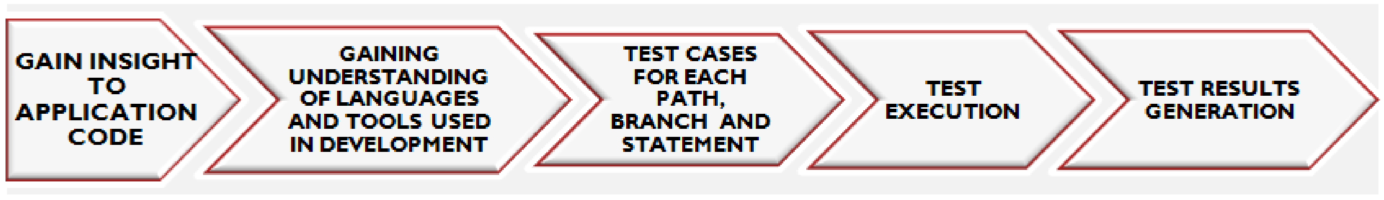
Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.

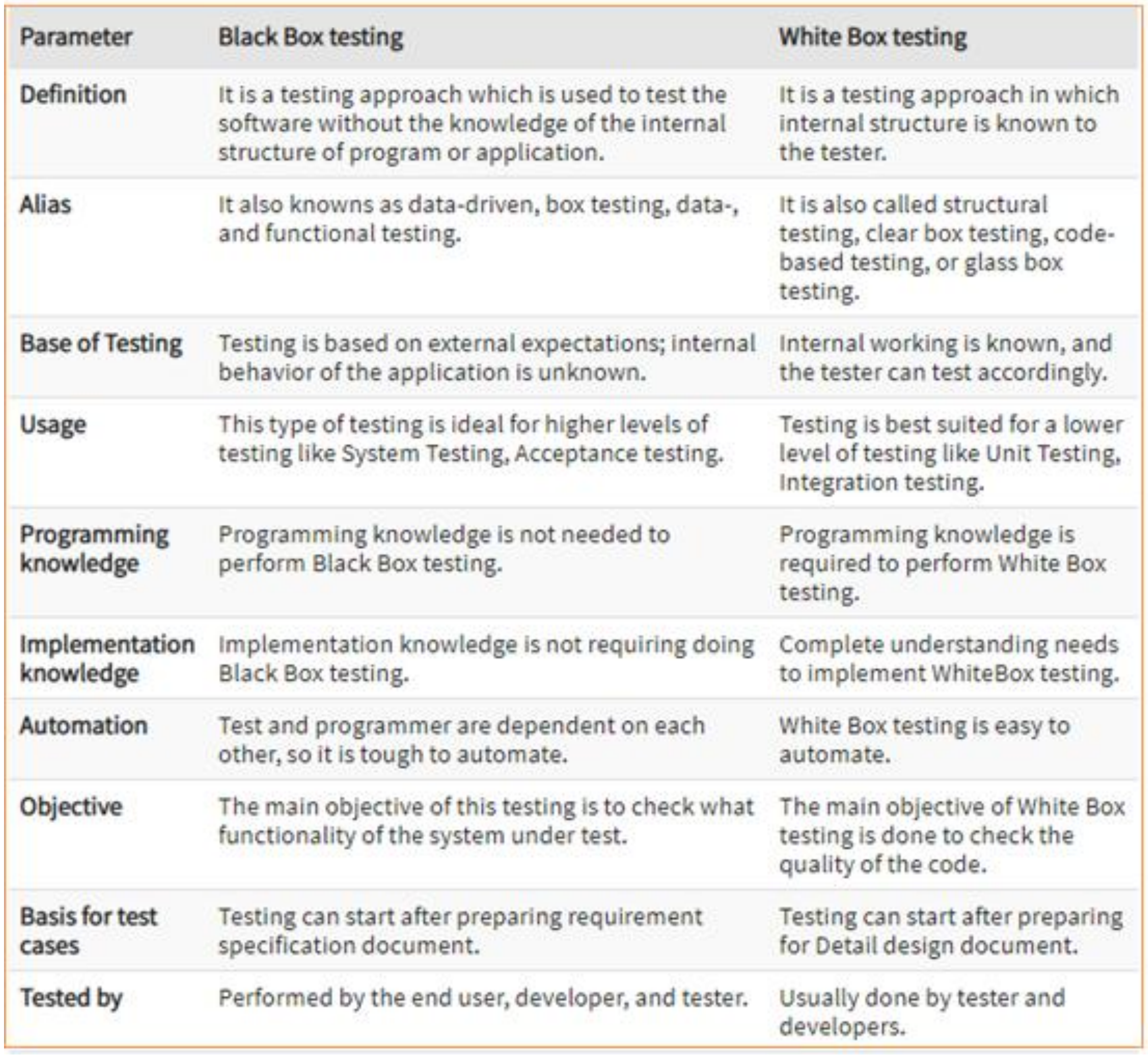
Test script maintenance can be a burden if the implementation changes too frequently.

Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

White Box Testing Process

White-box testing follows a designated testing process, wherein each aspect of the software is tested thoroughly by the team or the test manager, to ensure its accuracy as well as quality. The process of white box testing includes five important steps, which are:


GAIN INSIGHT TO APPLICATION CODE
GAINING UNDERSTANDING OF LANGUAGES AND TOOLS USED IN DEVELOPMENT
TEST CASES FOR EACH PATH, BRANCH AND STATEMENT
TEST EXECUTION
TEST RESULTS GENERATION



Parameter	Black Box testing	White Box testing
Definition	It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.
Alias	It also knowns as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
Implementation knowledge	Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
Objective	The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.

4.2.1 Statement Coverage Testing

Statement coverage is one of the widely used software testing. It comes under white box testing.

Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once.

It is used to calculate the total number of executed statements in the source code out of total statements present in the source code.

Statement coverage derives scenario of test cases under the white box testing process which is based upon the structure of the code.

Generally in any software, if we look at the source code, there will be a wide variety of elements like operators, functions, looping, exceptional handlers, etc. Based on the input to the program, some of the code statements may not be executed.

The main purpose of Statement Coverage is to cover all the possible paths, lines and statements in source code.

Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

What is covered by Statement Coverage?

Unused Statements

Dead Code

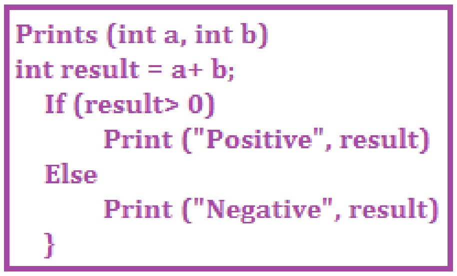
Unused Branches

Missing Statements

Let's understand this with an example, how to calculate statement coverage.

Scenario to calculate Statement Coverage for given source code. Here we are taking two different scenarios to check the percentage of statement coverage for each scenario.

Source Code:

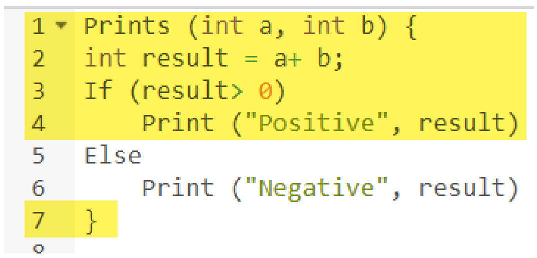


```
Prints (int a, int b)
int result = a+ b;
   If (result> 0)
         Print ("Positive", result)
   Else
         Print ("Negative", result)
   }
```

```
int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }
```

Scenario 1:

If A = 3, B = 9

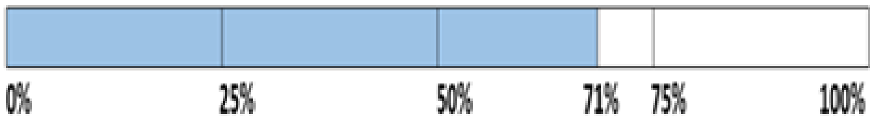


```
Prints (int a, int b) {
int result = a+ b;
If (result> 0)
    Print ("Positive", result)
Else
    Print ("Negative", result)
}
```

The statements marked in yellow color are those which are executed as per the scenario

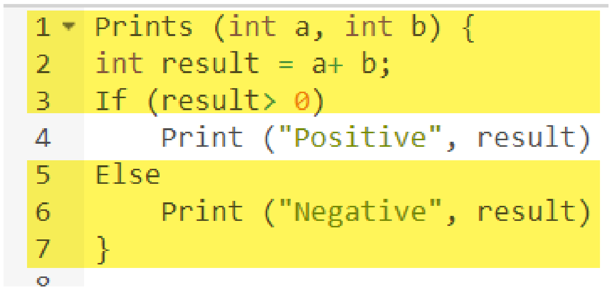
Number of executed statements = 5, Total number of statements = 7

Statement Coverage: 5/7 = 71%



Scenario 2:

If A = -3, B = -9



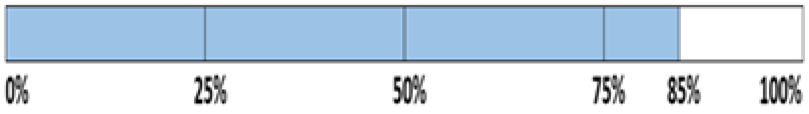
```
Prints (int a, int b) {
int result = a+ b;
If (result> 0)
    Print ("Positive", result)
Else
    Print ("Negative", result)
}
```

The statements marked in yellow color are those which are executed as per the scenario.

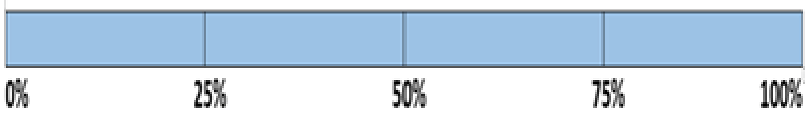
Number of executed statements = 6

Total number of statements = 7

Statement Coverage: 6/7 = 85%



But overall if you see, all the statements are being covered by 2nd scenarios considered. So we can conclude that overall statement coverage is 100%.



2. Decision Coverage

Decision Coverage is a white box testing technique which reports the true or false outcomes of each Boolean expression of the source code.

Whenever there is a possibility of two or more outcomes from the statements like **do while statement, if statement and case statement (Control flow statements)**, it is considered as decision point because there are two outcomes either true or false.

The goal of decision coverage testing is to cover and validate all the accessible source code by checking and ensuring that each branch of every possible decision point is executed at least once.

Generally, a decision point has two decision values one is true, and another is false that's why most of the times the total number of outcomes is two. The percent of decision coverage can be found by dividing the number of exercised outcome with the total number of outcomes and multiplied by 100.

$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes Excercised}}{\text{Total Number of Decision Outcomes}}$$

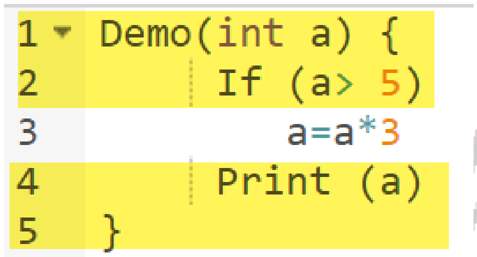
In this coverage, expressions can sometimes get complicated. Therefore, it is very hard to achieve 100% coverage.

Example of decision coverage

Consider the following code-

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```

Scenario 1: Value of a is 2

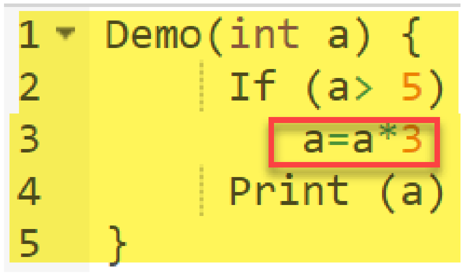


```
Demo(int a) {
     If (a> 5)
          a=a*3
     Print (a)
}
```

The code highlighted in yellow will be executed. Here the "No" outcome of the decision If (a>5) is checked.

Decision Coverage = 50%

Scenario 2: Value of a is 6



```
Demo(int a) {
     If (a> 5)
          a=a*3
     Print (a)
}
```

The code highlighted in yellow will be executed. Here the "Yes" outcome of the decision If (a>5) is checked.

Decision Coverage = 50%

Result table of Decision Coverage:

Test Case	Value of A	Output	Decision Coverage
1	2	2	50%
2	6	18	50%

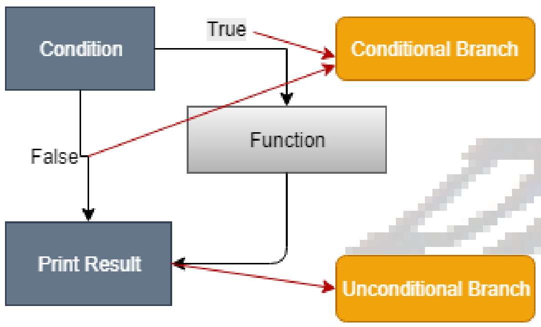
3. Branch Coverage

Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once. Branch coverage technique is a white box testing technique that ensures that every branch of each decision point must be executed.

For example, if the outcomes are binary, you need to test both True and False outcomes.

The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

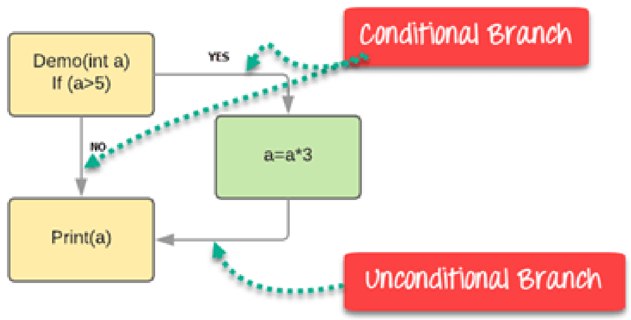


Example of Branch Coverage

To learn branch coverage, let's consider the same example used earlier

Consider the following code

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```



Branch Coverage will consider unconditional branch as well

Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	**33%**
2	6	18	50%	**67%**

4. Path coverage

A technique more powerful than both, statement & branch coverage, path coverage, as the name suggests, verifies every single path of an application at least once.

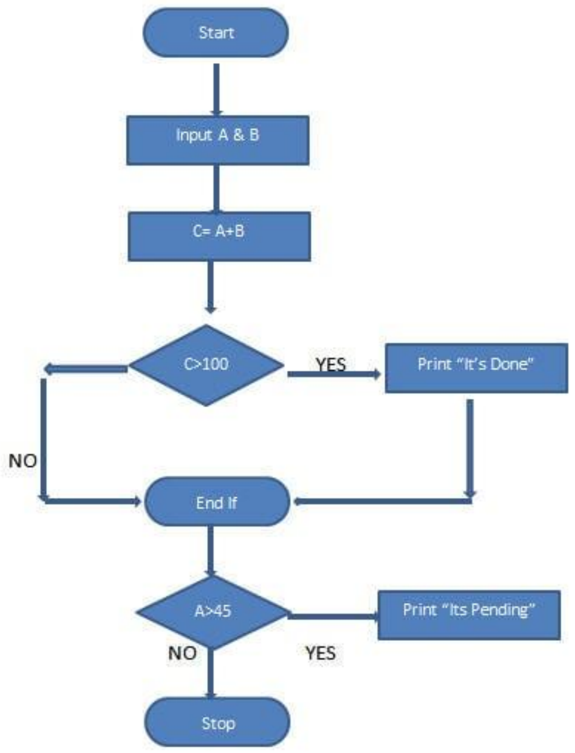
Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

```
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

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

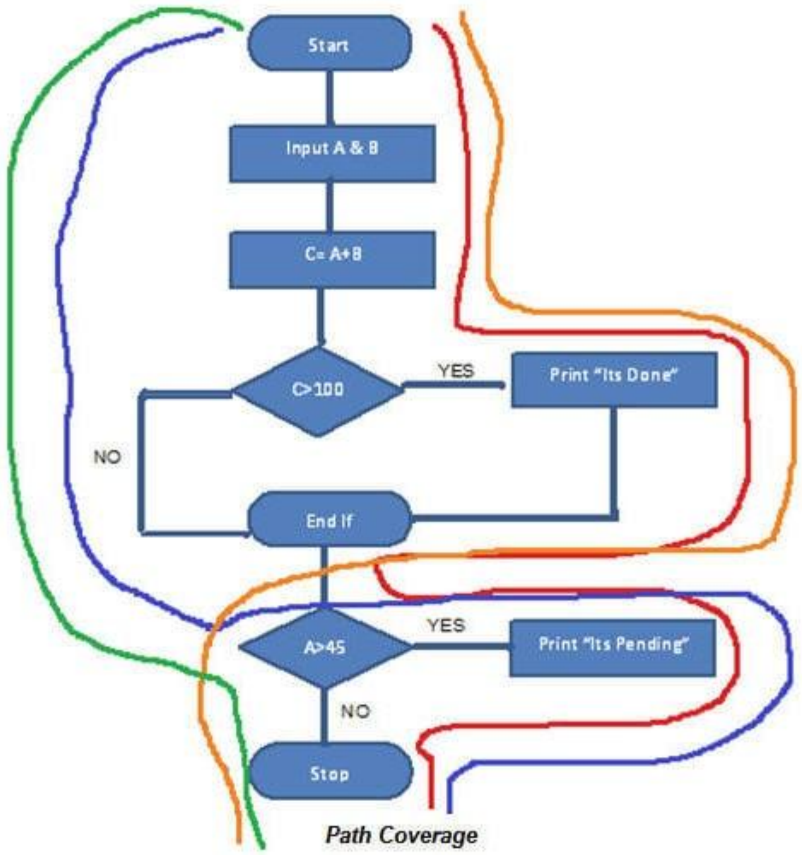
TestCase_01: A=50, B=60

TestCase_02: A=55, B=40

TestCase_03: A=40, B=65

TestCase_04: A=30, B=30

So the path covered will be:



Path Coverage

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)

Cyclomatic complexity

It is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.

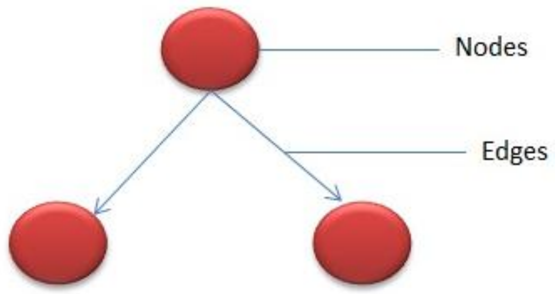
Cyclomatic complexity is computed using the control flow graph of the program**.**

A program structure is represented by a control flow graph (CFG).

CFG is a directed graph that shows a sequence of events (paths) in the execution through a component or system.

CFG consists of nodes and edges

- Node represents a statement or a sequence of statements
- Edge represents control flows from one statement to another



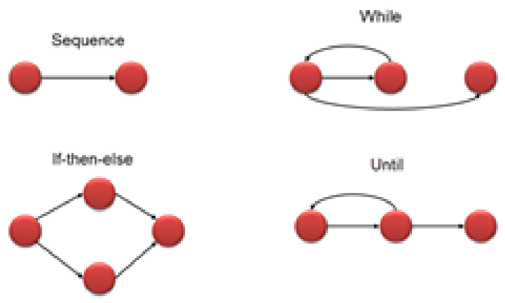
G = (N, E)

G: Control Flow Graph

V: Set of nodes

E: Set of edges

- Notations:



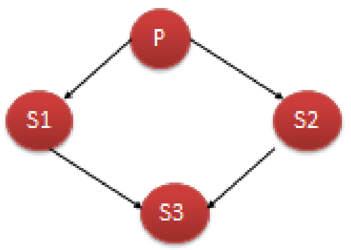
- Example:

if P

then S1

else S2

Go to S3



One testing strategy, called basis path testing, also called as path testing/ coverage, by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.

Cyclomatic complexity can be calculated in 3 ways:

1. The number of regions:

2. The cyclomatic complexity, V(G) for a flow graph G is defined as,

V(G) = E –N + 2

E: No. of edges N: No. of nodes

3. V(G) = P + 1

P: No. of Predicate node