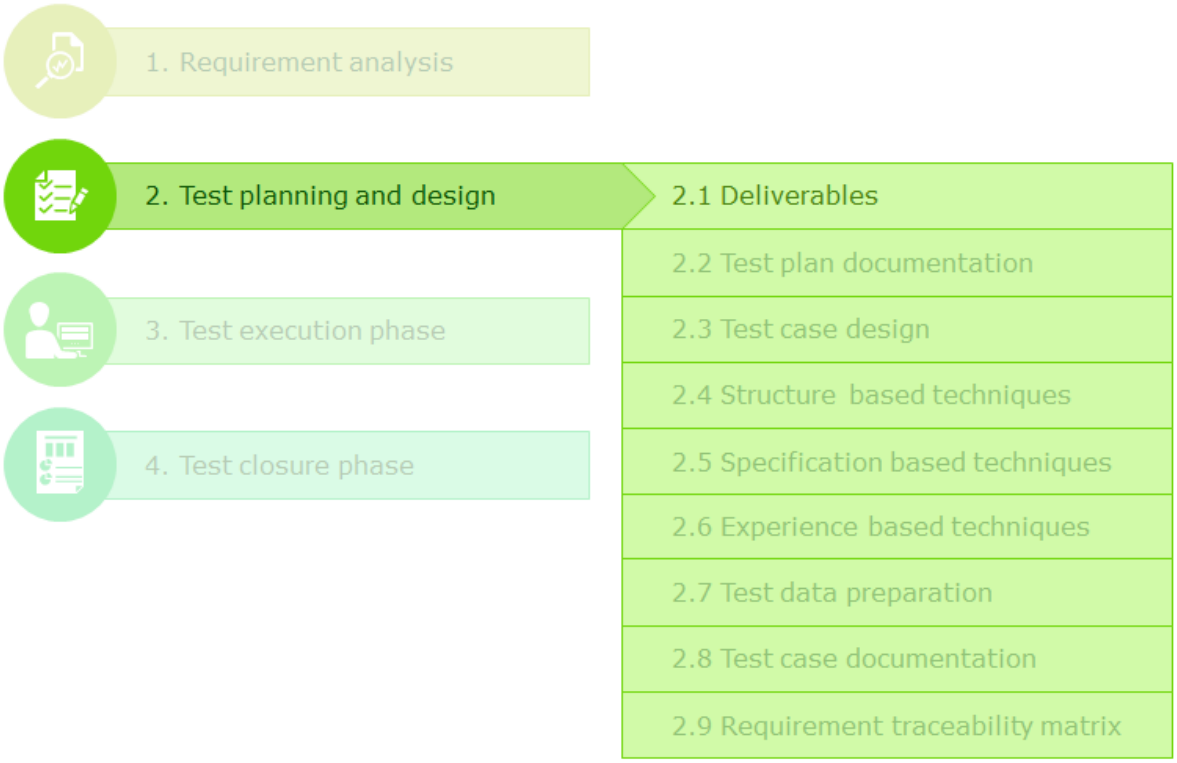
Deliverables of Test Planning and Design Phase



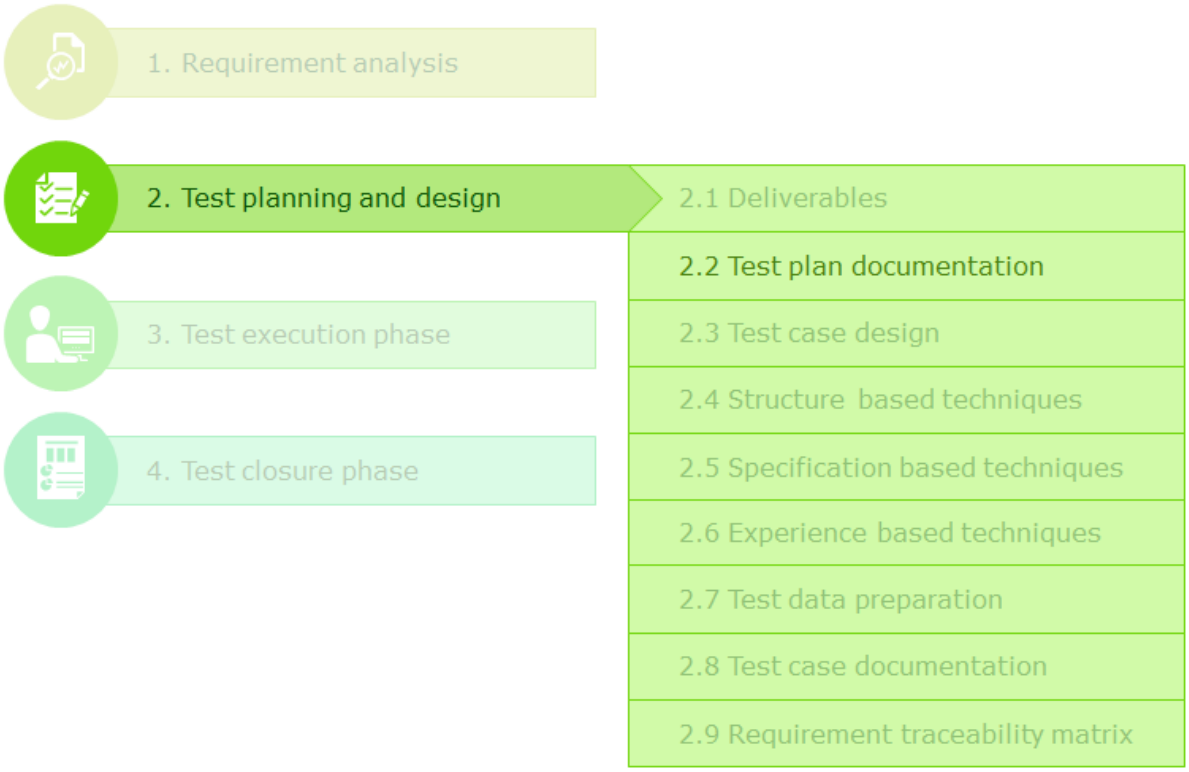
This is the stage from where the testing team is going to have deliverables.

Deliverables of test planning and design phase

The three major deliverables of this phase are

- Test Plan Document (TPD)
- Test Case Document (TCD)
- Requirement Traceability Matrix (RTM)

2.2 Test Plan Documentation



A Test Plan Document is prepared by the testing team at the beginning of Test Planning and Design Phase of the STLC. Its purpose is to elucidate, well in advance, to all the stakeholders (requirement owners, development team and testing team) about

- Scope of Testing - Define what to be tested and what not.
- Time and Resource Planning - Define time schedules and milestones, number of people required, knowledge and skills needed, hardware and software configurations of the test environments
- Operating Protocols - Define roles and responsibilities of stakeholders, criteria for starting, pausing, resuming and stopping various testing activities, risk management and deliverables.

It is usually prepared by the test manager or the test lead. The testers and other stakeholders use it as their reference.

This document can go through multiple drafts before it is agreed upon and finalized by all the stakeholders.

Contents of a Test Plan Document

As per IEEE 829-2008, also known as the 829 Standard for Software Test Documentation, a Test Plan Document needs to possess the following information.

Test Plan Identifier: A number, name or an alphanumeric code that uniquely identifies the Test Plan Document among all other project related documents.

Introduction: A high-level outline of the AUT and necessity for the present testing activity.

Test Items: The software and/or its specific components which have to be tested.

Features to be tested: Parts/Sections of the Software Requirements Specifications that are to be tested.

Features not to be tested: Parts/Sections of the Software Requirements Specifications that are NOT to be tested and reasons for it.

Approach: Types of tests that are to be done for specific software components or specific requirements and the intention for choosing that specific approach.

Item Pass/Fail Criteria: Pass/fail criteria defined at the level of software components or requirement sections or test types.

Suspension criteria and resumption criteria: A condition (or a combination of conditions) that would affect the testing tasks to an extent that they would be suspended. A condition or combination of conditions that are required to be met to resume suspended tests.

Test deliverables: Documents, artifacts and proofs that need to be delivered at specific milestones during the course of the testing phase.

Test tasks: Diverse kinds of testing tasks that must be performed, the execution order and their respective entry and exit criteria.

Environmental needs: The hardware and software configurations that the testing team needs along with their required dates and timelines to achieve the testing objectives as planned.

Staffing and training needs: The number of testers needed to perform all the testing tasks based on the estimated testing effort, the skills that they require, their current skill levels and training plans to cover the skill gaps, if any.

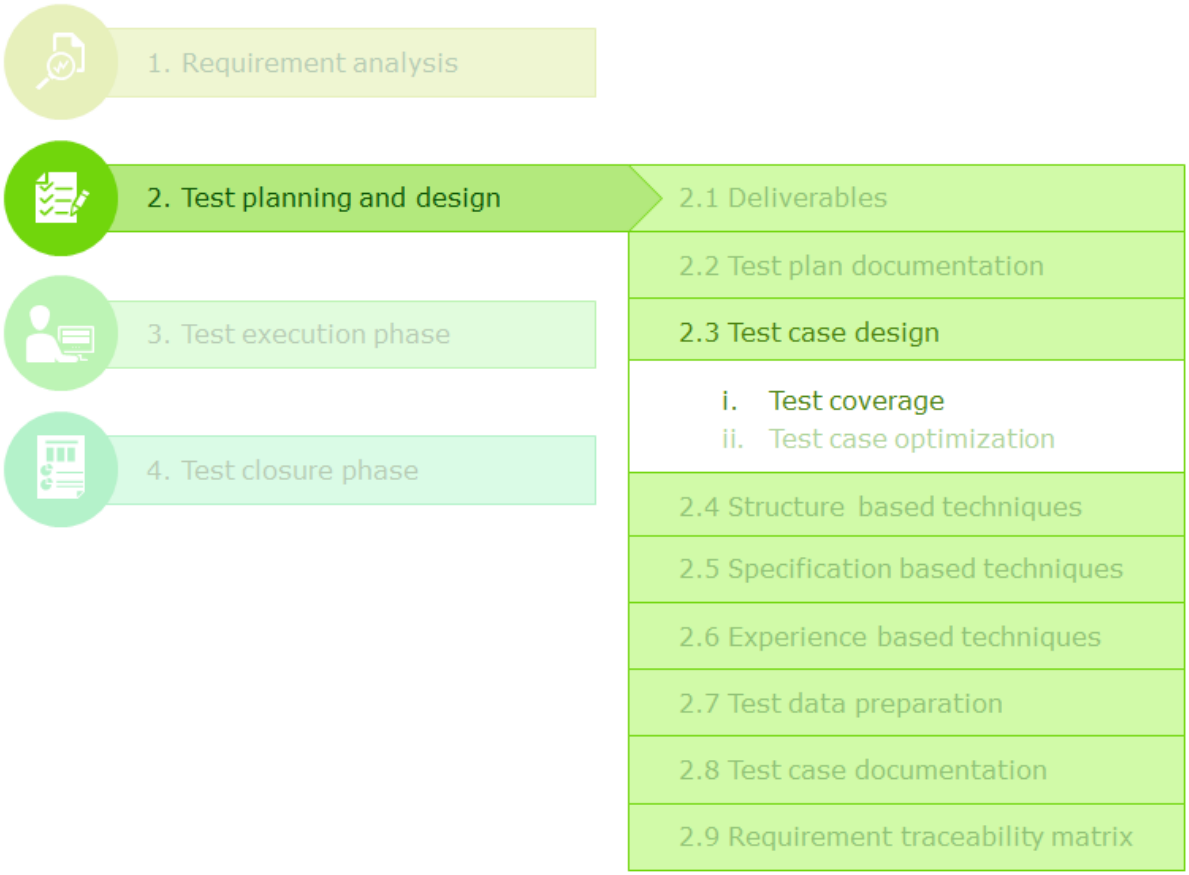
Responsibilities: Specific responsibilities of different project team members and stake holders.

Schedule: The planned start and end dates of different testing tasks in the STLC with defined milestones at different points in time.

Planning risks and contingencies: List of possible risks during the STLC, their probability of occurrence, their impact and the contingency plan to reduce or eliminate their impact, if they occur.

Approvals: This section is used to record the approvals given for the test plan by different stakeholders of the project along with the dates and versions that were approved.

2.3.1 Test Coverage



Once the test planning is done, the testers start designing the test cases.

Test designing is the act of creating and documenting the set of test cases (test suite) to be used for testing the software.

A well designed test suite, upon execution, would be able to objectively state the quality of the software in terms of

- Test coverage
- Number of defects found

Test coverage

- Test coverage is a measure of the size or amount of software that has been validated using the test suite.
- Based on the measuring unit used, test coverage can be stated in two ways.

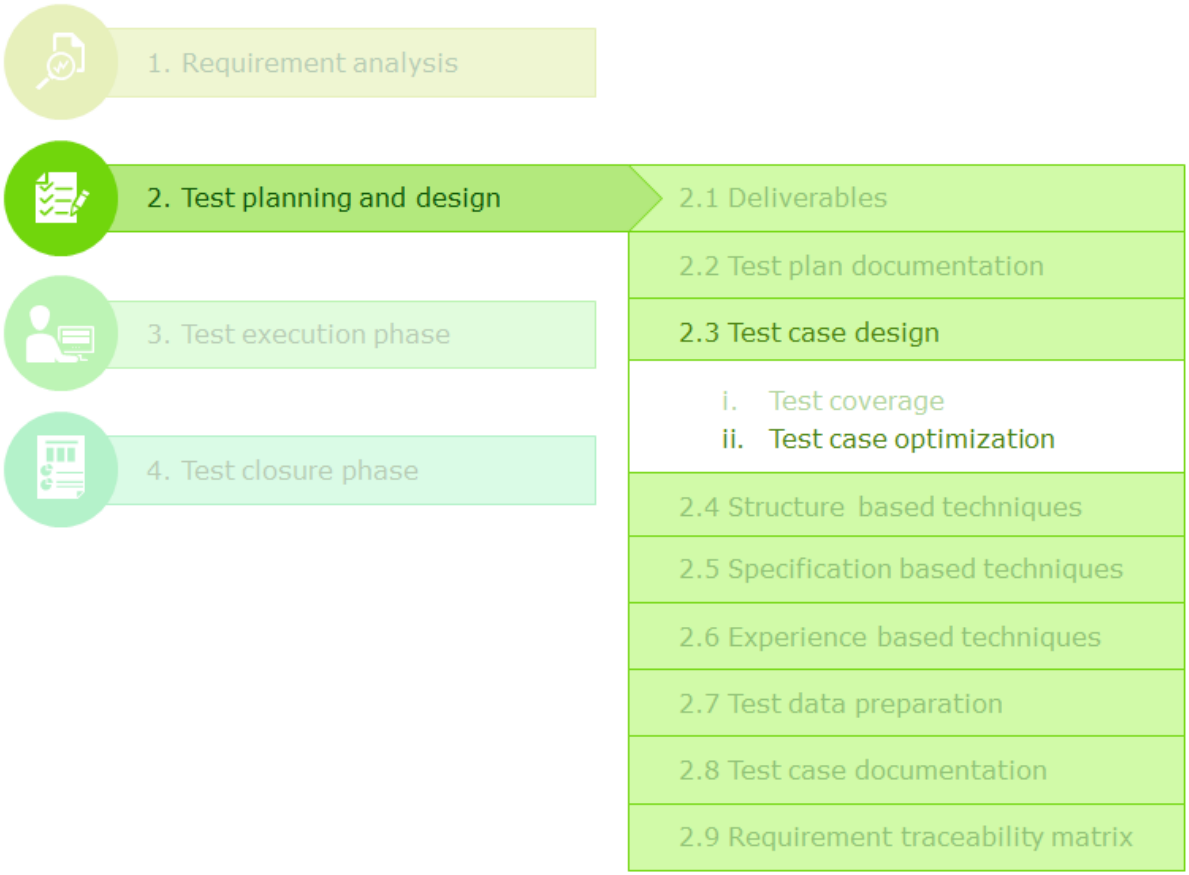
Code coverage

- Test coverage is expressed in terms of number of program statements tested.
- 100% code coverage is usually the goal of white box tests designed by developers. E.g., unit tests and integration tests.

Specification coverage

- Test coverage is expressed in terms of number of requirements in the SRS that have been tested.
- 100% requirement specification coverage is usually the goal of tests performed by black box testers, who are restricted in terms of code access or programming knowledge(or both). E.g., system tests and user acceptance tests.

2.3.2 Test Case Optimization



If you achieve 100% test coverage, it gives the confidence that you have encountered all possible defects that are existing in the software, at least once.

Coverage Effectiveness

- A test suite design is said to **effective** if 100% test coverage is achieved by its test cases.
- Exhaustive testing (testing a software using all probable data inputs and configurations) can ensure 100% test coverage but it is impossible as it would take months (or even years).

Coverage Efficiency

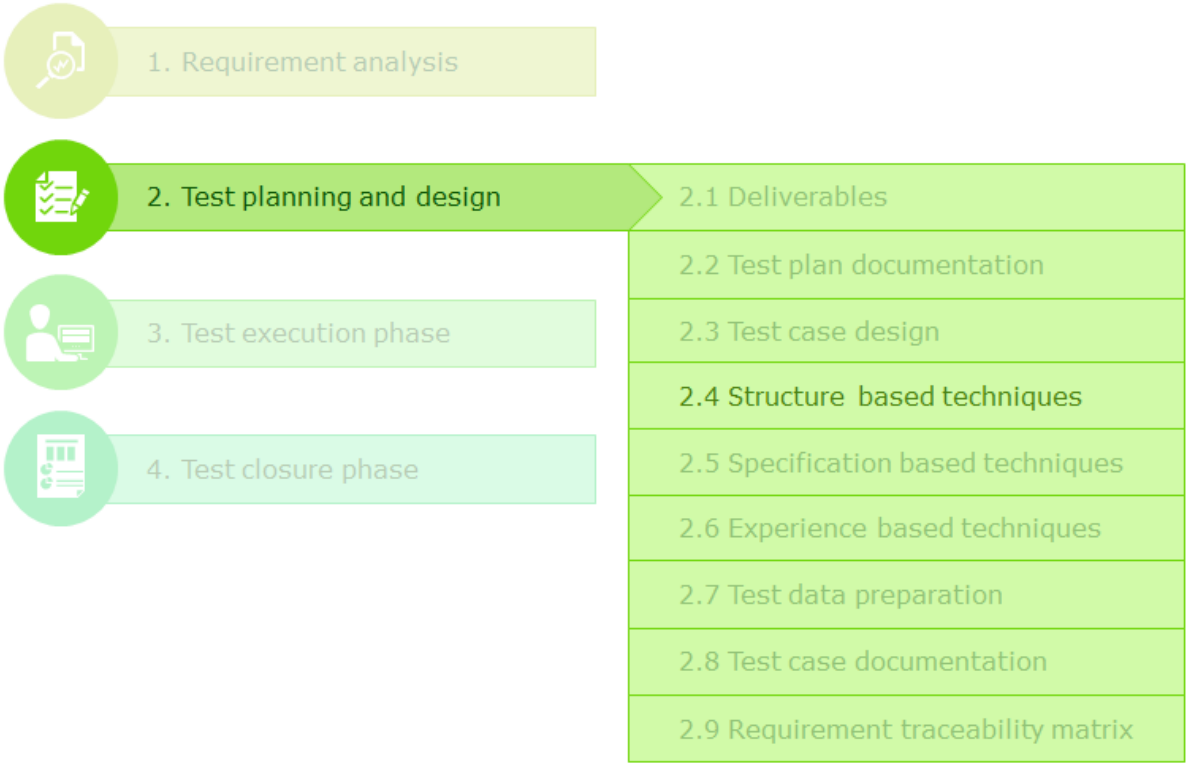
- If you take exit criteria (when the testing should stop) from the test plan document into account, then you will understand that you have to achieve 100% test coverage with as less number test cases as possible.
- A test suite design is said to be **efficient** if it uses the least number of tests possible to achieve 100% test coverage.

Test case optimization is the process limiting the number of test cases, on the basis of sound principles and scientific assumptions, required to achieve maximum test coverage.

The techniques that testers use to optimize their test cases can be classified into three groups, as shown below:

- Structure based techniques
- Specification based techniques
- Experience techniques

2.4 Structure Based Techniques



When and by whom are structure based tests done?

Developers, who have coding skills and code visibility, perform structure based tests during the Component Testing level to make sure that 100% of the application is covered in terms of source code structure.

100% requirement coverage is not the goal of structure based tests, but 100% of the source code. Test design techniques differ based on how the source code coverage is measured.

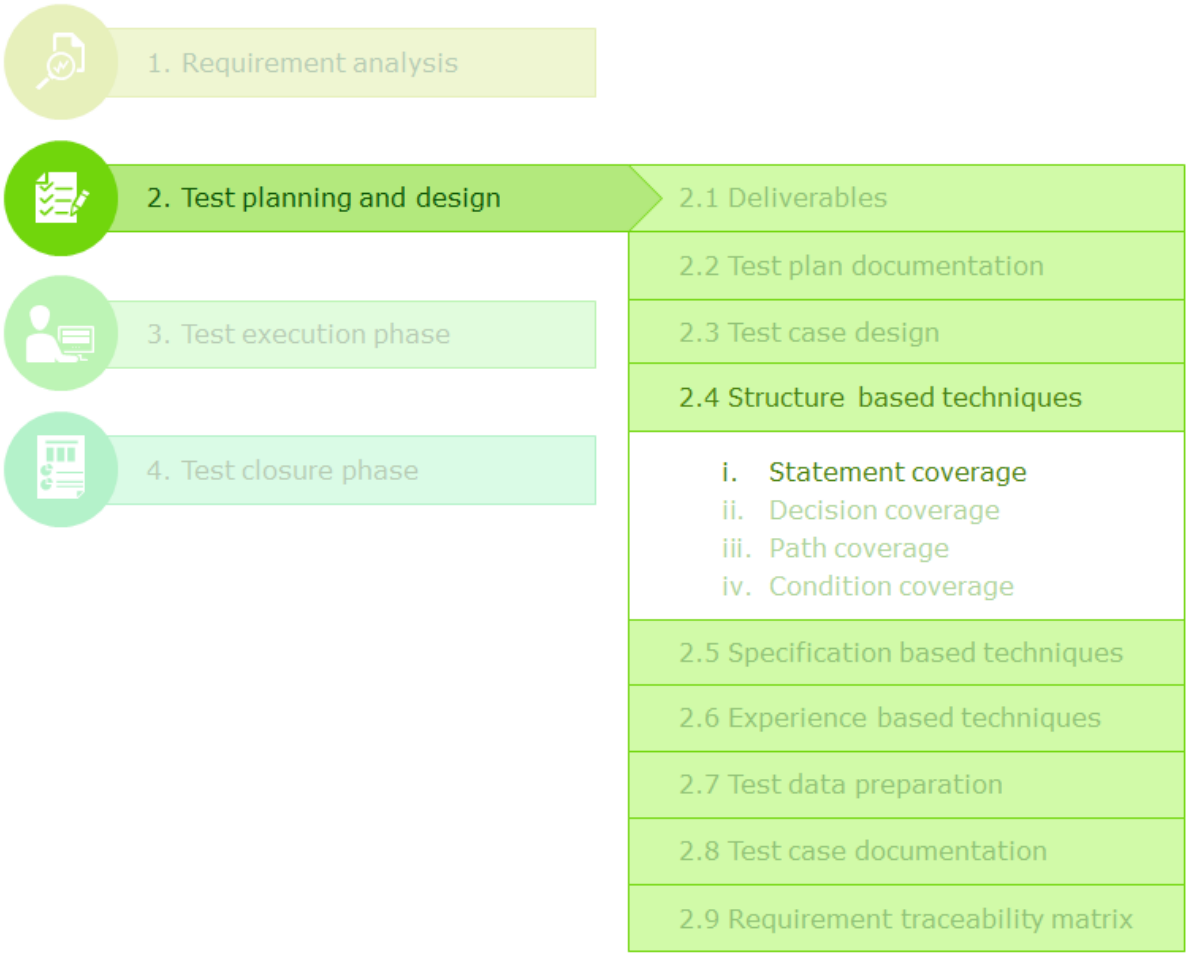
Depending on the organization practices and metrics, design specifications, etc. developers can choose one of the following program components as the 'coverage item' to measure their test coverage.

1. Statement coverage
2. Decision coverage
3. Path coverage
4. Condition coverage

Structure based test coverage is derived as:

$$\text{Coverage} = \frac{\text{Number of coverage item exercised}}{\text{Total number of coverage item}} \text{ X } 100\%$$

2.4.1 Statement Coverage



- The objective of statement coverage technique is to ensure that every statement gets executed at least once by the set of tests chosen.
- Developers usually use tools that measure statement coverage during test execution.
- It is a must for testing the source code of safety critical systems such as X-ray machines, aircraft control modules, nuclear-reactor management systems, etc. since any failure due to defects might lead to life threatening situations.
- We will use a simple 'control flow' diagram, as it simplifies the understanding of the structure of a program, for this technique (and all other structural techniques too). In a 'control flow' diagram

Generic statements are represented by rectangles 

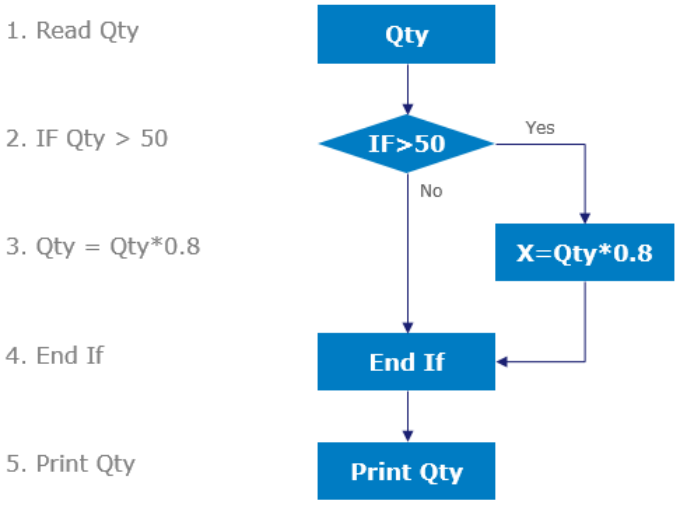
Branching statements are represented by a diamonds 

-
-

Example

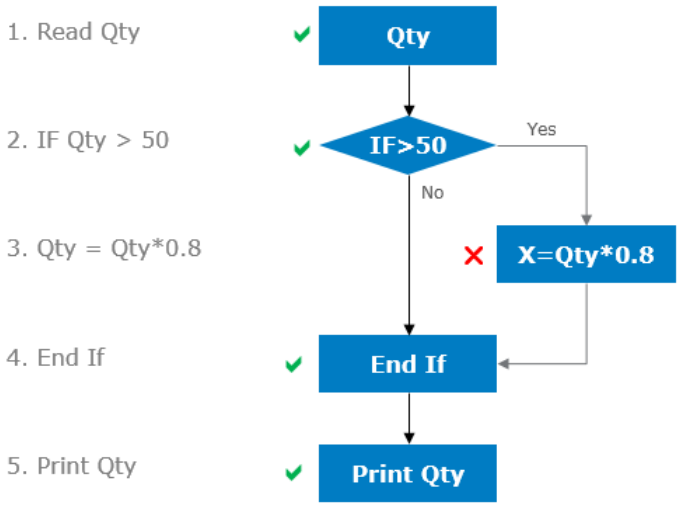
Lets consider the pseudo code below in which there are numerous values which can be used as test data for the variable 'Qty'.

First we have to arrive at the control flow diagram.

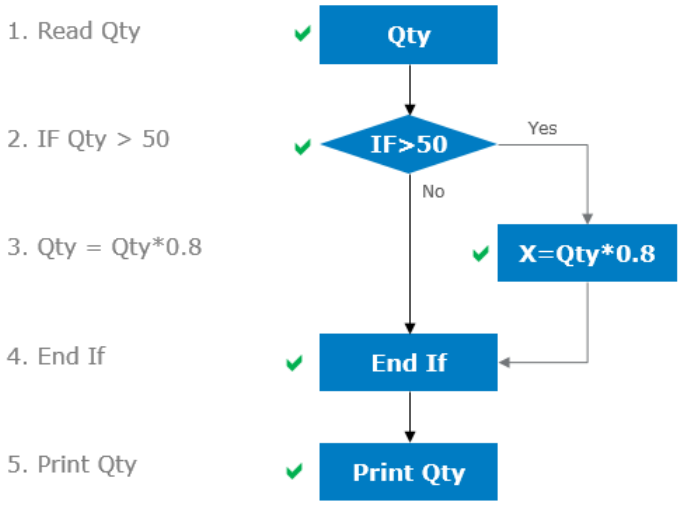


Now the control flow is analyzed and value for the variable 'Qty' is chosen such that maximum number of statements get executed at least once.

For example if we choose the value as 10, then it executes 4 out of 5 statements. Hence the test case with test data of 10 will ensure 80% statement coverage.



However, if we choose a test data value of 80, then all 5 out of 5 statements will get executed.

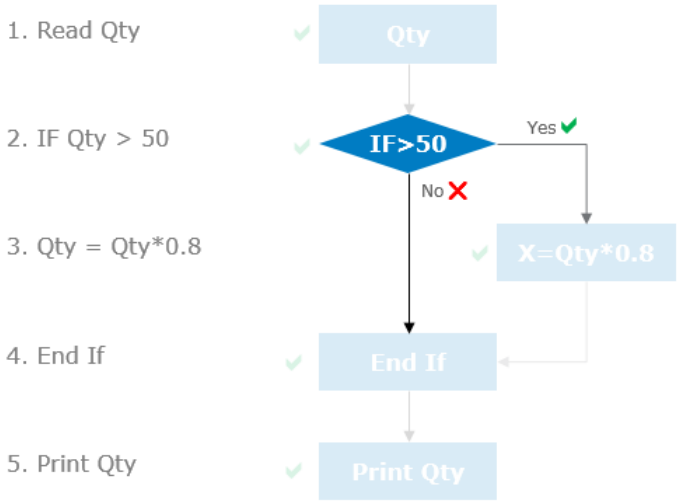


Hence, the tester/developer chooses minimum number of test input values, which in our case is only 1 (Qty = 80), to achieve 100% statement coverage.

Drawback of Statement Coverage

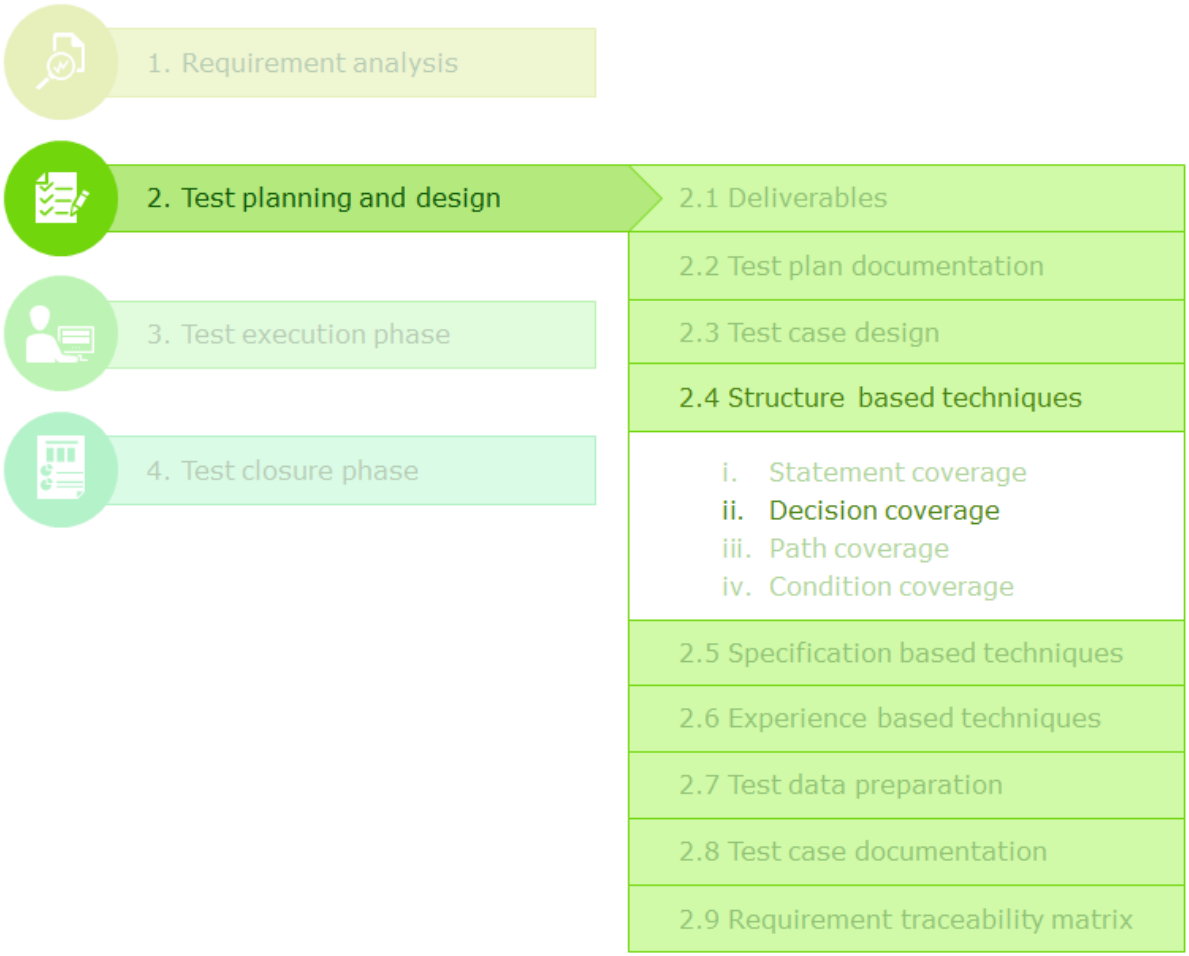
In the example in the previous page, even though we achieved 100 % statement coverage with an input value of 80, the decision making 'If' statement has not been tested for both possible decisions - Yes and No.

Only the 'Yes' decision been tested. There is a possibility that the 'No' decision might cause some defect.



To avoid such scenarios, instead of statement coverage, developers choose decision coverage techniques.

2.4.2 Decision Coverage

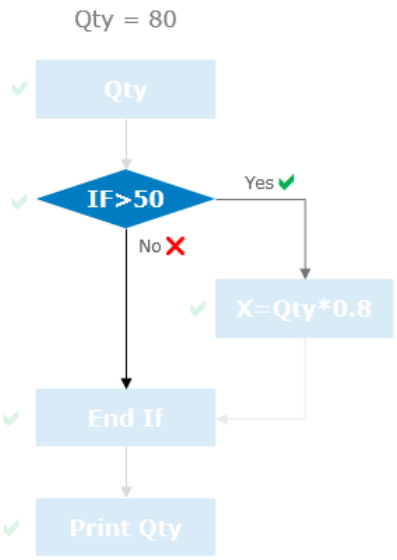


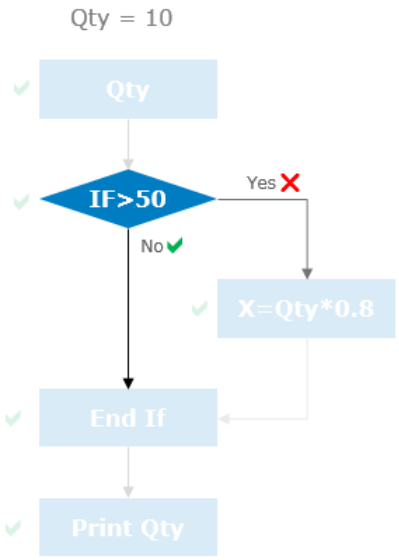
Decision coverage involves designing tests to achieve coverage of 100% decision outcomes that are available (e.g. the True and False conditions of the IF statement)

- It can also be called branch testing or control flow testing
- 100% decision coverage ensures 100% statement coverage, but the vice versa is not true
- Decision coverage can also be measured using tools

If we're measuring the test coverage by decisions and use either Qty = 80 or Qty =10, only one of the two outcomes of the decision statement (If > 50) is covered (only 50% decision coverage).

However if we use both the values, both the possible outcomes of the decision are going to be covered. (100% decision coverage)





Hence, 2 tests are essential to attain 100% decision coverage.

Statement Coverage and Decision Coverage

Problem Statement:

```
1.  Read Q
2.  If Q > 50
3.      X = Q * 0.08
4.  Else
5.      If Q < 0
6.          Print "Error"
7.      Else
8.          X = Q * 2
9.      End If
10. End If

11. Print X
```

1. Create the control flow diagram for the above pseudo code.
2. With the help of the flow diagram determine the number of tests required for statement coverage.

3. With the help of the flow diagram determine the number of tests required for decision coverage.

Statement Coverage and Decision Coverage in a 'While' Loop

Problem Statement:

```
1. Read NumberOfExams
2. While NumberOfExams != 0
3.     Read Mark
4.     TotalMarks = TotalMarks + Mark
5. End While

6. Print TotalMarks
```

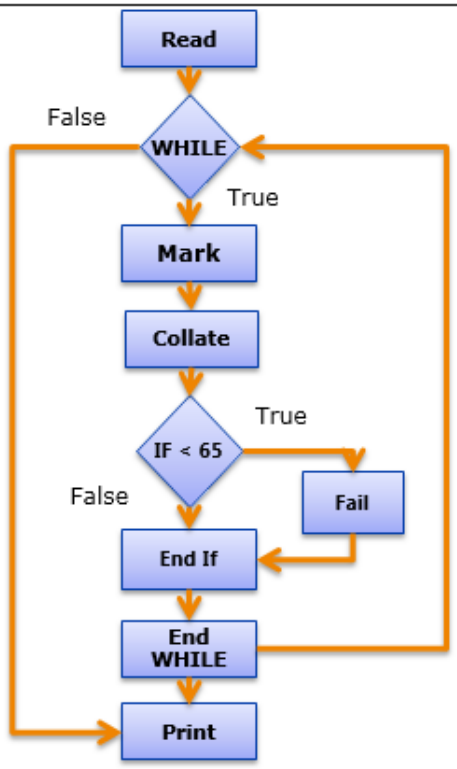
For the above pseudo code, find the number of tests necessary to attain

1. 100% statement coverage
2. 100% decision coverage

Statement Coverage and Decision Coverage in a 'While' Loop with 'If' Condition inside

Problem Statement:

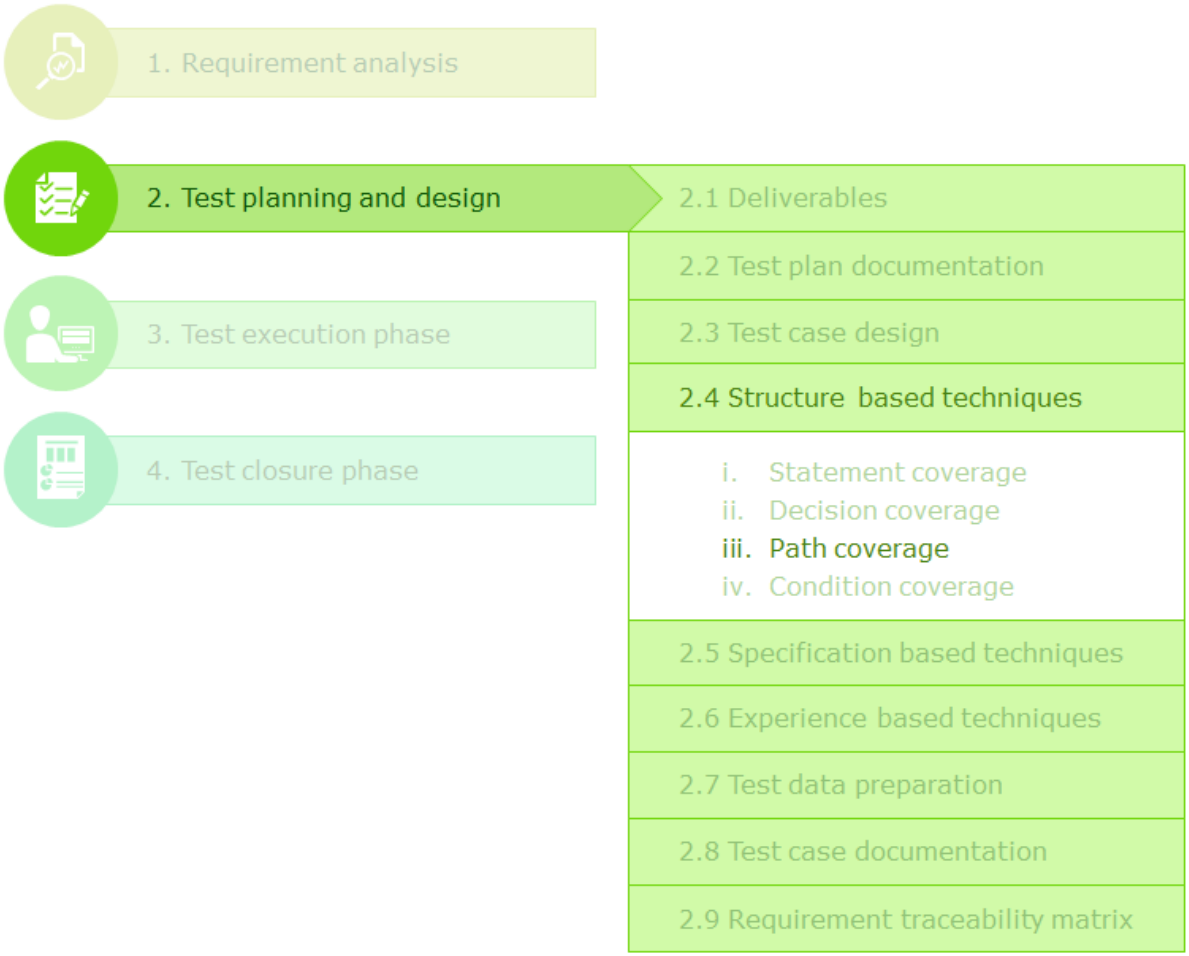
1. Read Number of Exams
2. WHILE more exams DO
3. Mark answers
4. Collate total marks
5. IF Score < 65
6. Fail = Fail + 1
7. End If
8. END WHILE
9. Print "No more to mark"



In the above example, how many tests are necessary to attain

1. 100% statement coverage?
2. 100% decision coverage?

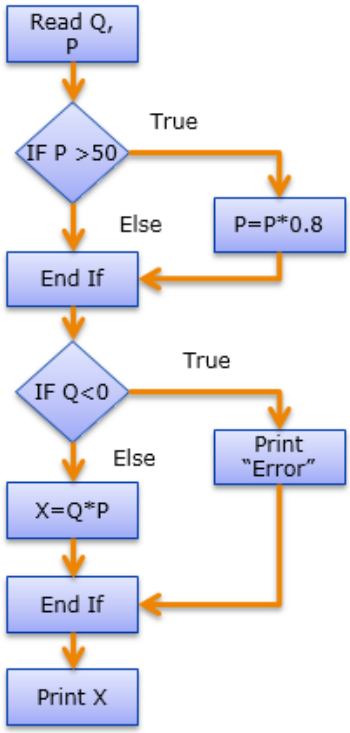
Path Coverage Technique



Paths refer to the number of ways in which the program execution can flow through the program code.

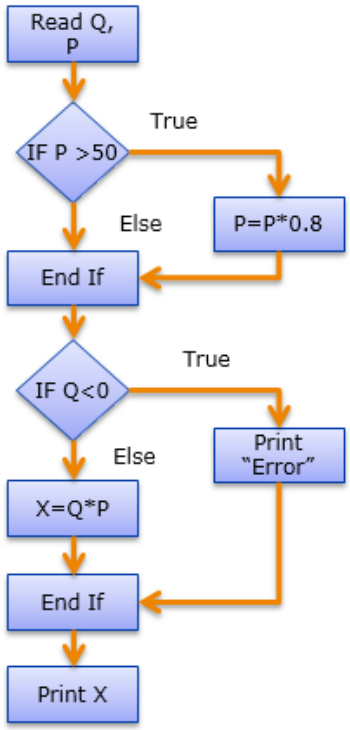
Can you guess the number of ways in with the program execution can flow through the below program?

```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```

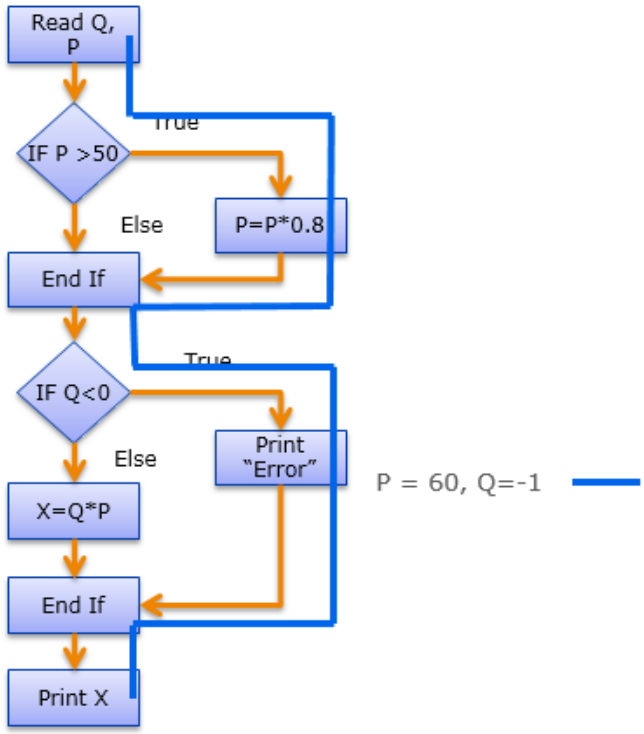


Path Coverage and Path Testing

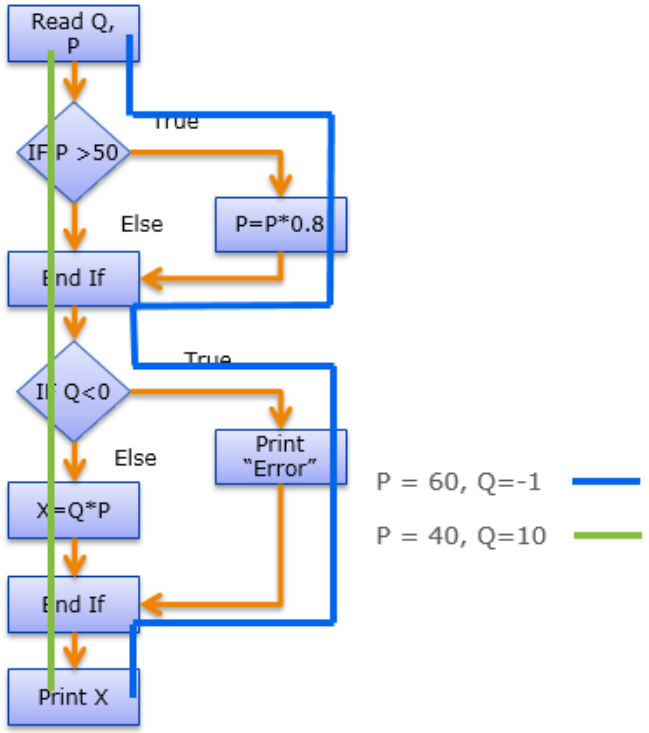
```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```



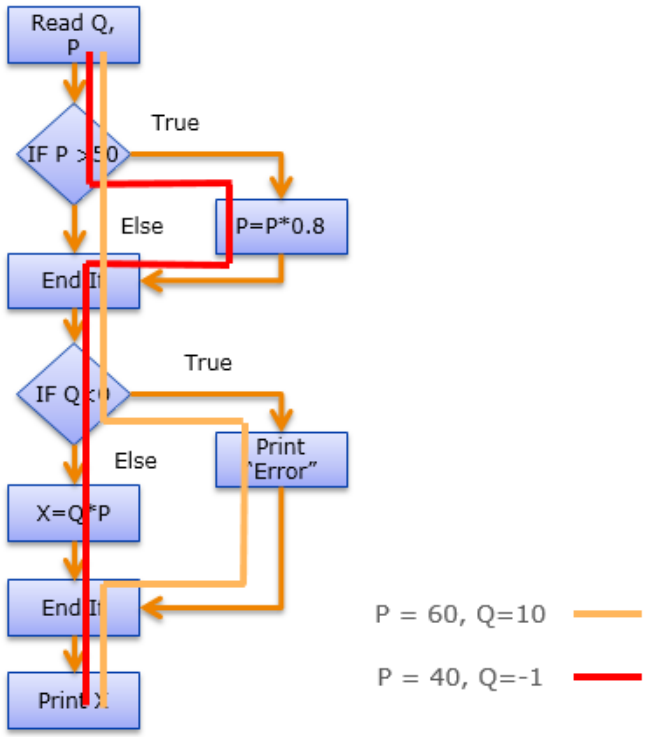

```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```



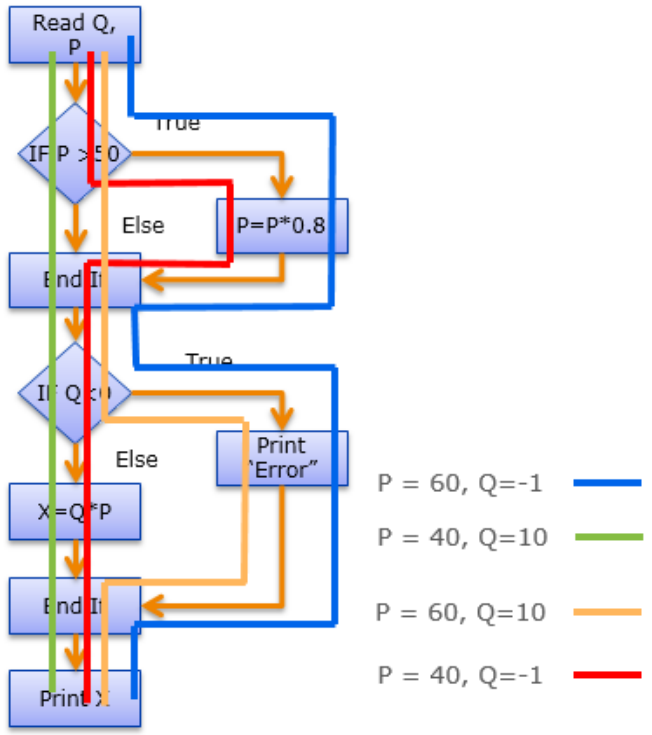
```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```



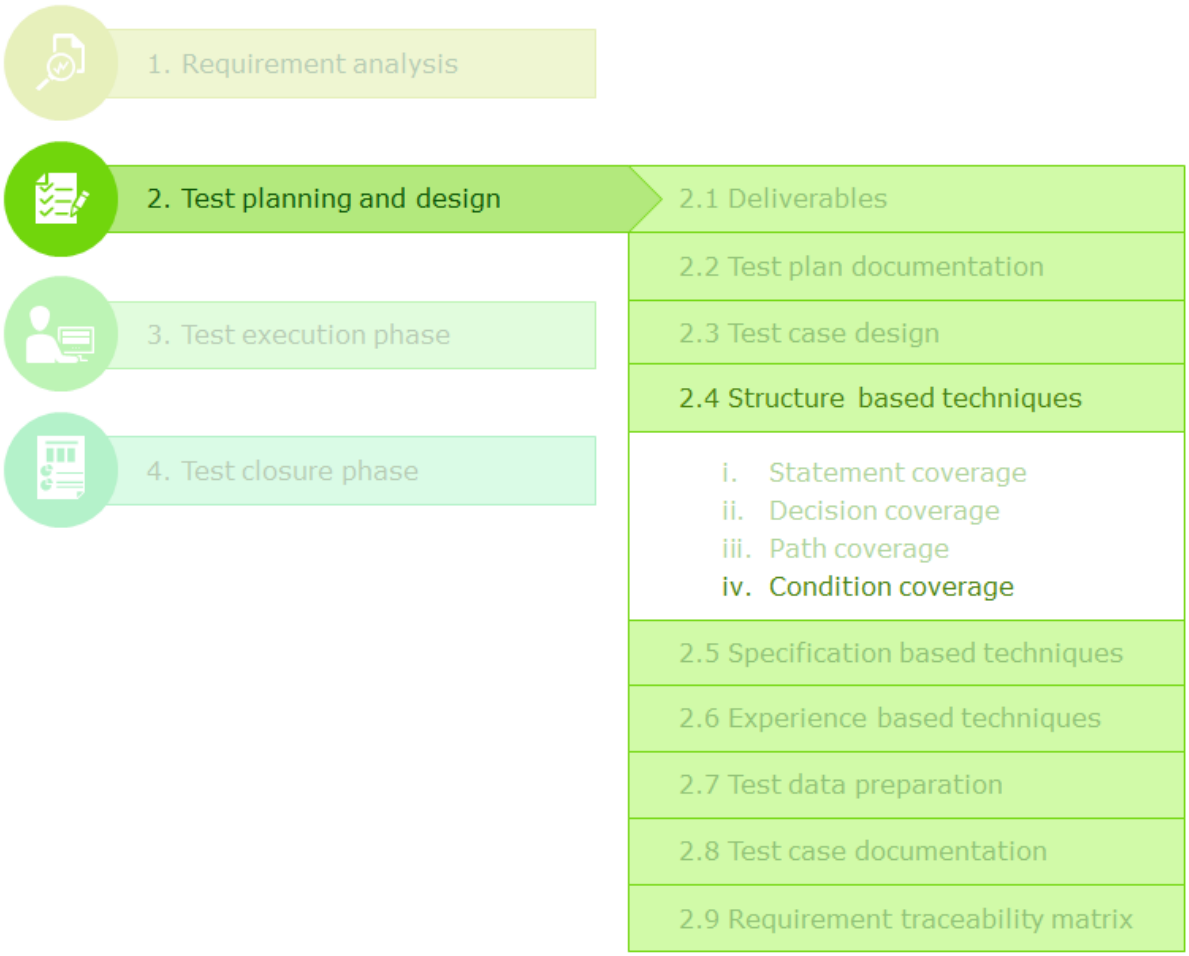

```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```



```
1.   Read Q, P


2.   IF P > 50
3.       X = P * 0.8
4.  Else
5.  End If


5.  IF Q < 0
6.       Print "Error"
7.  Else
8.       X = Q * P
9.  End If
11.  Print X
```



Condition Coverage Technique



Complex software like a life insurance underwriting system can either accept or reject a policy enrollment request from a customer based on combination of a number of inputs - Income, age, already existing policies, number of claims in previous policies, existing health conditions etc. The customer is not only evaluated based on each individual data point collected, but also conditional combinations of such inputs.

A decision statement of a such computer programs can involve complex boolean expressions which have

- Multiple input values
- Multiple relational evaluations between the inputs

Such expressions are called decision predicates.

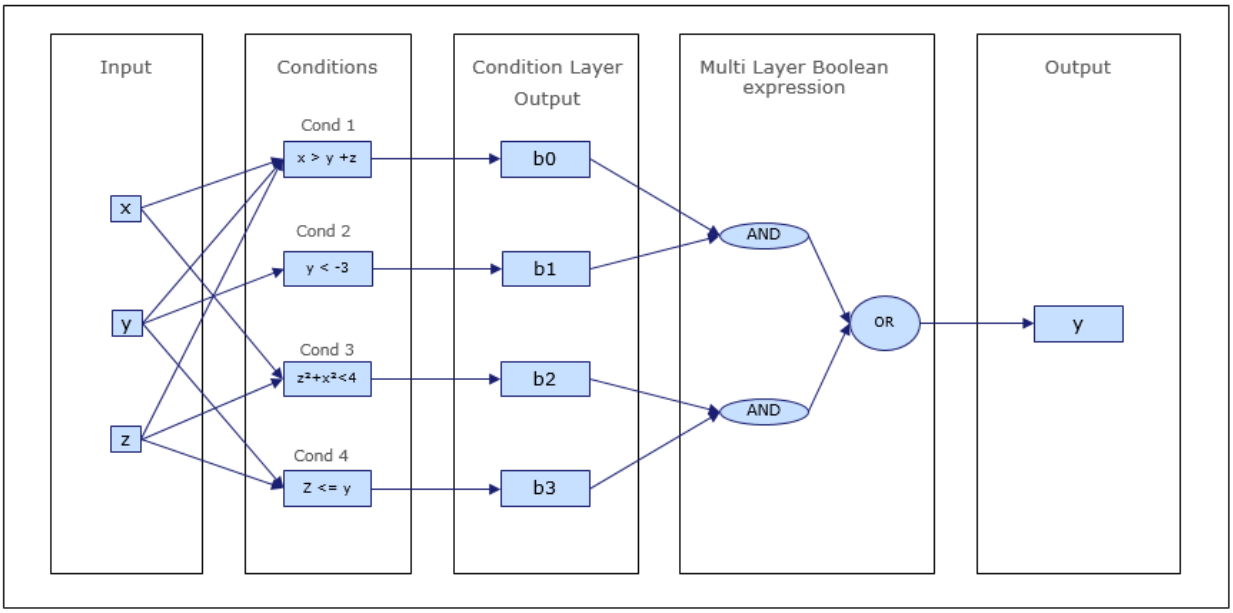
Here's an example of a decision predicate.

1. b_out = ((x>y+z) AND (y<-3)) OR ((z²+x²<4) AND (z≤y))

Even though there are multiple inputs and multiple evaluations, the final result can branch the program flow only in two ways (b_out = true, b_out = false). Here, testing all possible decisions (2) is not enough. We need to test all possible ways in which the inputs arrive at either of the decisions.

For scenarios like these, we use condition coverage techniques. In this technique, the goal is to cover all possible conditional evaluations of the boolean expressions.

Lets see how we can derive tests out of the decision predicate given in the example above, to achieve 100% decision coverage.



One attains "condition testing coverage" by executing test cases till all the conditions in the decision produced true at least once and false at least once.

Column 1	x	y	z	b0	b1	b2	b3	b_out
Test Case 1	1	0	0	TRUE	FALSE	TRUE	TRUE	TRUE
Test Case 2	0	1	0	FALSE	TRUE	TRUE	TRUE	TRUE
Test Case 3	NA	-4	NA	Do Not Know	TRUE	Do Not Know	Do Not Know	Do Not Know
Test Case 4	2	NA	2	Do Not Know	Do Not Know	FALSE	Do Not Know	Do Not Know
Test Case 5	NA	0	1	Do Not Know	FALSE	Do Not Know	FALSE	FALSE

Each of the condition outputs (b0, b1, b2, b3) gives true at least once and false at least once. We are not concerned about a given value in few test cases because it does not influence the condition for which we require correct output. They have been marked as “NA”.

Incidentally, the table also demonstrates that "100% decision coverage" has been achieved for this particular decision, since b_out gives true at least once and false at least once, hence both branches of any code will be covered.