18.1 SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. The product should be testable. "Software testability is simply how easily a computer program can be tested." **Testability** exhibits following characteristics:-

a) **Operability:-** If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests.
b) **Observability:-** "What you see is what you test." Inputs provided as part of testing produce distinct outputs.
c) **Controllability:-** "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input.
d) **Decomposability:-** By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.
e) **Simplicity:-** "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity; structural simplicity, and code simplicity.
f) **Stability:-** "The fewer the changes, the fewer the disruptions to testing."
g) **Understandability:-** "The more information we have, the smarter we will test."

Test Characteristics:- The following attributes of a "good" test:

i. A good test has a high probability of finding an error.
ii. A good test is not redundant. There is no point in conducting a test that has the same purpose as another test.
iii. A good test should be "best of breed".
iv. A good test should be neither too simple nor too complex.

18.2 INTERNAL AND EXTERNAL VIEWS OF TESTING

Any software can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2) Knowing the internal workings of a product, that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach takes an **external view** and is called **black-box testing**. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

Reference:- R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, Ed 7, 2010.

The second requires an **internal view** and is termed **white-box testing.** White-box testing of software is predicated on close examination of procedural detail. White-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results.

18.3 WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing. Using white-box testing methods, you can derive test cases that

(1) Guarantee that all independent paths within a module have been exercised at least once,

(2) Exercise all logical decisions on their true and false sides.

(3) Execute all loops at their boundaries and within their operational bounds, and

(4) Exercise internal data structures to ensure their validity.

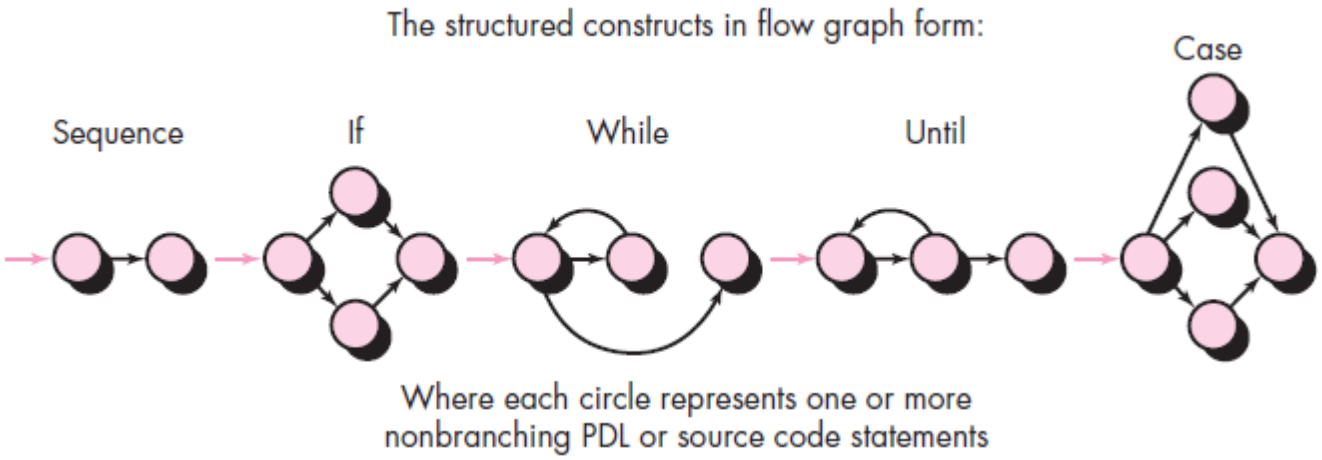
Types of White Box Testing

18.4 BASIS PATH TESTING

Basis path testing is a white-box testing technique. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

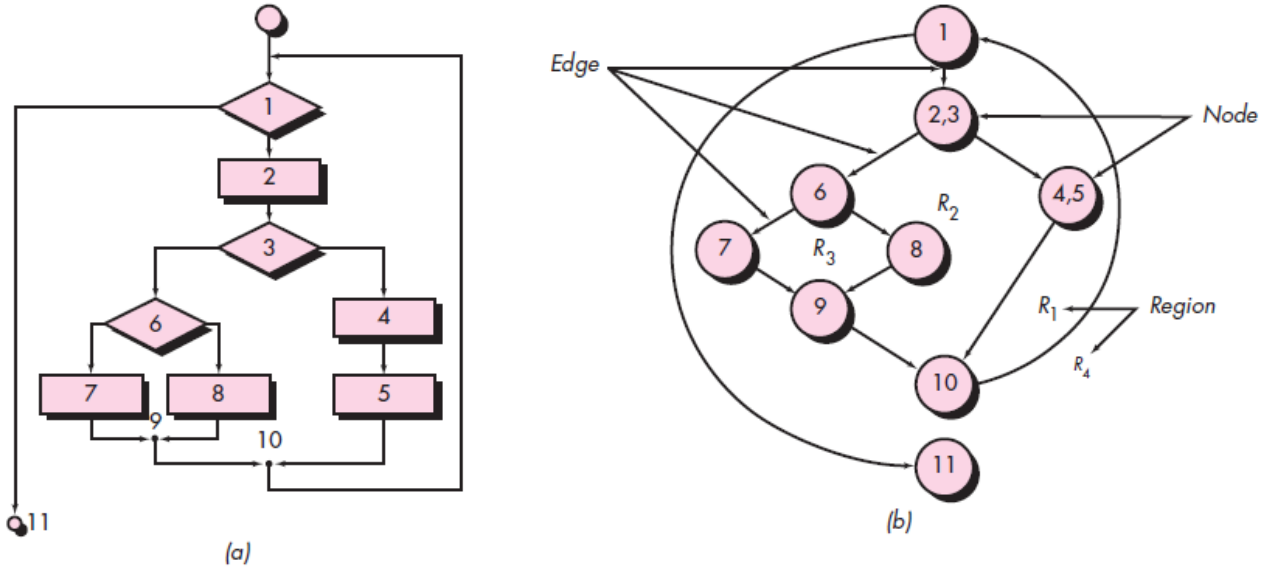
18.4.1 Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Figure 18.1.



Reference:- R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, Ed 7, 2010.

FIGURE 18.2 (a) Flowchart and (b) flow graph



A flowchart is used to depict program control structure. Referring to Figure 18.2b, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. Areas bounded by edges and nodes are called regions. Each node that contains a condition is called **a predicate node** and is characterized by two or more edges emanating from it. **When counting regions, we include the area outside the graph as a region.**

18.4.2 Independent Program Paths

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. For example, a set of independent paths for the flow graph illustrated in Figure 18.2b is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Reference:- R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, Ed 7, 2010.

Paths 1 through 4 constitute a basis set for the flow graph in Figure 18.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. a number of different basis sets can be derived for a given procedural design.

How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer. Cyclomatic complexity is a software metric. the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity V(G) for a flow graph G is defined as

V(G) = E - N + 2

where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity V(G) for a flow graph G is also defined as

V(G) = P + 1

Where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 18.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. V(G) = 11 edges - 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 18.2b is 4.

18.4.3 Deriving Test Cases

The following steps can be applied to derive the basis set:

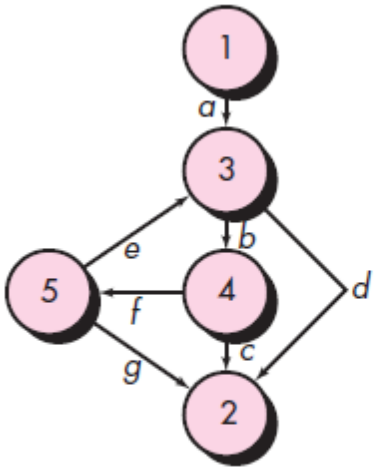
1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

Reference:- R.S. Pressman, Software Engineering: A Practitioner’s Approach, McGraw-Hill, Ed 7, 2010.

18.4.4 Graph Matrices

A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).

See the slides (104-154) chapter 8 Software testing by KK Aggarwal for further numerical examples of white Box testing.

Reference:- R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, Ed 7, 2010.