## 5.1 Software Testing

Testing is the process of executing a program with the intent of finding errors.

**5.1.1 Why should We Test?**

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved. In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

**5.1.2 Who should Do the Testing?**

o Testing requires the developers to find errors from their software.

o It is difficult for software developer to point out errors from own creations.

o Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

**5.1.3 What should We Test?**

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs.

**5.2 Some Terminologies**

**5.2.1 Error, Mistake, Bug, Fault and Failure**

People make errors. A good synonym is mistake. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "bugs".

A fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A failure occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

**5..2.2 Test, Test Case and Test Suite**

Test and Test case terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description. Inputs are of two types: pre conditions and actual inputs. Expected outputs are also two types: post conditions and actual outputs. Every test case will have an identification.

During testing, we set necessary preconditions, give required inputs to program, and compare the observed output to know the outcome of a test case. If expected and observed outputs are different, then there is a failure and it must be recorded properrly inorder to identify the cause of failure. If both are same, then, there is no failure and program behaved in the expected manner. A good test case has a high probability of finding an error. The test case designer's main objective is to identify good test cases. The template for typical test case is given in Fig. 2.

The set of test cases is called a test suite. Hence any combination of test cases may generate a test suite.

| Test Case ID | |
|---|---|
| Section-I (Before Execution) | Section-II (After Execution) |
| Purpose : | Execution History: |
| Pre condition: (If any) | Result: |
| Inputs: | If fails, any possible reason (Optional); |
| Expected Outputs: | Any other observation: |
| Post conditions: | Any suggestion: |
| Written by: | Run by: |
| Date: | Date: |

**Fig. 2:** Test case template

### 5.2.3 Verification and Validation

Verification is primarily related to manual testing, because it requires looking at documents and reviewing them. However, validation usually requires the execution of program.

**Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Hence verification activities are applied to early phases of SDLC such as requirements, design, planning etc. We check or review the documents generated after the completion of every phase in order to ensure that what comes out of that phase is what w expected to get.

**Validation** is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements. Therefore validation requires actual execution of the program and is also known as computer based testing.

Hence, testing incudes both verification and validation.

**Testing= Verification+Validation**

### 5.2.4 Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user /customer and may range from adhoc tests to well planned systematic series of tests. Acceptance testing may be conducted for few weeks or months. The discovered errors will be fixed and better quality software will be delivered to the customer.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

**Alpha Tests** are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

**Beta Tests** are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

After testing, customers are expected to report failures to the company. After receiving such failure reports, developers modify the code and fix the bug and prepare the product for final release.

## 5.3 Functional Testing

Functional tesing refers to testing, which involves only observation of the output for certain input values. There is no attempt to analyse the code. We ignore the internal structure of the code. Therefore, functional testing is also referred to as black box testing. Functionality of the blac box is understood completly in terms of inputs and outputs as shown in Fig. 3.

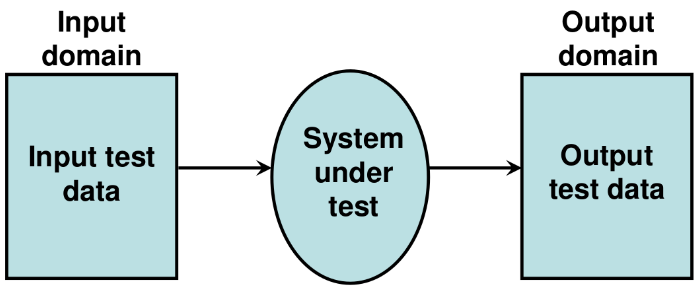


**Fig. 3:** Black box testing

## Different techniques for designing test cases in Functional Testing

- Boundary Value Analysis
- Equivalence Class Testing
- Decision Table Based Testing
- Cause Effect Graphing Technique

### 5.3.1 Boundary Value Analysis

Experience shows that test cases that are close to boundary conditions have higher chances of detecting an error. The basic idea of boundary value analysis is to use input variable values at their minimum, just above minimum, a nominal value, just below their maximum and at their maximum. Boundary value analysis test cases are obtained by holding the values of all, one variable at nominal value and letting that variable assume its extreme values.

Consider a program with two input variables x and y with boundaries (100, 300). The input domain is shown in Fig. 5. In figure each dot represents a test case. The boundary value analysis test cases are (200, 100), (200, 101), (200, 200) (200, 299), (200, 300), (100, 200), (101, 200) (299, 200) and (300, 200). Thus, for a program of n variables, boundary value analysis yield **4n+1** test cases.

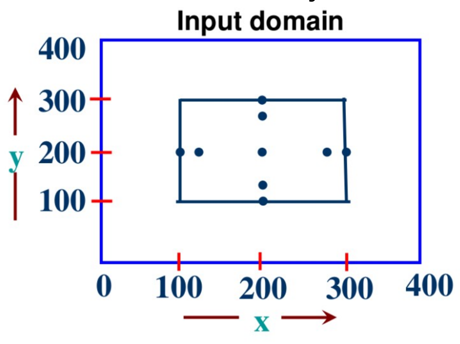


**Fig. 5:** Input domain of two variables x and y with boundaries [100,300] each

Example

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

**Solution**

Quadratic equation will be of type:

$ax^2 + bx + c = 0$

Roots are real if $(b^2 - 4ac) > 0$

Roots are imaginary if $(b^2 - 4ac) < 0$

Roots are equal if $(b^2 - 4ac) = 0$

Equation is not quadratic if $a = 0$

The boundary value test cases are:

| ***Test Case*** | ***a*** | ***b*** | ***c*** | ***Expected output*** |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not Quadratic |
| 2 | 1 | 50 | 50 | Real Roots |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 99 | 50 | 50 | Imaginary Roots |
| 5 | 100 | 50 | 50 | Imaginary Roots |
| 6 | 50 | 0 | 50 | Imaginary Roots |
| 7 | 50 | 1 | 50 | Imaginary Roots |
| 8 | 50 | 99 | 50 | Imaginary Roots |
| 9 | 50 | 100 | 50 | Equal Roots |
| 10 | 50 | 50 | 0 | Real Roots |
| 11 | 50 | 50 | 1 | Real Roots |
| 12 | 50 | 50 | 99 | Imaginary Roots |
| 13 | 50 | 50 | 100 | Imaginary Roots |

**Robustness Testing**

It is nothing but the extension of boundary value analysis. the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum. This type of testing is common in electric and electronic circuits. This form of boundary value analysis is called robustness testing and is shown in fig. 8.6

Hence total test cases in robustness testing are 6n+1, where n is the number of input variables. So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200).

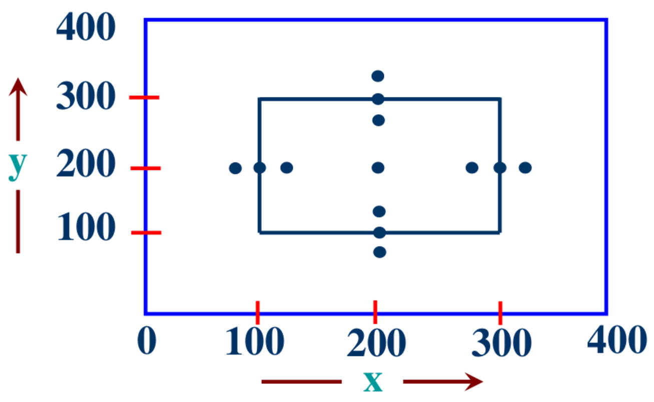


**Fig. 8.6:** Robustness test cases for two variables x and y with range [100,300] each

**Worst-case Testing**

In it more than one variable has an extreme value. It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of n variables generate **$5^n$** test cases as opposed to 4n+1 test cases for boundary value analysis. Our two variables example will have **$5^2$** =25 test cases and are given in table 1.

Table 1: Worst cases test inputs for two variables example

| Test case number | Inputs | | Test case number | Inputs | |
|---|---|---|---|---|---|
| | *x* | *y* | | *x* | *y* |
| 1 | 100 | 100 | 14 | 200 | 299 |
| 2 | 100 | 101 | 15 | 200 | 300 |
| 3 | 100 | 200 | 16 | 299 | 100 |
| 4 | 100 | 299 | 17 | 299 | 101 |
| 5 | 100 | 300 | 18 | 299 | 200 |
| 6 | 101 | 100 | 19 | 299 | 299 |
| 7 | 101 | 101 | 20 | 299 | 300 |
| 8 | 101 | 200 | 21 | 300 | 100 |
| 9 | 101 | 299 | 22 | 300 | 101 |
| 10 | 101 | 300 | 23 | 300 | 200 |
| 11 | 200 | 100 | 24 | 300 | 299 |
| 12 | 200 | 101 | 25 | 300 | 300 |
| 13 | 200 | 200 | -- | | |

## 5.3.2 Equivalence Class Testing

Equivalence partitioning is a technique of software testing in which input data is divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior. If a condition of one partition is true, then the condition of another equal partition must also be true, and if a condition of one partition is false, then the condition of another equal partition must also be false. The principle of equivalence partitioning is, test cases should be designed to cover each partition at least once. Each value of every equal partition must exhibit the same behavior as other.

The equivalence partitions are derived from requirements and specifications of the software. The advantage of this approach is, it helps to reduce the time of testing due to a smaller number of test cases from infinite to finite. It is applicable at all levels of the testing process.

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2.Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence classes.

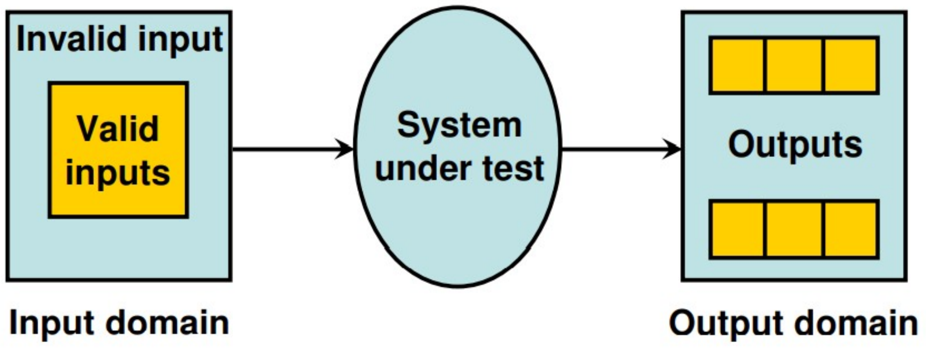


Fig. 7: Equivalence partitioning

Most of the time, equivalence class testing defines classes of the input domain.However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Identify the equivalence class test cases for output and input domains.

**Solution**

Output domain equivalence class test cases can be identified as follows:

O1={<a,b,c>:Not a quadratic equation if a = 0}

O1={<a,b,c>:Real roots if ($b^2$-4ac)>0}

O1={<a,b,c>:Imaginary roots if ($b^2$-4ac)<0}

O1={<a,b,c>:Equal roots if (b2-4ac)=0}`
The number of test cases can be derived form above relations and shown below.

| Test case | a | b | c | Expected output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | 1 | 50 | 50 | Real roots |
| 3 | 50 | 50 | 50 | Imaginary roots |
| 4 | 50 | 100 | 50 | Equal roots |

We may have another set of test cases based on input domain.
I1= {a: a = 0}
I2= {a: a < 0}
I3= {a: 1 ≤ a ≤ 100}
I4= {a: a > 100}
I5= {b: 0 ≤ b ≤ 100}
I6= {b: b < 0}
I7= {b: b > 100}
I8= {c: 0 ≤ c ≤ 100}
I9= {c: c < 0}
I10={c: c > 100}

| Test Case | a | b | c | Expected output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | -1 | 50 | 50 | Invalid input |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 101 | 50 | 50 | invalid input |
| 5 | 50 | 50 | 50 | Imaginary Roots |
| 6 | 50 | -1 | 50 | invalid input |
| 7 | 50 | 101 | 50 | invalid input |
| 8 | 50 | 50 | 50 | Imaginary Roots |
| 9 | 50 | 50 | -1 | invalid input |
| 10 | 50 | 50 | 101 | invalid input |

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

### 5.3.3 Decision Table Based Testing

Decision table is a brief visual representation for specifying which actions to perform depending on given conditions. Decision tables have been used to represent and analyse complex logical relationships since early 1960's.

There are four portions of a decision table namely, Condition Stub, Action stub, Condition Entries and Action Entries. When conditions c1, c2, c3 are all true, actions a1 and a2 occur. When conditions c1 and c2 are true and c3 is false, actions a1 and a3 occur. The decision tables in which all entries are binary are called limited entry decision tables. If conditions are allowed to have several values, the resulting tables are called Extended Entry Decision tables.

| Condition Stub | | Entry | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | $C_1$ | True | | | | False | | |
| | $C_2$ | True | | False | | True | | False |
| | $C_3$ | True | False | True | False | True | False | --- |
| Action Stub | $a_1$ | X | X | | | X | | |
| | $a_2$ | X | | X | | | X | |
| | $a_3$ | | X | | | X | | |
| | $a_4$ | | | | X | | X | X |

Table 2: Decision table terminology

**Test case design**

To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs. Sometimes, conditions end up referring to equivalence class of inputs, and actions refers to major functional processing portions of the item being tested. The rules are then interpreted as test cases. There are several techniques that produce decision tables that are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible.

| $C_1$:x,y,z are sides of a triangle? | N | Y | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $C_2$:x = y? | -- | Y | | | | N | | | |
| $C_3$:x = z? | -- | Y | | N | | Y | | N | |
| $C_4$:y = z? | -- | Y | N | Y | N | Y | N | Y | N |
| $a_1$: Not a triangle | X | | | | | | | | |
| $a_2$: Scalene | | | | | | | | | X |
| $a_3$: Isosceles | | | | | X | | X | X | |
| $a_4$: Equilateral | | X | | | | | | | |
| $a_5$: Impossible | | | X | X | | X | | | |

Table 3: Decision table for triangle problem

## 5.3.4 Cause Effect Graphing Technique

Cause Effect Graphing based technique is a technique in which a graph is used to represent the situations of combinations of input conditions. The graph is then converted to a decision table to obtain the test cases. Cause-effect graphing technique is used because boundary value analysis and equivalence class partitioning methods do not consider the combinations of input conditions. But since there may be some critical behaviour to be tested when some combinations of input conditions are considered, that is why cause-effect graphing technique is used.

**Steps used in deriving test cases using this technique are:**

**1. Division of specification:**

Since it is difficult to work with cause-effect graphs of large specifications as they are complex, the specifications are divided into small workable pieces and then converted into cause-effect graphs separately.

**2. Identification of cause and effects:**

This involves identifying the causes(distinct input conditions) and effects(output conditions) in the specification.

**3. Transforming the specifications into a cause-effect graph:**

The causes and effects are linked together using Boolean expressions to obtain a cause-effect graph. Constraints are also added between causes and effects if possible.

**4. Conversion into decision table:**

The cause-effect graph is then converted into a limited entry decision table.

**5. Deriving test cases:**

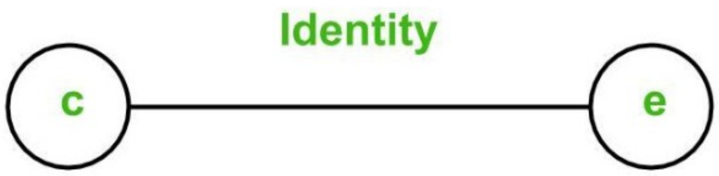
Each column of the decision-table is converted into a test case.

**Basic Notations used in Cause-effect graph:**

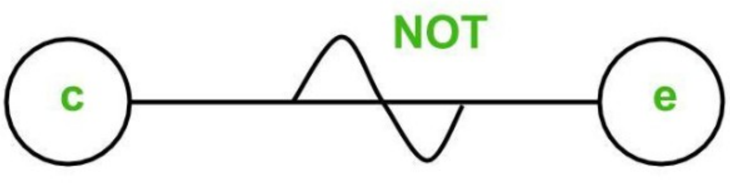
Here **c** represents cause and **e** represents effect.

The following notations are always **used between a cause and an effect:**

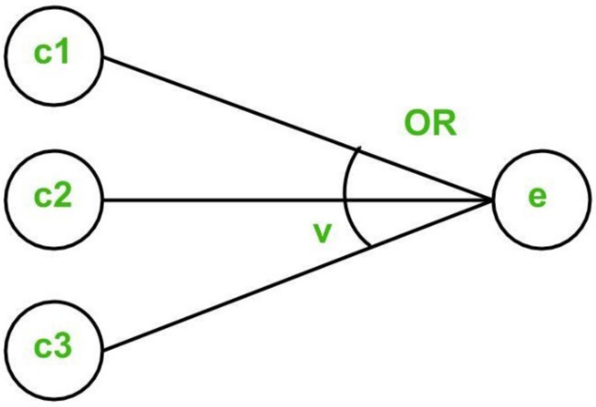
1. **Identity Function:** if c is 1, then e is 1. Else e is 0.



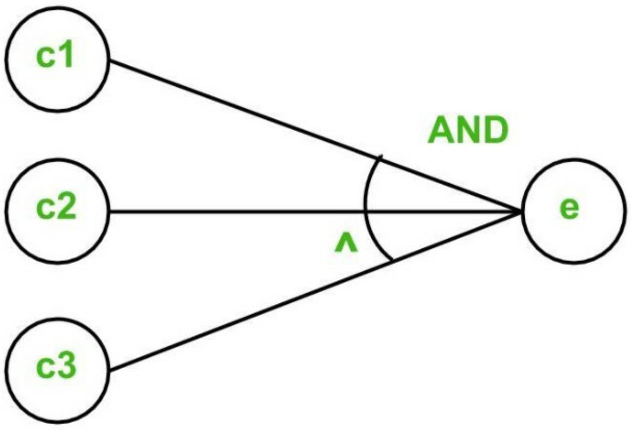
2. **NOT Function:** if c is 1, then e is 0. Else e is 1.



3. **OR Function:** if c1 or c2 or c3 is 1, then e is 1. Else e is 0.



4. **AND Function:** if both c1 and c2 and c3 is 1, then e is 1. Else e is 0.



Myers explained this effectively with following example. “The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message x is issued. If the character in column 2 is not a digit, message y is issued”.

The causes are
c1: character in column 1 is A
c2: character in column 1 is B
c3: character in column 2 is a digit
and the effects are
e1: update made
e2: message x is issued
e3: message y is issued

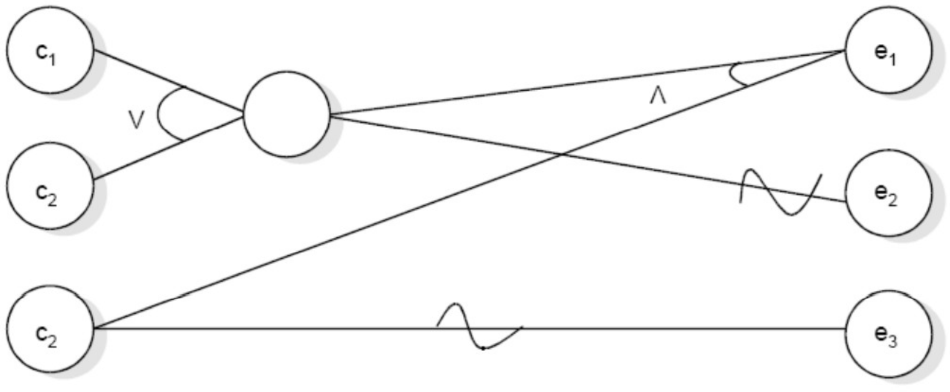
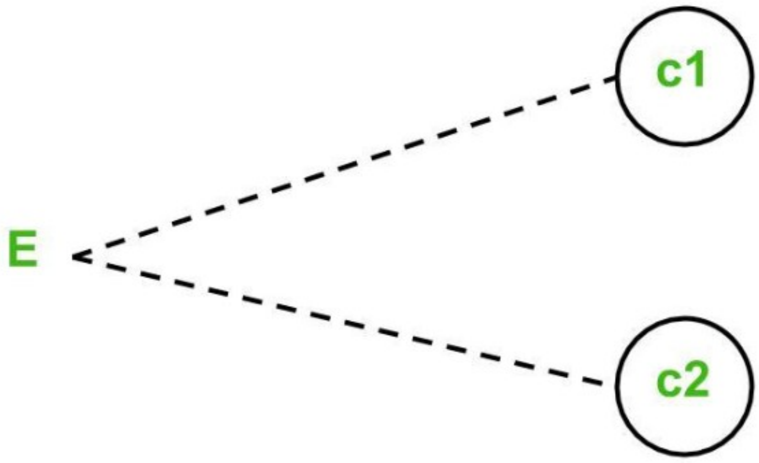


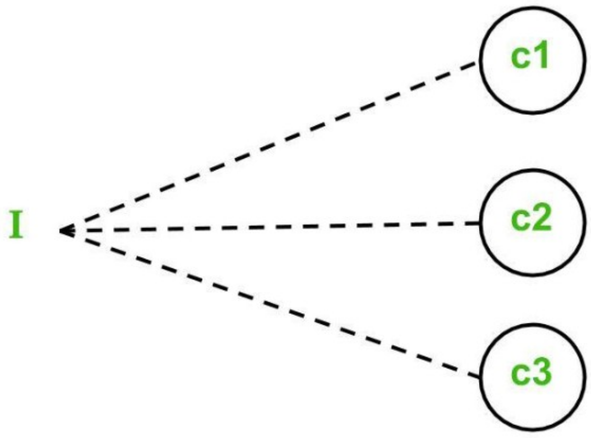
Fig. 9: Sample cause effect graph

To represent some impossible combinations of causes or impossible combinations of effects, constraints are used. The following constraints are used in cause-effect graphs:

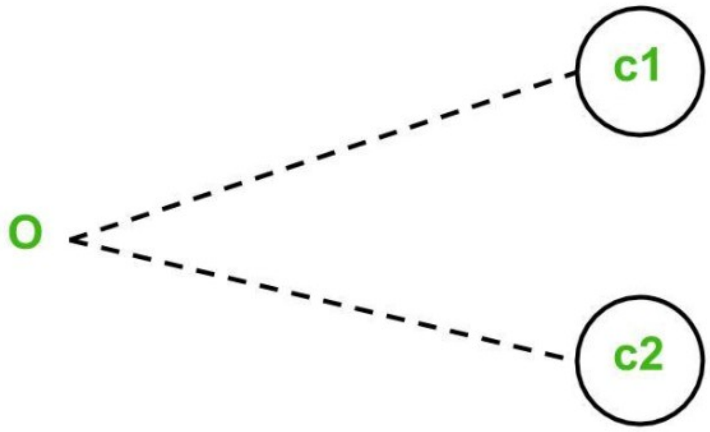
1. **Exclusive constraint or E-constraint:** This constraint exists between causes. It states that either c1 or c2 can be 1, i.e., c1 and c2 cannot be 1 simultaneously.



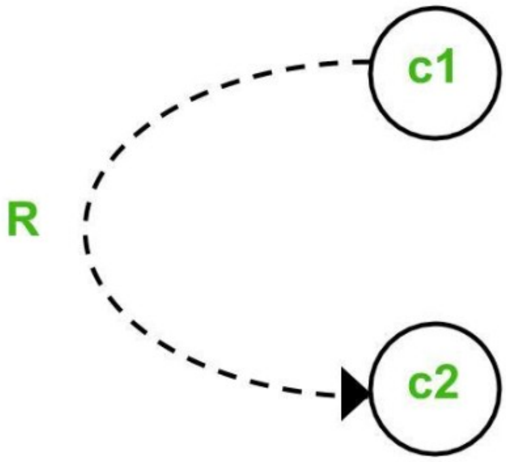
2. **Inclusive constraint or I-constraint:** This constraint exists between causes. It states that atleast one of c1, c2 and c3 must always be 1, i.e., c1, c2 and c3 cannot be 0 simultaneously.



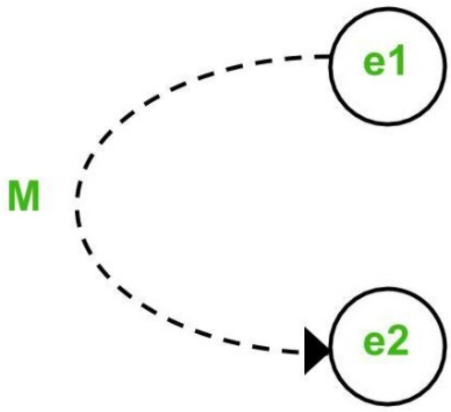
3. **One and Only One constraint or O-constraint:** This constraint exists between causes. It states that one and only one of c1 and c2 must be 1.



4. **Requires constraint or R-constraint:** This constraint exists between causes. It states that for c1 to be 1, c2 must be 1. It is impossible for c1 to be 1 and c2 to be 0.



5. **Mask constraint or M-constraint:** This constraint exists between effects. It states that if effect e1 is 1, the effect e2 is forced to be 0.



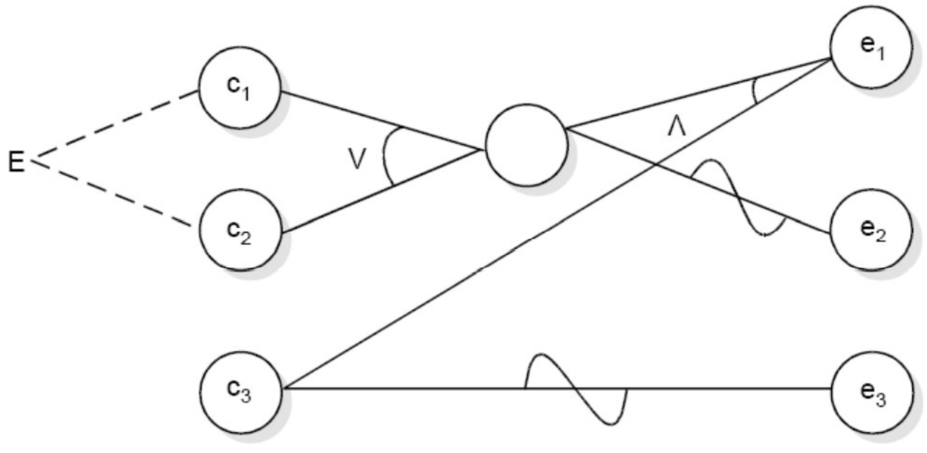


**Fig. 12 : Sample cause effect graph with exclusive constraint**

## 5.4 STRUCTURAL TESTING

Structural or white box testing permits us to examine the internal structure of the program. In using this strategy, we derive test cases from an examination of the program's logic.

The white box testing which examine the code and testing when the program is in running is called dynamic white box testing. If we want to test the program without running it, it is called static white box testing.

Different types of structural testing

1.Path Testing 2.Graph Matrices 3. Data flow Testing

### 5.4.1 Path Testing

It is most applicable to new software for module testing or unit testing. It requires complete knowledge of the program's structure and used by developers to unit test their own code.

This type of testing involves:

a) Generating a set of paths that will cover every branch in the program.

b) Finding a set of test cases that will execute every path in this set of program paths.

**Flow graph:-** Flow graph generation is the first step in path testing. It is used to analyse the control flow of a program. It is generated from the code of program. It is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.

Suppose i and j represents nodes, then there will be an edge from node i to node j if the statement corresponding to j is executed immediately after the statement corresponding to i.

The basic constructs of flow graph are,

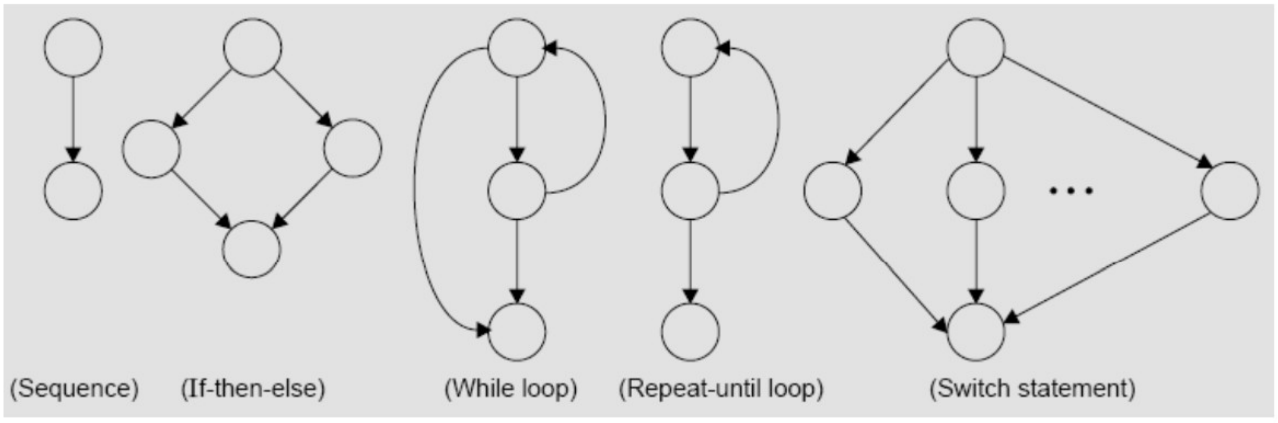


**Fig. 14: The basic construct of the flow graph**

**DD path graph:-** The second step of path testing is to draw a DD path graph from the flow graph. The DD path graph is also known as decision to decision path graph. Here the nodes of flow graph , which are in a sequence are combined into a single node and is called decision node.

**Independent paths:-** Independent paths are finding from DD path graph. An independent path is any path through the DD path graph that introduces at least one new set of processing statements. We should execute all independent paths at least once during path testing.

Independent paths are used in order to ensure that

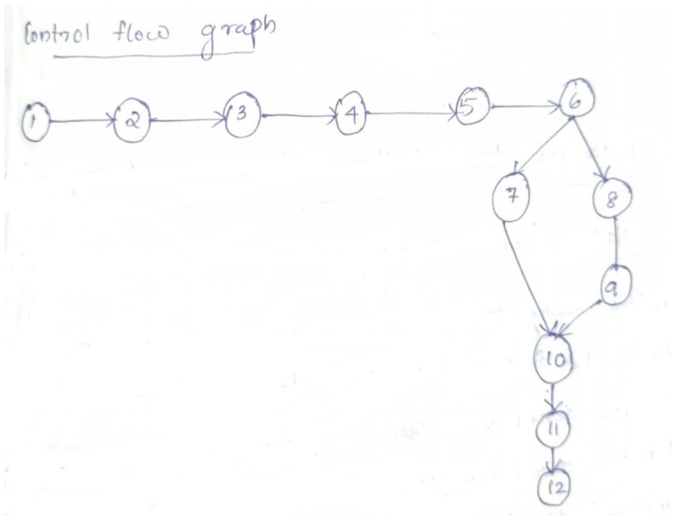
a) Every statement in the program has been executed at least once.

b) Every branch has been exercised for true and false conditions

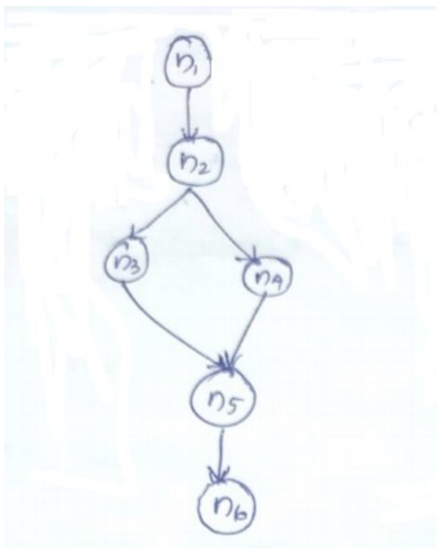
Eg: Consider the program to find the largest number.

```
#include <stdio.h>
#include <conio.h>
1.  Void main()
2.  {
3.  int a,b,c;
4.  printf("Enter the two numbers");
5.  scanf("%d %d", &a,&b);
6.  if (a>b)
7.  c=a;
8.  else
9.  c=b;
10. printf("The larger number is %d", c);
11. getch();
12. }
```

Draw the flow graph and DD path graph. Also find independent paths from DD path graph.



*Flow Graph*



*DD path graph*

**DD path graph**

| Flow graph nodes | DD path graph corresponding nodes |
|---|---|
| 1-5 | n1 |
| 6 | n2 |
| 7 | n3 |
| 8,9 | n4 |
| 10 | n5 |
| 11,12 | n6 |

**Independent paths**

n1, n2, n3, n5, n6

n1, n2, n4, n5, n6

**5.4.2 Cyclomatic Complexity**

The cyclomatic complexity is also known as structural complexity because it gives internal view of the code. This approach is used to find the number of independent paths through a program.

Cyclomatic complexity can be calculated by any of the three methods.

1. V(G) = e – n + 2P

    e – edges

    n – nodes.

    P- Connected Components

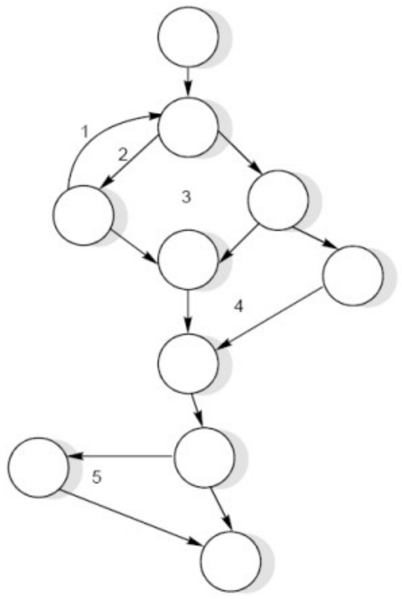
2. V(G) = π + 1

Where π is the number of predicate nodes contained in the flow graph G. The only restriction is that every predicate node should have two outgoing edges i.e., one for "true" condition and another for "false" condition.

3. Cyclomatic complexity is equal to the number of regions of the flow graph.

Example

Consider a flow graph given below, calculate the cyclomatic complexity by all three methods.



1. V(G) = e – n + 2P

    = 13 – 10 + 2 = 5

2. V(G) = π + 1

    = 4 + 1 = 5

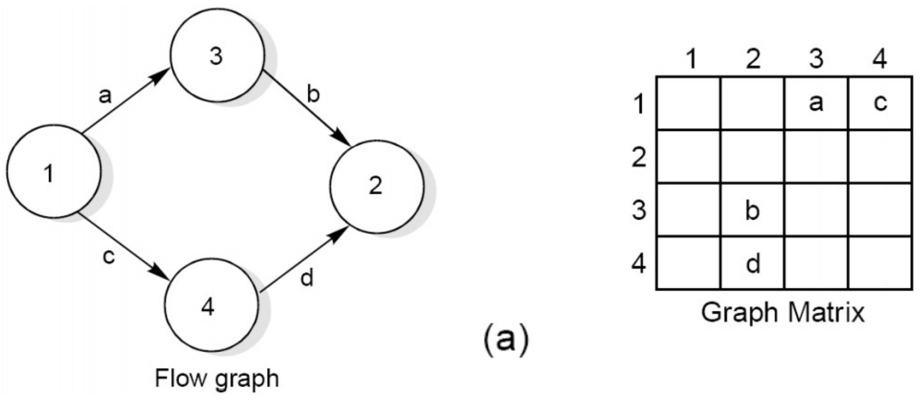
3. V(G) = number of regions

    = 5

Therefore, complexity value of a flow graph in above Fig. is 5.

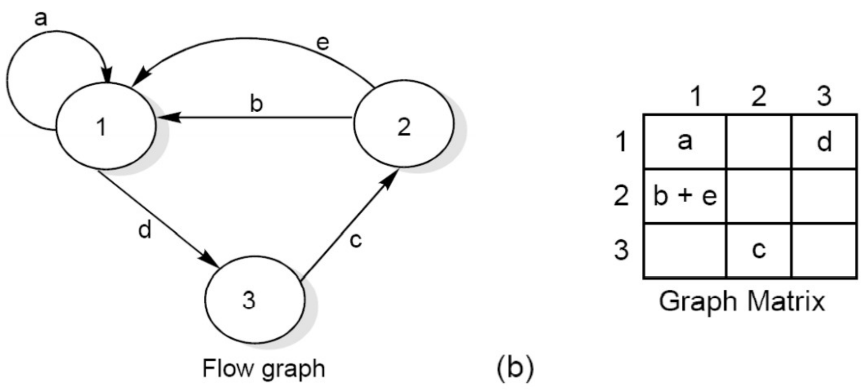
### 5.4.3 Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Graph matrix is the tabular representation of a flow graph.



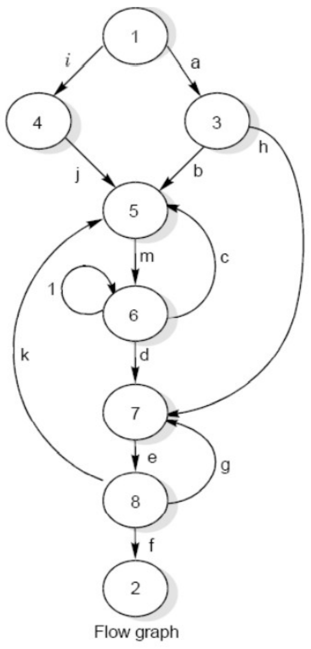
| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | a | c |
| 2 | | | | |
| 3 | | b | | |
| 4 | | d | | |

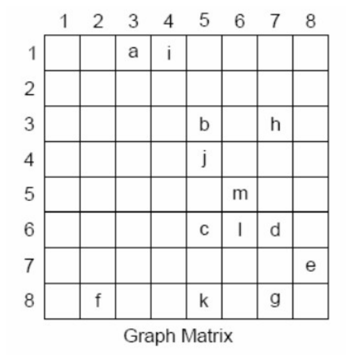
Fig. 24 (a): Flow graph and graph matrices



| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | a | | d |
| 2 | b + e | | |
| 3 | | c | |

Fig. 24 (b): Flow graph and graph matrices





|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 |   |   | a | i |   |   |   |   |
| 2 |   |   |   |   |   |   |   |   |
| 3 |   |   |   |   | b |   | h |   |
| 4 |   |   |   |   | j |   |   |   |
| 5 |   |   |   |   |   | m |   |   |
| 6 |   |   |   |   | c | l | d |   |
| 7 |   |   |   |   |   |   |   | e |
| 8 |   | f |   |   | k |   | g |   |

Graph Matrix

(c)

Fig. 24 (c): Flow graph and graph matrices

If we assign weight to each entry in the graph matrix it can be used for evaluating useful information required during testing. If there is a connection then the weight is 1, otherwise 0. A matrix with such weights is called a connection matrix. The connection matrix is used to find cyclomatic complexity.

The connection matrix for Fig.24 (c) is obtained by replacing each entry with 1, if there is a link and 0 if there is no link. As usual to reduce clutter we do not write down 0 entries and this matrix is shown in Fig

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Connections |
|---|---|---|---|---|---|---|---|---|---|
| 1 |   |   | 1 | 1 |   |   |   |   | 2 – 1 = 1 |
| 2 |   |   |   |   |   |   |   |   |   |
| 3 |   |   |   |   | 1 |   | 1 |   | 2 – 1 = 1 |
| 4 |   |   |   |   | 1 |   |   |   | 1 – 1 = 0 |
| 5 |   |   |   |   |   | 1 |   |   | 1 – 1 = 0 |
| 6 |   |   |   |   | 1 | 1 | 1 |   | 3 – 1 = 2 |
| 7 |   |   |   |   |   |   |   | 1 | 1 – 1 = 0 |
| 8 |   | 1 |   |   | 1 |   | 1 |   | 3 – 1 = 2 |
|   |   |   |   |   |   |   |   |   | **6 + 1 = 7** |

Fig. 25 : Connection matrix of flow graph shown in Fig. 24 (c)

Following are the steps to compute the cyclomatic complexity:

- Count the number of 1s in each row and write it in the end of the row
- Subtract 1 from this count for each row (Ignore the row if its count is 0)
- Add the count of each row calculated previously
- Add 1 to this total count
- The final sum in Step 4 is the cyclomatic complexity of the control flow graph

### 5.4.4 Data Flow Testing

Data Flow Testing is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.

It is concerned with:

i. Statements where variables receive values.

ii. Statements where these values are used or referenced

As we know, variables are defined and referenced throughout the program. We
may have few define/ reference anomalies:

I. A variable is defined but not used/ referenced.
II. A variable is used but never defined.
III. A variable is defined twice before it is used.

**Advantages of Data Flow Testing:**

Data Flow Testing is used to find the following issues-

- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.

**Disadvantages of Data Flow Testing**

- Time consuming and costly process
- Requires knowledge of programming languages

**Definitions**

The definitions refer to a program P that has a program graph G(P) and a set of program variables V. The G(P) has a single entry node and a single exit node. The set of all paths in P is PATHS(P)

**Defining Node:** Node n ϵ G(P) is a defining node of the variable v ϵ V, written as DEF (v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

**Usage Node:** Node n ϵ G(P) is a usage node of the variable v ϵ V, written as USE (v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.

**Definition use:** A definition use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some v ϵ V, there are define and usage nodes DEF(v, m) and USE(v,n) such that m and n are the initial and final nodes of the path.

**Definition clear:** A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n), such that no other node in the path is a defining node of v.

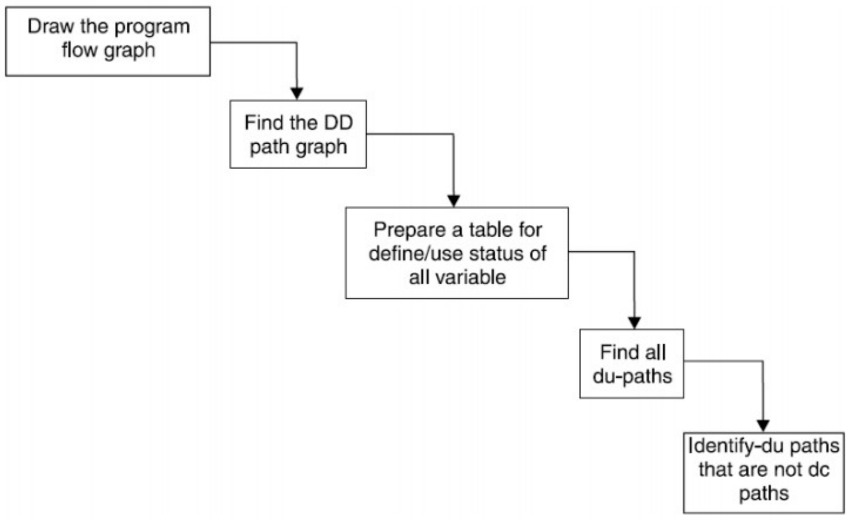

Draw the program flow graph
Find the DD path graph
Prepare a table for define/use status of all variable
Find all du-paths
Identify-du paths that are not dc paths

**Fig. 27 : Steps for data flow testing**

Consider the program given in Fig. 20 for the classification of a triangle. Its input is a triple of positive integers (say a,b,c) from the interval [1,100]. The output may be:

[Scalene, Isosceles, Equilateral, Not a triangle, Invalid inputs].

Find all du-paths and identify those du-paths that are definition clear.

```
#include <stdio.h>
#include <conio.h>
    int main()
    {
      int a,b,c,validInput=0;
      printf("Enter the side 'a' value: ");
      scanf("%d",&a);
      printf("Enter the side 'b' value: ")
      scanf("%d",&b);
      printf("Enter the side 'c' value:");
      scanf("%d",&c);
      if ((a > 0) && (a <= 100) && (b > 0) && (b <= 100) && (c > 0)
          && (c <= 100)) {
        if ( (a + b) > c) && ((c + a) > b) && ((b + c) > a)) {
          validInput = 1;
        }
      }
      else {
        validInput = -1;
      }
      If (validInput==1) {
        If ((a==b) && (b==c)) {
          printf("The trinagle is equilateral");
        }
        else if ( (a == b) || (b == c) || (c == a) ) {
```

```
      printf("The triangle is isosceles");
    }
    else {
      printf("The trinagle is scalene");
    }
  }
  else if (validInput == 0) {
    printf("The values do not constitute a Triangle");
  }
  else {
    printf("The inputs belong to invalid range");
  }
  getche();
  return 1;
}
```

**Fig. 20 : Code of triangle classification problem**

**Solution :**

**Flow graph of triangle problem is:**

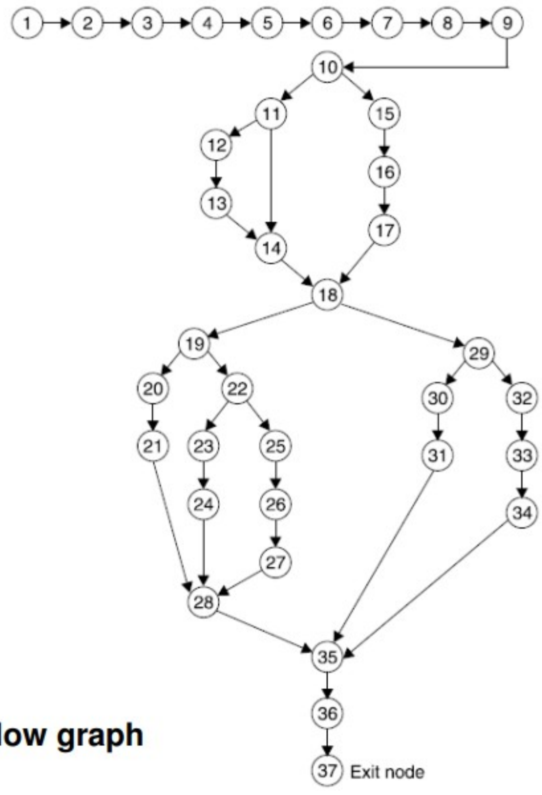


**Fig.8. 20 (a): Program flow graph**

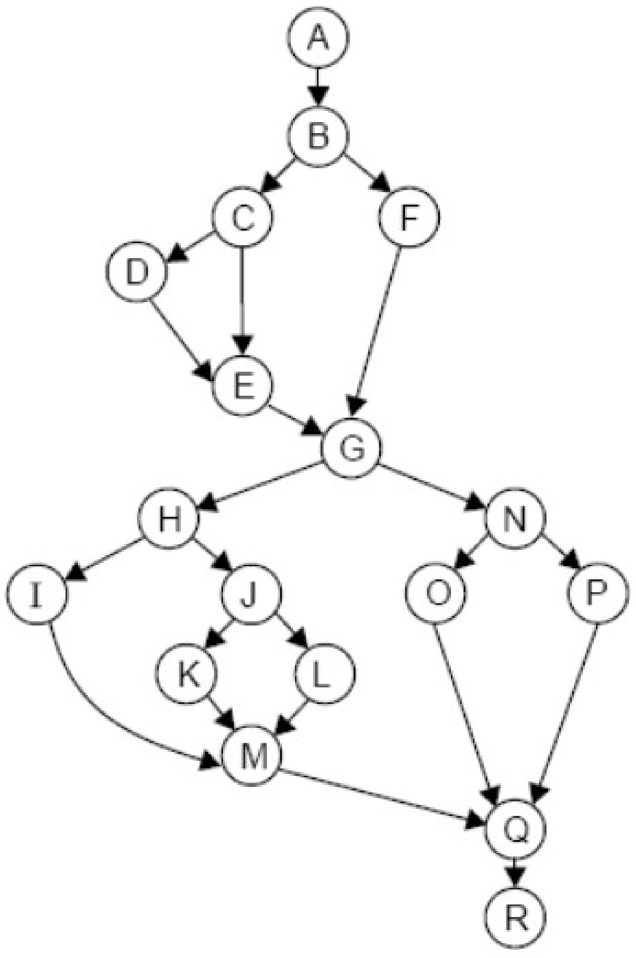
**The mapping table for DD path graph is:**

| Flow graph nodes | DD Path graph corresponding node | Remarks |
|---|---|---|
| 1 TO 9 | A | Sequential nodes |
| 10 | B | Decision node |
| 11 | C | Decision node |
| 12, 13 | D | Sequential nodes |
| 14 | E | Two edges are joined here |
| 15, 16, 17 | F | Sequential nodes |
| 18 | G | Decision nodes plus joining of two edges |
| 19 | H | Decision node |
| 20, 21 | I | Sequential nodes |
| 22 | J | Decision node |
| 23, 24 | K | Sequential nodes |
| 25, 26, 27 | L | Sequential nodes |

| Flow graph nodes | DD Path graph corresponding node | Remarks |
|---|---|---|
| 28 | M | Three edges are combined here |
| 29 | N | Decision node |
| 30, 31 | O | Sequential nodes |
| 32, 33, 34 | P | Sequential nodes |
| 35 | Q | Three edges are combined here |
| 36, 37 | R | Sequential nodes with exit node |

**Fig. 20 (b): DD Path graph**

DD Path graph is given in Fig. 20 (b)



Independent paths are:

i. ABFGNPQR
ii. ABFGNOQR
iii. ABCEGNPQR
iv. ABCDEGNOQR
v. ABFGHIMQR
vi. ABFGHJKMQR
vii. ABFGHJLMQR

**Solution**

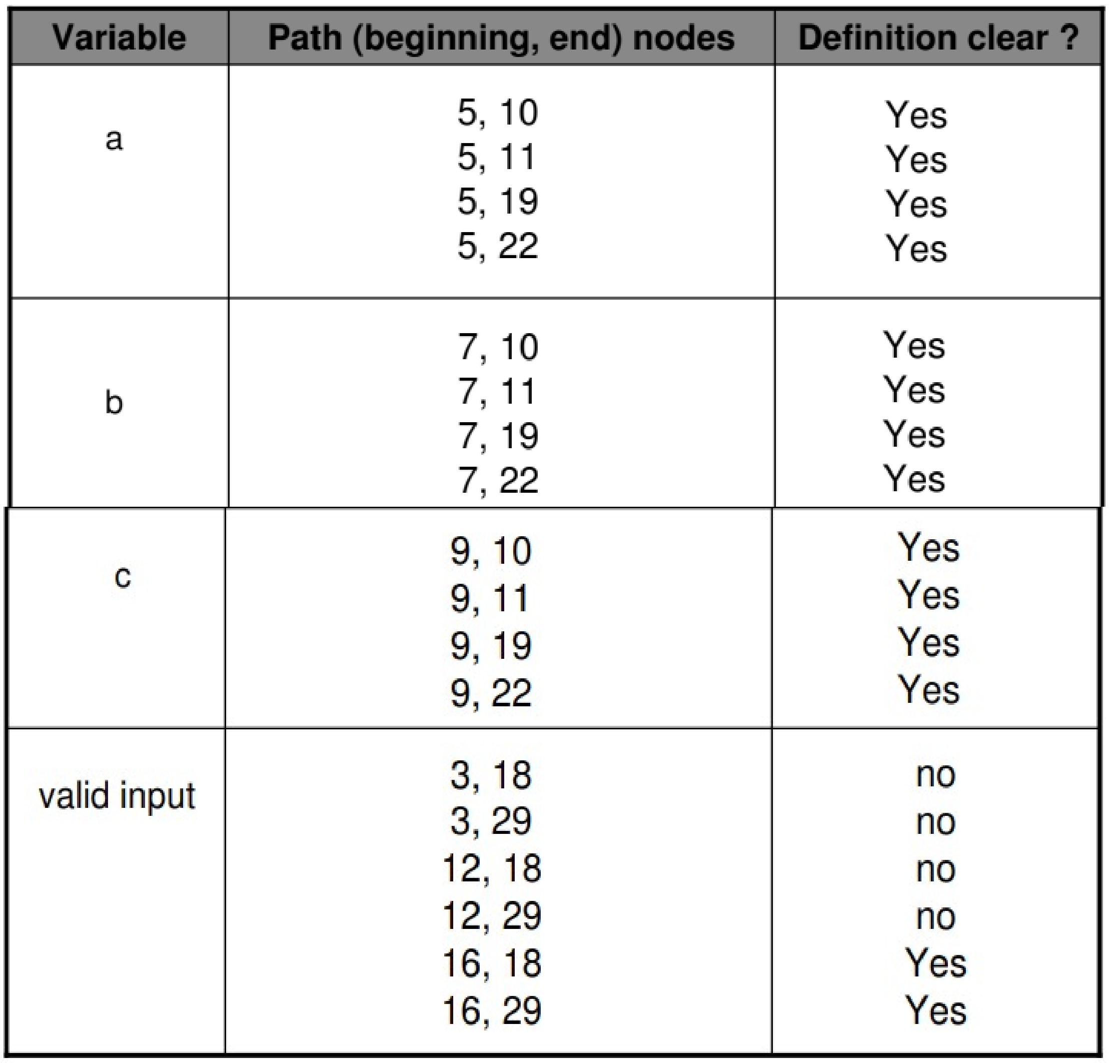
Step I: The program flow graph is given in Fig. 20 (a). The variables used in the program are a,b,c, valid input.

Step II: DD Path graph is given in Fig. 20(b). The cyclomatic complexity of this graph is 7 and thus, there are 7 independent paths.

Step III: Define/use nodes for all variables are given below:

| Variable | Defined at node | Used at node |
|---|---|---|
| a | 5 | 10, 11, 19, 22 |
| b | 7 | 10, 11, 19, 22 |
| c | 9 | 10, 11, 19, 22 |
| Validinput | 3, 12, 16 | 18, 29 |

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 20 (a).



| Variable | Path (beginning, end) nodes | Definition clear ? |
|---|---|---|
| a | 5, 10 | Yes |
| | 5, 11 | Yes |
| | 5, 19 | Yes |
| | 5, 22 | Yes |
| b | 7, 10 | Yes |
| | 7, 11 | Yes |
| | 7, 19 | Yes |
| | 7, 22 | Yes |
| c | 9, 10 | Yes |
| | 9, 11 | Yes |
| | 9, 19 | Yes |
| | 9, 22 | Yes |
| valid input | 3, 18 | no |
| | 3, 29 | no |
| | 12, 18 | no |
| | 12, 29 | no |
| | 16, 18 | Yes |
| | 16, 29 | Yes |

Hence total du-paths are 18 out of which four paths are not definition clear.

**Differences between Black Box Testing vs White Box Testing**

| # | Black Box Testing | White Box Testing |
|---|---|---|
| 1 | Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program. | White box testing is the software testing method in which internal structure is being known to tester who is going to test the software. |
| 2 | This type of testing is carried out by testers. | Generally, this type of testing is carried out by software developers. |
| 3 | Implementation Knowledge is not required to carry out Black Box Testing. | Implementation Knowledge is required to carry out White Box Testing. |
| 4 | Programming Knowledge is not required to carry out Black Box Testing. | Programming Knowledge is required to carry out White Box Testing. |
| 5 | Testing is applicable on higher levels of testing like System Testing, Acceptance testing. | Testing is applicable on lower level of testing like Unit Testing, Integration testing. |
| 6 | Black box testing means functional test or external testing. | White box testing means structural test or interior testing. |
| 7 | In Black Box testing is primarily concentrate on the functionality of the system under test. | In White Box testing is primarily concentrate on the testing of program code of the system under test like code structure, branches, conditions, loops etc. |
| 8 | The main aim of this testing to check on what functionality is performing by the system under test. | The main aim of White Box testing to check on how System is performing. |
| 9 | Black Box testing can be started based on Requirement Specifications documents. | White Box testing can be started based on Detail Design documents. |
| 10 | The Functional testing, Behavior testing, Close box testing is carried out under Black Box testing, so there is no required of the programming knowledge. | The Structural testing, Logic testing, Path testing, Loop testing, Code coverage testing, Open box testing is carried out under White Box testing, so there is compulsory to know about programming knowledge. |

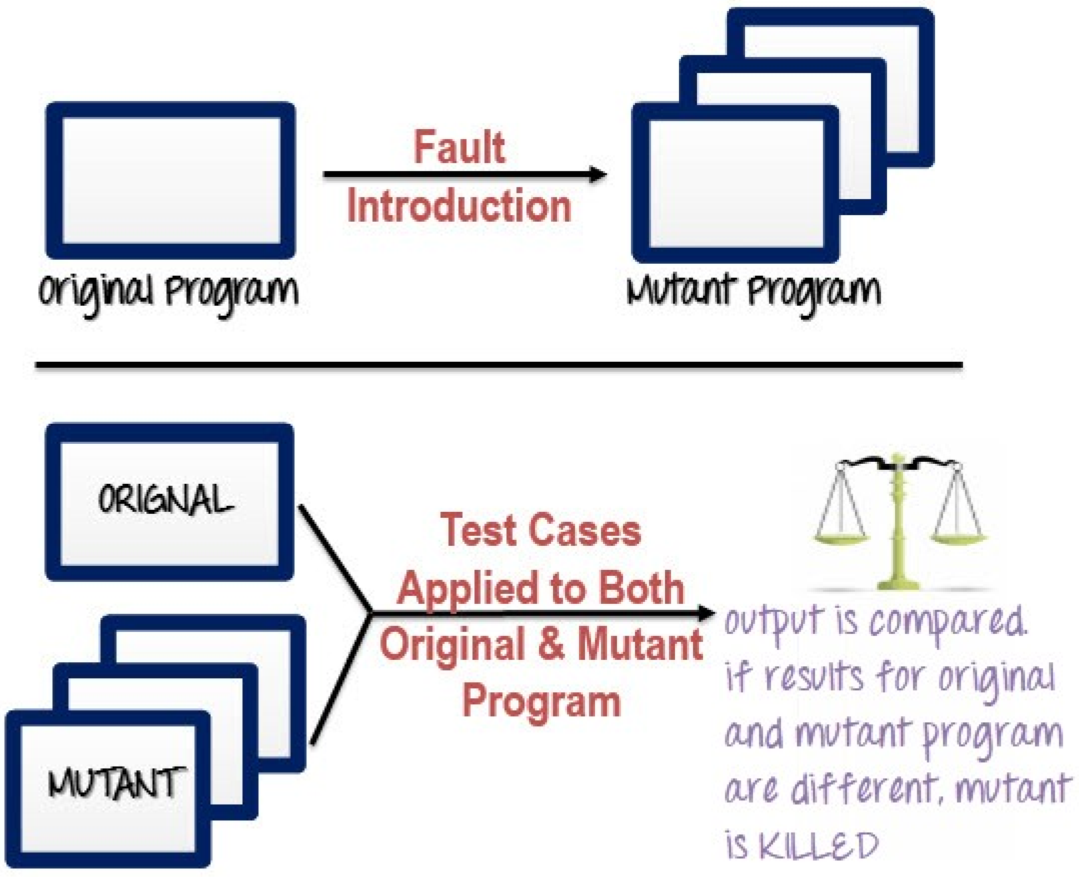
### 5.4.5 Mutation Testing

Mutation Testing is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in source code. The goal of Mutation Testing is ensuring the quality of test cases in terms of robustness that it should fail the mutated source code.

The changes made in the mutant program should be kept extremely small that it does not affect the overall objective of the program. Mutation Testing is also called Fault-based testing strategy as it involves creating a fault in the program and it is a type of White Box Testing which is mainly used for Unit Testing.

Mutation was originally proposed in 1971 but lost forever due to the high costs involved. Now, again it has picked steam and is widely used for languages such as Java and XML.

How to execute Mutation Testing?



Following are the steps to execute mutation testing(mutation analysis):

Step 1: Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.

Step 2: Test cases are applied to the original program and also to the mutant program. A Test Case should be adequate, and it is tweaked to detect faults in a program.

Step 3: Compare the results of an original and mutant program.

Step 4: If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.

Step 5: If the original program and mutant program generate the same output, Mutant is kept alive. In such cases, more effective test cases need to be created that kill all mutants.

**Types of mutation testing**

Mutation testing can be classified into three parts, which are as follows:

- Decision mutations
- value mutations
- Statement mutations

**Decision mutations**

In this type of mutation testing, we will check the design errors. And here, we will do the modification in arithmetic and logical operator to detect the errors in the program.

Like if we do the following changes in arithmetic operators:

- plus(+)→ minus(-)
- asterisk(*)→ double asterisk(**)
- plus(+)→incremental operator(i++)

Like if we do the following changes in logical operators

Exchange P > → P<, OR P>=

Now, let see one example for our better understanding:

| Original Code | Modified Code |
|---|---|
| if(p >q)<br>r= 5;<br>else<br> r= 15; | if(p < q)<br> r = 5;<br>else<br> r = 15; |

**Value mutations**

In this, the values will modify to identify the errors in the program, and generally, we will change the following:

- Small value à higher value
- Higher value à Small value.

For Example:

| Original Code | Modified Code |
|---|---|
| int add =9000008;<br>int p = 65432;<br>int q =12345;<br>int r = (p+ q); | int mod = 9008;<br>int p = 65432;<br>int q =12345;<br>int r= (p + q); |

**Statement Mutations**

Statement mutations means that we can do the modifications into the statements by removing or replacing the line as we see in the below example:

| Original Code | Modified Code |
|---|---|
| if(p * q)<br> r = 15;<br>else<br> r = 25; | if(p* q)<br> s = 15;<br>else<br> s = 25; |

In the above case, we have replaced the statement r=15 by s=15, and r=25 by s=25.

**Advantages and disadvantages of Mutation Testing**

**Advantages of Mutation Testing:**

- It brings a good level of error detection in the program.
- It discovers ambiguities in the source code.

**Disadvantages of Mutation Testing:**

- It is highly costly and time-consuming.
- It is not able for Black Box Testing.

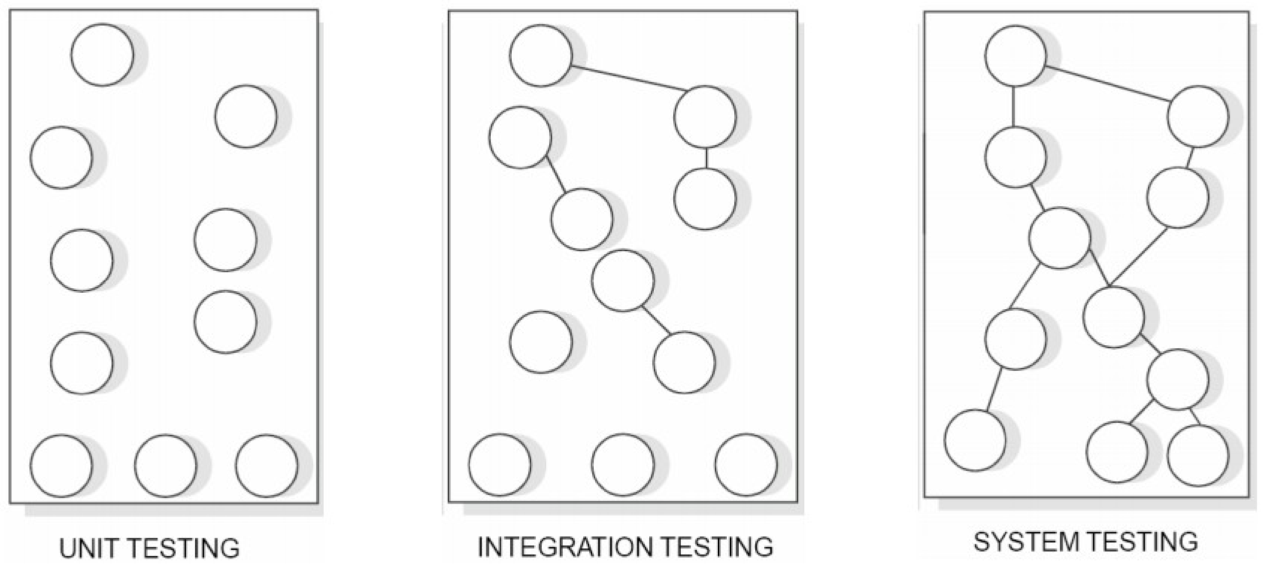
## 5.5 Levels of Testing

There are 3 levels of testing:

i. Unit Testing

ii. Integration Testing

iii. System Testing



### 5.5.1 Unit Testing

It is the process of taking a module and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specifications and design of the module.

There are number of reasons in support of unit testing than testing the entire product.

1. The size of a single small module is small enough that we can locate an error easily.

2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
3. Confusing interactions of multiple errors in different parts of software are eliminated.

There are problems when running a module in isolation. They are

a) there is no calling statement to call it

b) this module is not calling any other module

c) the intermediate values obtained during execution can not be given as output

To overcome these problems we use driver routine which has some code to call the testing module. And we use simple stubs that has some code and called by the testing module. Also we insert some output statements. This overhead code, called scaffolding represents effort that is important to testing , but does not appear in the delivered product.

The white box testing approaches are used for unit testing.

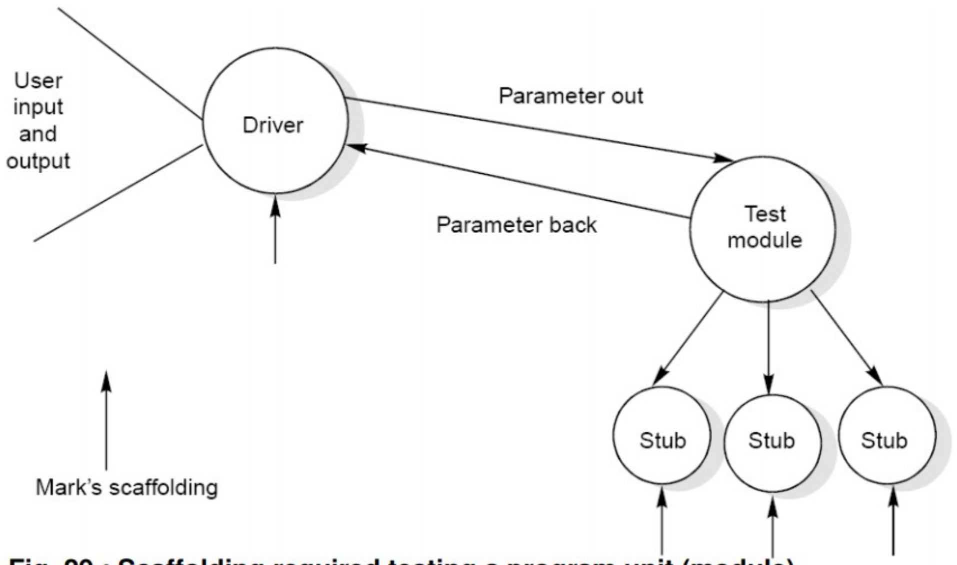


**Fig. 29 : Scaffolding required testing a program unit (module)**

**5.5.2 Integration Testing**

The purpose of **unit testing** is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between module is also correct, and for this reason integration testing must be performed.

The purpose of **integration testing** is to determine the interface between modules is correct. Whether parameters on both sides match as to types, permissible ranges, meaning and utilization.

Top-down integration proceeds down the hierarchy, adding one module at a time until an entire tree level is integrated, and thus it eliminates the need for drivers. The bottom up strategy works similarly from the bottom and has no need of stubs. The sandwich strategy runs from top and bottom concurrently. These approaches are shown below.

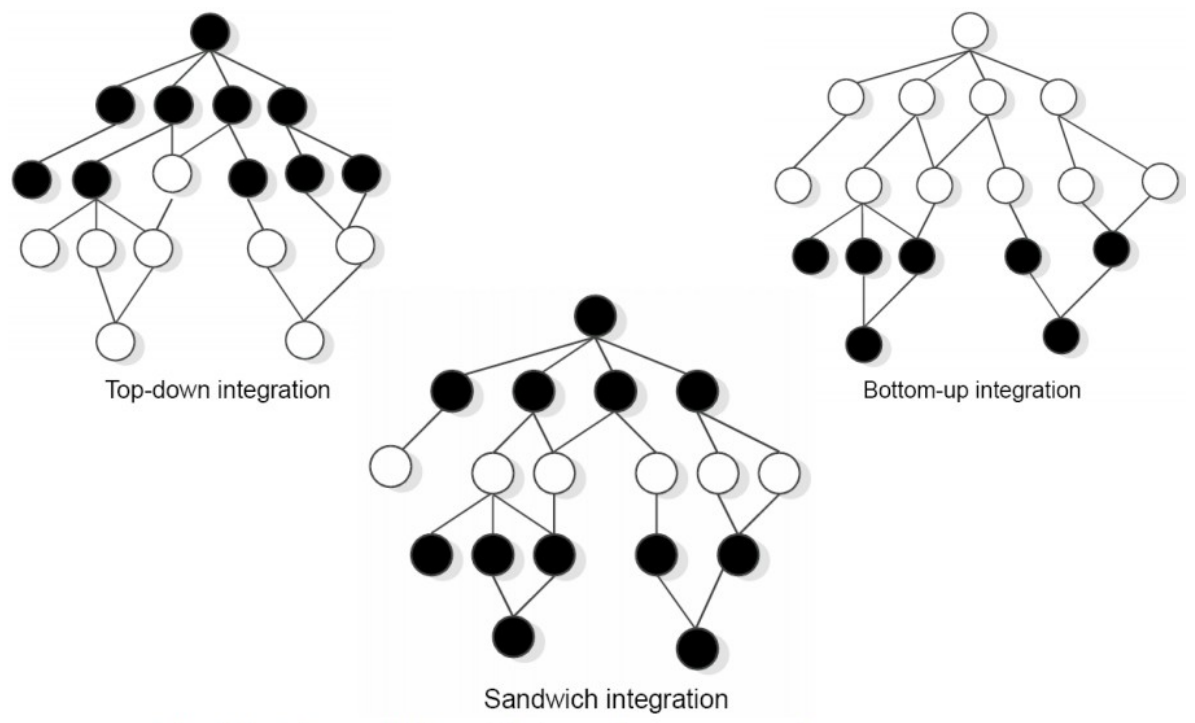


**Fig. 30 : Three different integration approaches**

**5.5.3 System Testing**

System testing involves the testing of the entire system, whereas software is a part of the system. This is essential to build confidence in the developers before software is delivered to the customer or released in the market.

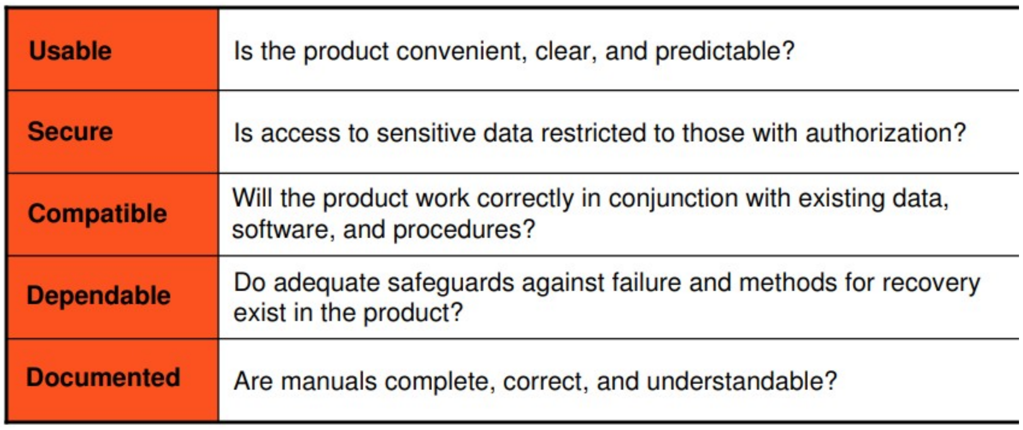
Petschenik gives some guidelines for choosing test cases during system testing.

The first is that testing the system's capabilities is more important than testing its components. This implies that failures that are catastrophic should be looked for whereas failures that are merely annoying need not worry us. That is, a user can deal with a badly formatted report, but cannot deal with unavailability of the report.

Petschenik's second rule is that testing the usual is more important than testing the exotic. This can be accomplished by subjecting the software to the kind of use described in the operational profile.

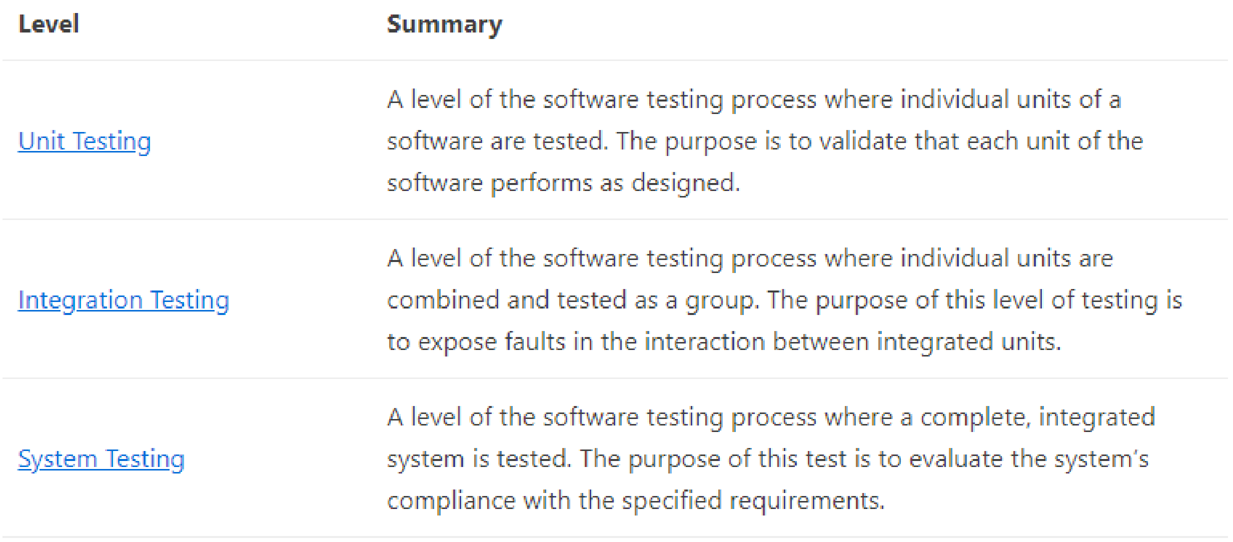
Third, if we are testing after modification of an existing product, we should test old capabilities rather than new ones. The user is not depending on the new functions of the software, and would not paralysed if these were not rigid. But a failure in the old functionality could do just that paralyse the user's entire operation.

Attributes of software to be tested during system testing are shown below in the table.

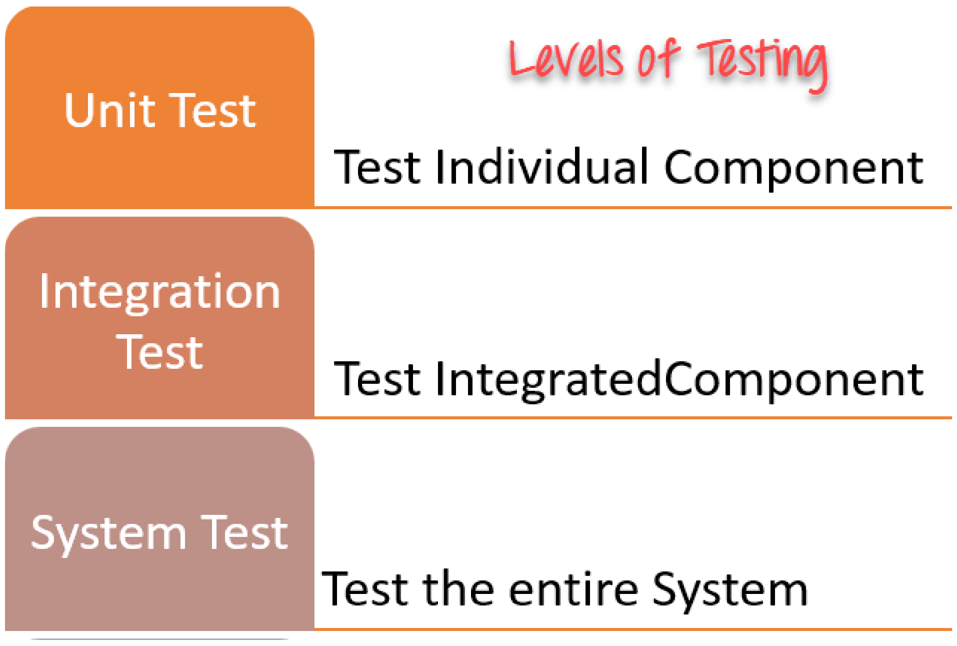


| | |
|---|---|
| **Usable** | Is the product convenient, clear, and predictable? |
| **Secure** | Is access to sensitive data restricted to those with authorization? |
| **Compatible** | Will the product work correctly in conjunction with existing data, software, and procedures? |
| **Dependable** | Do adequate safeguards against failure and methods for recovery exist in the product? |
| **Documented** | Are manuals complete, correct, and understandable? |

**Fig. 31 : Attributes of software to be tested during system testing**



| Level | Summary |
|---|---|
| Unit Testing | A level of the software testing process where individual units of a software are tested. The purpose is to validate that each unit of the software performs as designed. |
| Integration Testing | A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. |
| System Testing | A level of the software testing process where a complete, integrated system is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements. |



## 5.6 VALIDATION TESTING

Validation testing is done after unit and integration testing. It refers to test the software as a complete product. We test the software with the perspective of the customers and ensure that the software meets the expectations of the customers.

Alpha, beta and acceptance testing are the various ways of involving customers during testing. The alpha and beta testing are used when the software is developed for anonymous customers. Compilers, operating systems, CASE tools are examples of such software. In this case potential customers are invited in the premises of the company and are requested to use the software. The process is carried out under the guidance of developers and controlled environment is provided.

In beta testing the potential customers test the software in their respective premises. It is conducted in a real environment, without any control of the developer. The developers receive the failure reports or suggestions, and may modify the code.

The acceptance testing is popular with customized software. Here, customer is available to check the software as per expectations. These tests may range from adhoc tests to well planned systematic series of tests. Its duration may be few weeks or months. the identified bugs will be fixed and improved software will be delivered to the customer.

IEEE has developed a standard (IEEE standard 1059-1993) entitled “ IEEE guide for software verification and validation “ to provide specific guidance about planning and documenting the tasks required by the standard so that the customer may write an effective plan.

Validation testing improves the quality of software product in terms of functional capabilities and quality attributes.