# Software Engineering

## Unit 4

## Software Testing

### Introduction

Once the source code has been developed, testing is required to uncover the errors before it is implemented. In order to perform software testing a series of test cases is designed. Since testing is a complex process hence, in order to make the process simpler, testing activities are broken into smaller activities. Due to this, for a project incremental testing is generally preferred. In this testing process the system is broken in set of subsystems and there subsystems are tested separately before integrating them to form the system for system testing.

### Definition of Testing

1. **According to IEEE** – "Testing means the process of analyzing a software item to detect the differences between existing and required condition (i.e. bugs) and to evaluate the feature of the software item".
2. **According to Myers** – "Testing is the process of analyzing a program with the intent of finding an error".

### Primary objectives of Software Testing

According to Glen Myers the primary objectives of testing are:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding and as yet undiscovered error.
3. A successful test is one that uncovers as yet undiscovered error.

Testing cannot show the absence of errors and defects, it can show only error and detects present. Hence, objective of testing is to design tests that systematically uncover different errors and to do so with a minimum amount of time and effort.

### ERRORS, FAULT AND FAILURE

**Error** – The term error is used to refer to the discrepancy between computed, observed or measured value and the specified value. In other terms errors can be defined as the difference between actual output of software and correct output.

**Fault** – It is a condition that causes a system to fail in performing its required function.

**Failure** – A software failure occurs if the behavior of the software is different from specified behavior. It is a stage when system becomes unable to perform a required function according to the specification mentioned.

### TEST ORACLES

A test oracle is a mechanism, different from the program itself that can used to check the correctness of the output of the program for the test cases. In order to test any program we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this test oracle is needed.

Test oracles generally use the system specification of the program to decide what the correct behavior of the program should do. In order to determine the correct behavior it is important that the behavior of the system be unambiguously specified and the specification itself should be error free.

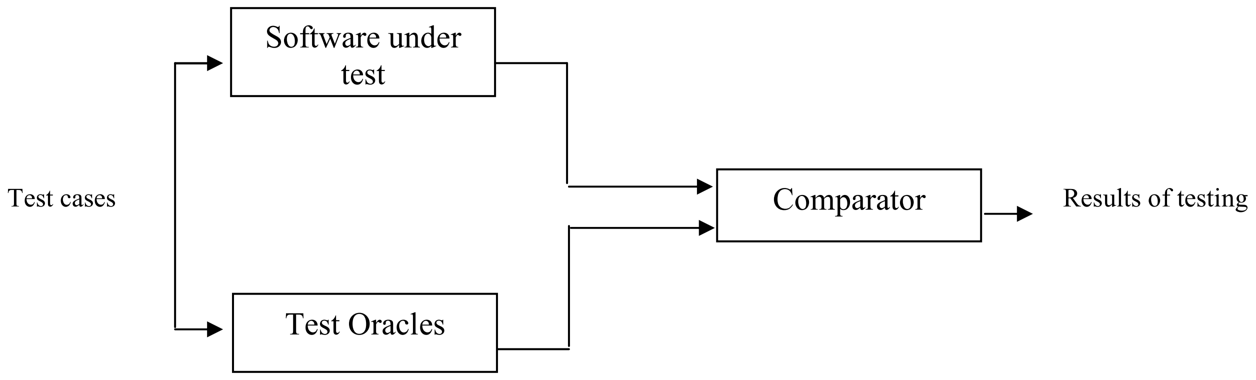


Fig (a) Test Oracles

**Testing Principles**

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggested a set of testing principles as follows

1. **All tests should be traceable to customer requirements –** According to customer's point of view the most severe defects are those that cause the program to fail to meet its requirements. Hence, the tests should be done in keeping the objective of customers requirement is must.
2. **Tests should be planned long before testing begins –** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
3. **The Pareto principle applies to software testing –** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
4. **Testing should begin "in the small" and progress toward testing "in the large" –** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. **Exhaustive testing is not possible –** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
6. **To be most effective, testing should be conducted by an independent third party –** By most effective, we mean testing that has the highest probability of finding errors. The software engineer who created the system is not the best person to conduct all tests for the software, so to be more effective third independent party should be involved on performing test.

**Attributes of Good Test**

Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

1. **A good test has a high probability of finding an error.**

   To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.
2. **A good test is not redundant.**

   Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.

3. **A good test should be "best of breed".**

   In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. **A good test should be neither too simple nor too complex.**

   Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

## Test Case Design

The test case design provides the developer with a systematic approach to test and also provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood errors in software.

Since, effort and time both is needed to execute the test cases and also machine time is needed to execute the program for those test cases also, effort is needed to evaluate the results. Hence, the fundamental goals of a testing activity are to minimize the test cases and maximize the error unfolded. According to Yamaura there is only one rule in designing test cases and it is that it should cover all features, but do not make too many test cases.

An ideal set of test cases is one that includes all the possible inputs to the program. This is called **exhaustive testing**. However, this testing is practically not possible because number of elements in the input domain is very large and testing them through one ideal set is not feasible. Hence, realistic goal for testing is to select a set of test cases that is close to the ideal.

## White Box Testing (Gloss Box Testing) (Structural Testing)

1. In this testing technique the internal logic of software components is tested.
2. It is a test case design method that uses the control structure of the procedural design test cases.
3. It is done in the early stages of the software development.
4. Using this testing technique software engineer can derive test cases that:
   a) All independent paths within a module have been exercised at least once.
   b) Exercised true and false both the paths of logical checking.
   c) Execute all the loops within there boundaries.
   d) Exercise internal data structures to ensure their validity.

**Advantages:**

1. As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.
2. The other advantage of white box testing is that it helps in optimizing the code.
3. It helps in removing the extra lines of code, which can bring in hidden defects.
4. We can test the structural logic of the software.
5. Every statement is tested thoroughly.
6. Forces test developer to reason carefully about implementation.
7. Approximate the partitioning done by execution equivalence.
8. Reveals errors in "hidden" code.

**Disadvantages:**

1. It does not ensure that the user requirements are fulfilled.
2. As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost.
3. It is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.
4. The tests may not be applicable in real world situation.
5. Cases omitted in the code could be missed out.

## <u>Black Box Testing (Behavioral Testing) (Closed box Testing)</u>

1. This technique exercises the input and output domain of the program to uncover errors in program, function, behavior and performance.
2. Software requirements are exercised using “black box” test case design technique.
3. It is done in the later stage of the software development.
4. Black box testing technique attempts to find errors related to:
   a) Missing functions or incorrect functions.
   b) Errors created due to interfaces.
   c) Errors in accessing external databases.
   d) Performance related errors.
   e) Behavior related errors.
   f) Initialization and termination errors.
5. In Black box testing the main focus is on the information domain. Test is designed for following questions:
   a) How is functionality validation testing?
   b) What class of input will make good test cases?
   c) Is the system particularly sensitive to certain input values?
   d) How are the boundaries of a data class isolated?
   e) What data rates and data volume can the system tolerate?
   f) What effects will specific combinations of data have on system operation?

**Advantages:**

1. More effective on larger units of code than glass box testing.
2. Tester needs no knowledge of implementation, including specific programming languages.
3. Tester and programmer are independent of each other.
4. Tests are done from a user's point of view.
5. Will help to expose any ambiguities or inconsistencies in the specifications.
6. Test cases can be designed as soon as the specifications are complete.

**Disadvantages:**

1. Only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever.
2. Without clear and concise specifications, test cases are hard to design.
3. There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
4. May leave many program paths untested.
5. Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone).
6. Most testing related research has been directed toward glass box testing.

**Difference between Black box and White box testing**

| No. | Black Box Testing | White Box Testing |
|---|---|---|
| 1 | This method focuses on functional requirements of the software i.e., it enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a system. | This method focuses on procedural details i.e., internal logic of a program. |
| 2 | It is not an alternative approach to white box technique rather is complementary approach that is likely to uncover a different class of errors. | It concentrates on internal logic, mainly. |
| 3 | Black box testing is applied during later stages of testing. | White box testing is performed early in the testing process. |
| 4 | It attempts to find errors in following categories:<br>(a) incorrect or missing functions.<br>(b) interface errors.<br>(c) errors in data structures or external database access.<br>(d) performance errors.<br>(e) initialization and termination errors. | White box testing attempts errors in following cases:<br>(a) internal logic of your program<br>(b) status of program. |
| 5 | It disregards control structure of procedural design (i.e., what is the control structure of our program, we do not consider here). | It uses control structure of the procedural design to derive test cases. |
| 6 | Black box testing, broadens our focus on the information domain and might be called as “testing in the large” i.e., testing bigger monolithic programs. | White box testing is “testing in small” i.e., testing small program components (e.g. modules or small group of modules). |
| 7 | Using black box testing techniques, we derive a set of test cases that satisfy following criteria:<br>(a) Test cases that reduce, (by a count that is greater than 1), the number of additional test cases that must be designed to achieve reasonable testing.<br>(b) and test cases that tell us something about the presence or absence of classes of errors rather than an error associated only with the specific tests at hand. | Using White box testing, the software engineer can derive test cases that:<br>(a) guarantee that all independent paths within a module have been exercised at least once.<br>(b) exercise all logical decisions on their true and false sides.<br>(c) execute all loops at their boundaries and within their operational bounds.<br>(d) and exercise internal data structures to ensure their validity. |
| 8 | It includes the tests that are conducted at the software interface | A close examination of procedural detail is done. |
| 9 | Are used to uncover errors. | Logical paths through the software are tested by providing test cases, that exercise specific sets of conditions or loops. |
| 10 | To demonstrate that software functions are operational i.e., input is properly accepted and output is correctly produced. Also, the integrity of external information (e.g. a data base) is maintained. | A limited set of logical paths be however examined. |

## Basic Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

**Flow Graph Notation**

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure below. Each structured construct has a corresponding flow graph symbol.

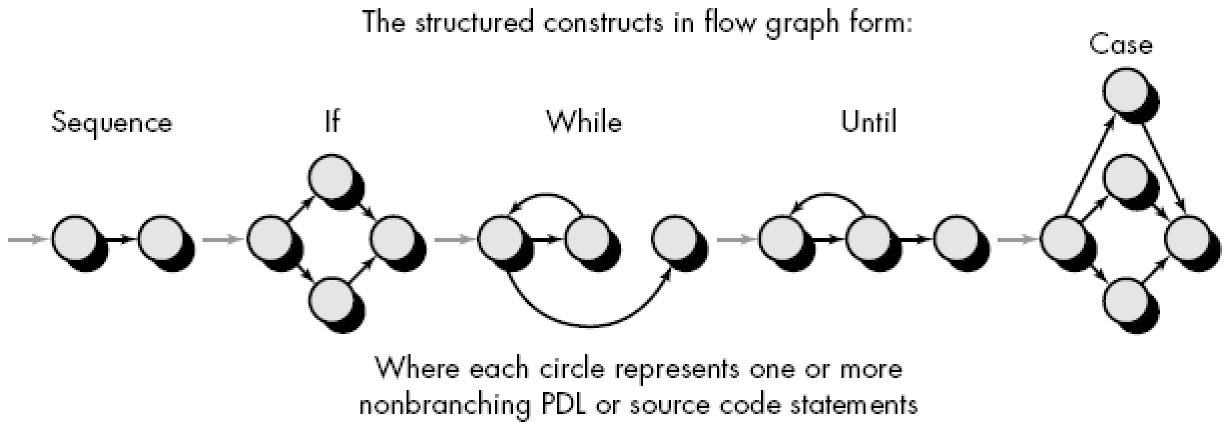

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Fig (b) Flow Graph Notation

To illustrate the use of a flow graph, we consider the procedural design representation in Figure (c) below. Here, a flowchart is used to depict program control structure.

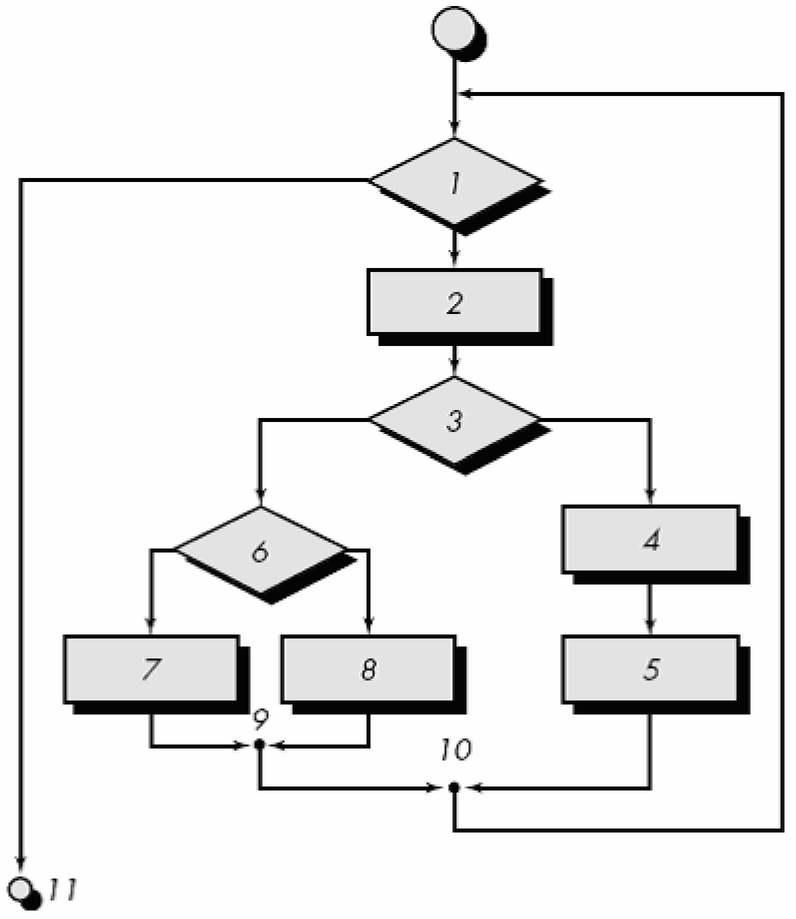


Fig (c) Flow Chart

Figure (d) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).

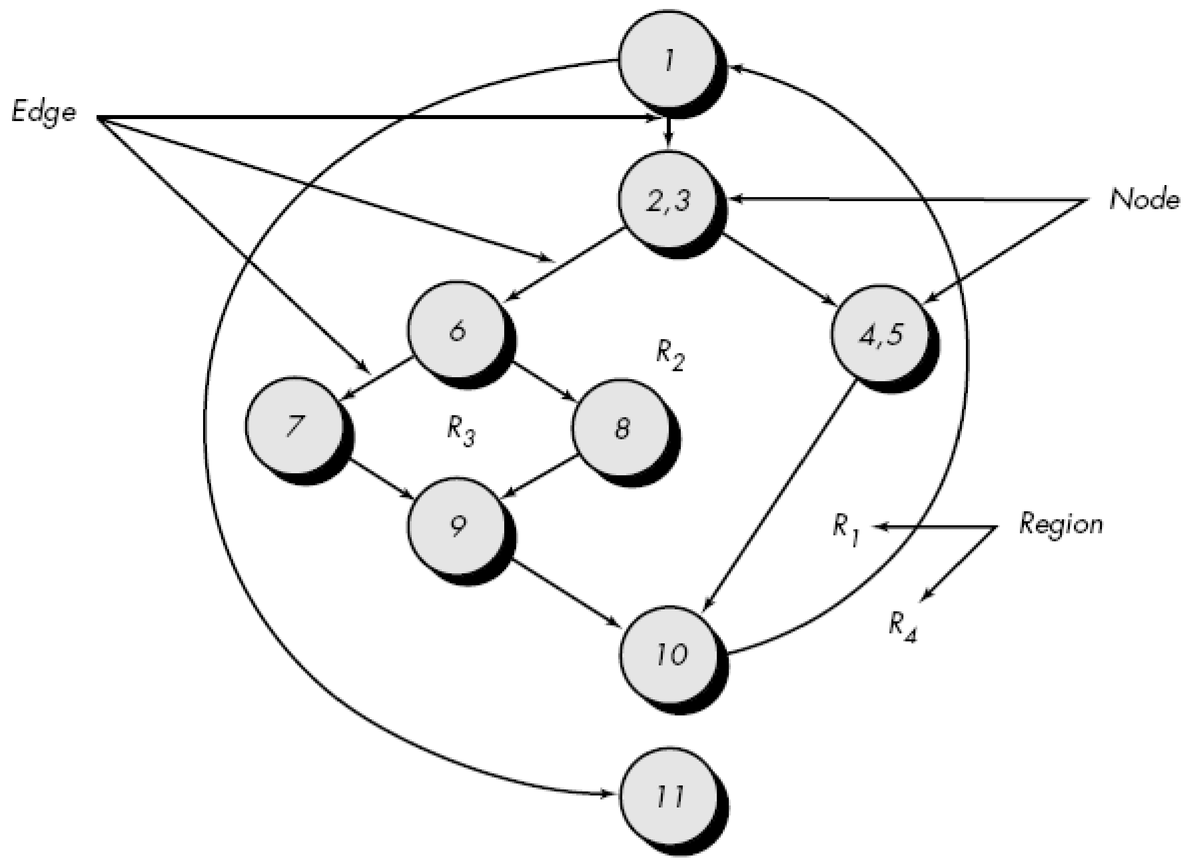


Fig (d) Flow Graph

Referring to Figure above, 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. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region.

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure (e) below, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it.

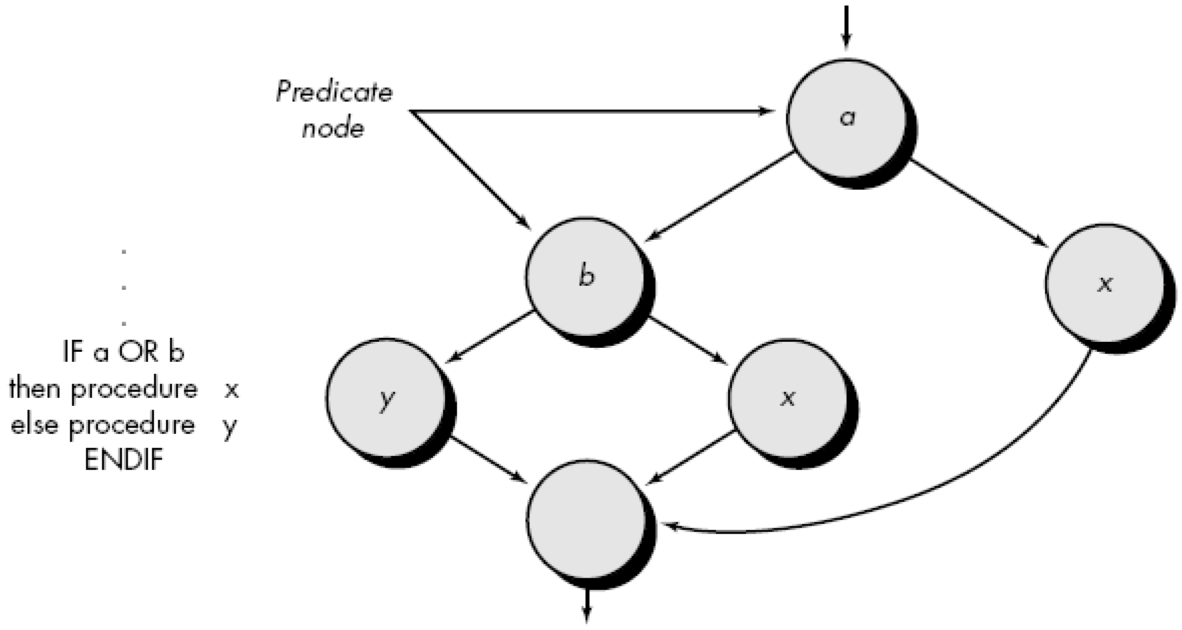


Fig (e) Compound logic

**Cyclomatic Complexity**

***Cyclomatic complexity*** is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An ***independent path*** is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure (d) 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. Paths 1, 2, 3, and 4 constitute a ***basis set*** for the flow graph in Figure (d). That is, if tests can be designed 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. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. 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, $N$ 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 (d), 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 (d) is 4.

More important, the value for V (G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

## Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure ***average,*** depicted in PDL in Figure (f), will be used as an example to illustrate each step in the test case design method. Note that ***average,*** although an extremely simple algorithm contains compound conditions and loops. 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.** A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure (f), a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure (g).

**2. Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, V (G), is determined by applying the algorithms. It should be noted that V (G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure (g),

V (G) = 6 regions

V (G) = 17 edges – 13 nodes + 2 = 6

V (G) = 5 predicate nodes + 1 = 6

**3. Determine a basis set of linearly independent paths.** The value of V (G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

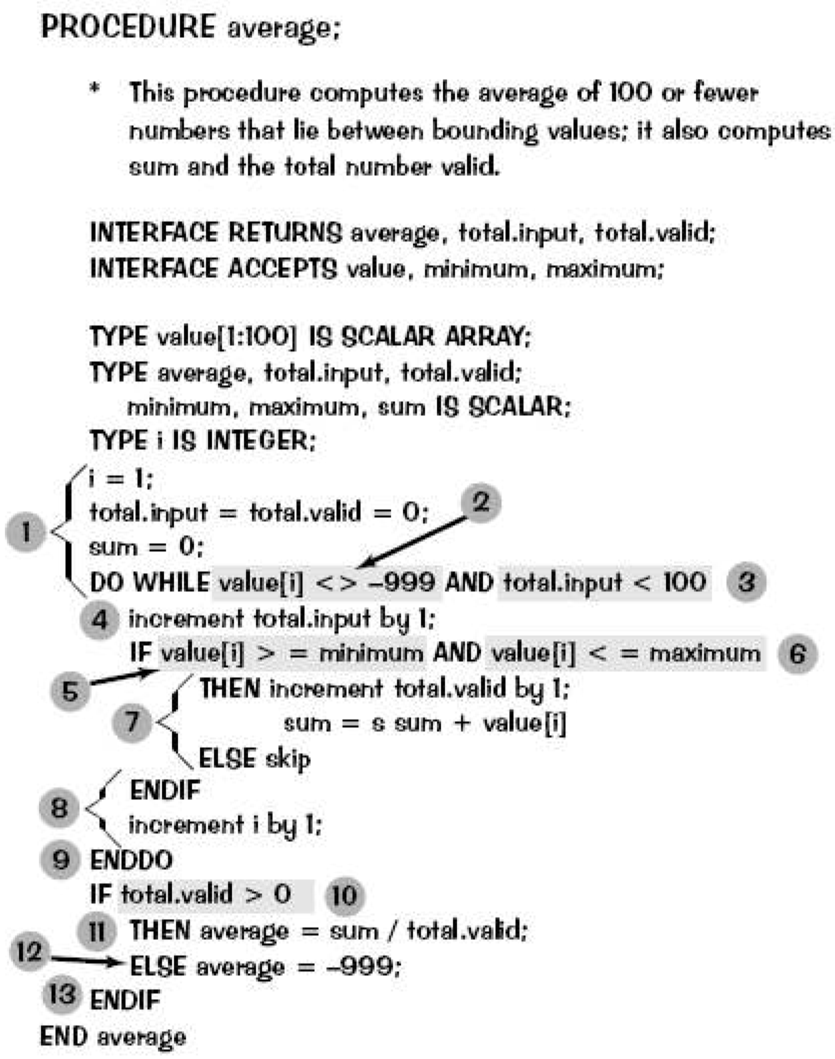
Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.


1
2
3
4
5
6
7
8
9
10
11
12
13

```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes the
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
        increment total.input by 1;
        IF value[i] > = minimum AND value[i] < = maximum
            THEN increment total.valid by 1;
                    sum = s sum + value[i]
            ELSE skip
        ENDIF
        increment i by 1;
    ENDDO
    IF total.valid > 0
        THEN average = sum / total.valid;
        ELSE average = -999;
    ENDIF
END average
```

Fig (f) PDL for the procedure average

**4. Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set are

**Path 1 test case:**

value ($k$) = valid input, where $k < i$ for $2 \leq i \leq 100$

value ($i$) = – 999 where $2 \leq i \leq 100$

Expected results: Correct average based on k values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

**Path 2 test case:**

value(1) = – 999

Expected results: Average = – 999; other totals at initial values.

**Path 3 test case:**

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

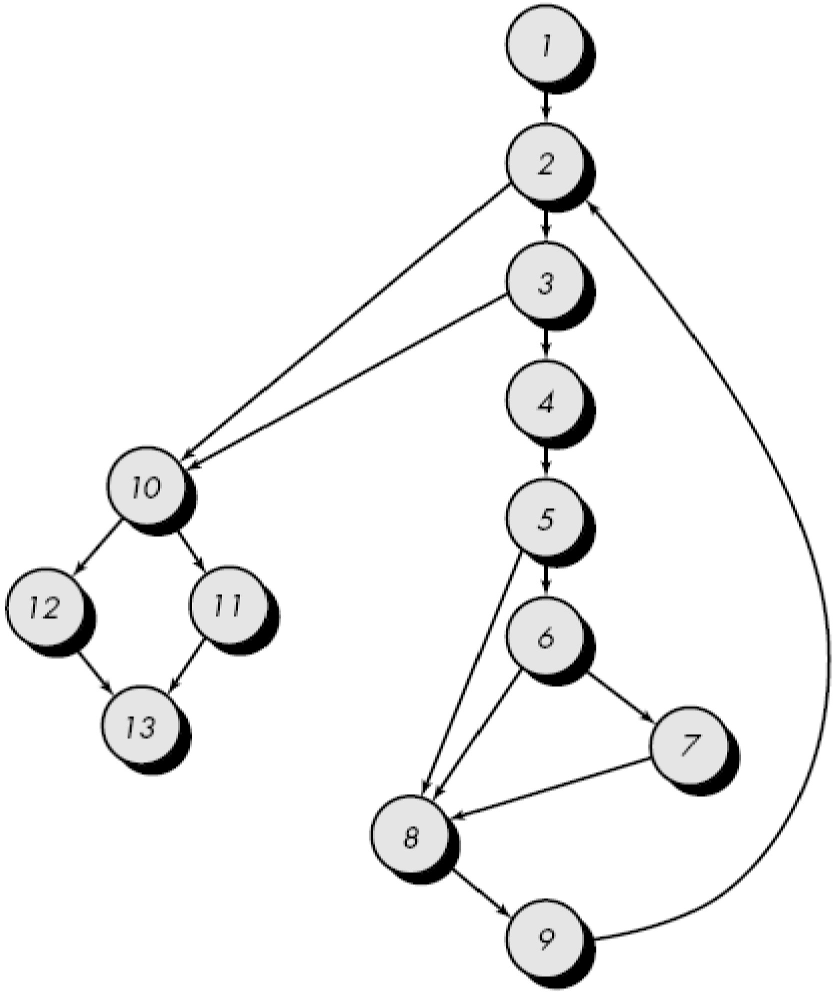


Fig (g) Flow graph for the procedure average

**Path 4 test case:**

value (*i*) = valid input where i < 100

value (*k*) < minimum where *k* < *i*

Expected results: Correct average based on k values and proper totals.

**Path 5 test case:**

value (*i*) = valid input where *i* < 100

value (*k*) > maximum where *k* <= *i*

Expected results: Correct average based on n values and proper totals.

**Path 6 test case:**

value (*i*) = valid input where *i* < 100

Expected results: Correct average based on n values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

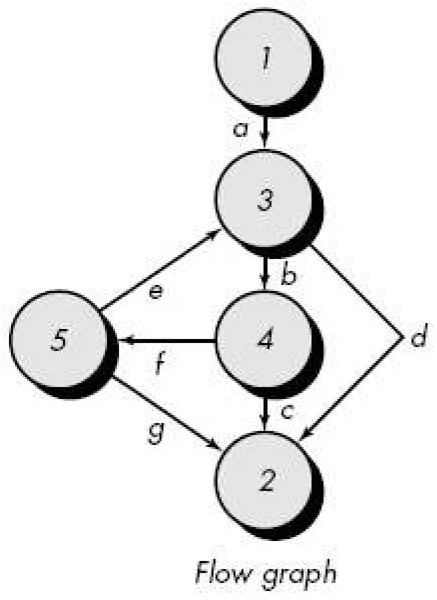
## Graph Matrices

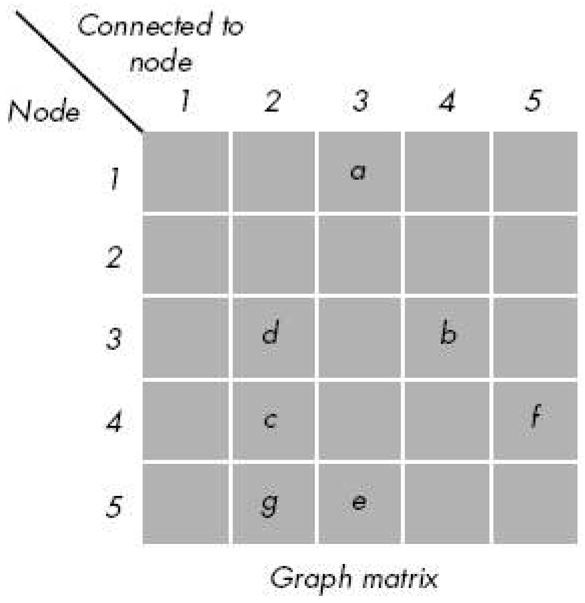
The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a ***graph matrix,*** can be quite useful. A ***graph matrix*** is a square matrix whose size (i.e., 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. A simple example of a flow graph and its corresponding graph matrix is shown in Figure (h).

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.*

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a ***link weight*** to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. 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). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.





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

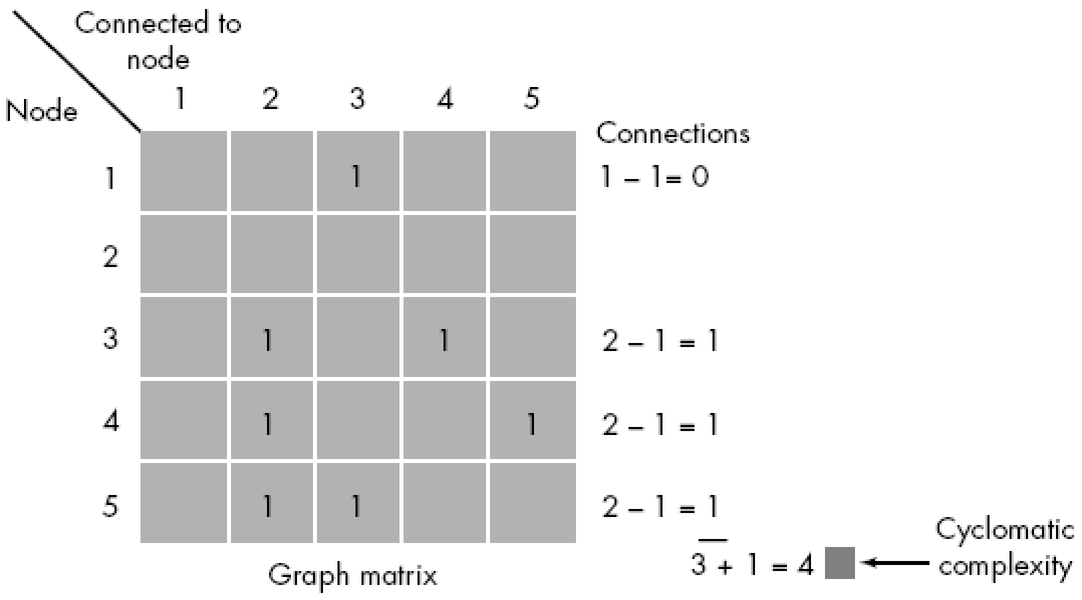
Graph matrix

Fig (h) Graph Matrix

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure (h) is redrawn as shown in Figure (i). Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a ***connection matrix.***

Referring to Figure (i), each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity.

Beizer provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.



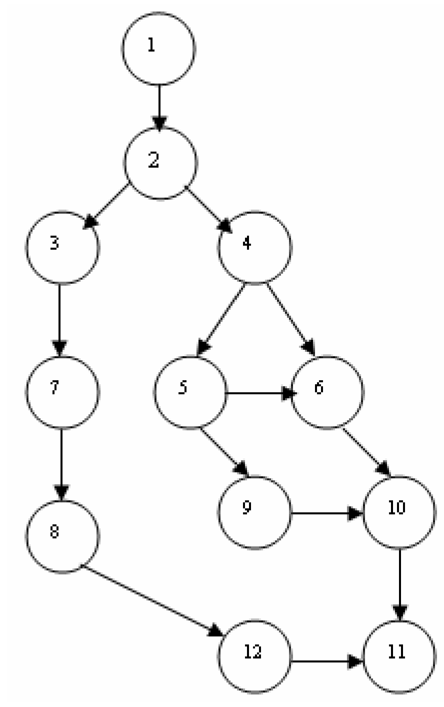
| Node \ Connected to node | 1 | 2 | 3 | 4 | 5 | Connections |
|---|---|---|---|---|---|---|
| 1 | | | 1 | | | 1 – 1= 0 |
| 2 | | | | | | |
| 3 | | 1 | | 1 | | 2 – 1 = 1 |
| 4 | | 1 | | | 1 | 2 – 1 = 1 |
| 5 | | 1 | 1 | | | 2 – 1 = 1 |

Graph matrix

$\overline{3} + 1 = 4$ ▨ ← Cyclomatic complexity

Fig (i) Connection Matrix

Q.1) Compute the cyclomatic complexity of a flow graph shown below and also find all independent paths:



Sol:

(a) Cyclomatic Complexity V (G) = E – N + 2

Here, E = 14

N = 12

So, V (G) = 14 – 12 + 2

V (G) = 4

(b) Cyclomatic Complexity V (G) = P + 1

Here, P = 3

So, V (G) = 3 +1

V (G) = 4.

(c) Cyclomatic Complexity V (G) = No of regions

So, V (G) = 4

(d) Independent Paths:

Path 1: 1 – 2 – 3 – 7 – 8 – 12 – 11

Path 2: 1 – 2 – 4 – 5 – 9 – 10 – 11

Path 3: 1 – 2 – 4 – 5 – 6 – 10 – 11

Path 4: 1 – 2 – 4 – 5 – 10 – 11

Q. 2) Determine the cyclomatic complexity for the function search.

```
void search (int a[ ], int n, int x)
{
        int i = 0, flag = 0;
        while (i<n)
        {
                if (a[i] == x)
                {
                        flag = 1;
                        break;
                }
                i = i +1;
        }
        if (flag == 1)
                printf("found");
        else
                printf("Not");
}
```

Sol:

The flow chart & flow graph of function search is shown below:

Cyclomatic Complexity V (G) = No. of Regions

V (G) = 4

Cyclomatic Complexity V (G) = E – N + 2

Here, E = 12

N = 10

So, V (G) = 12 – 10 + 2

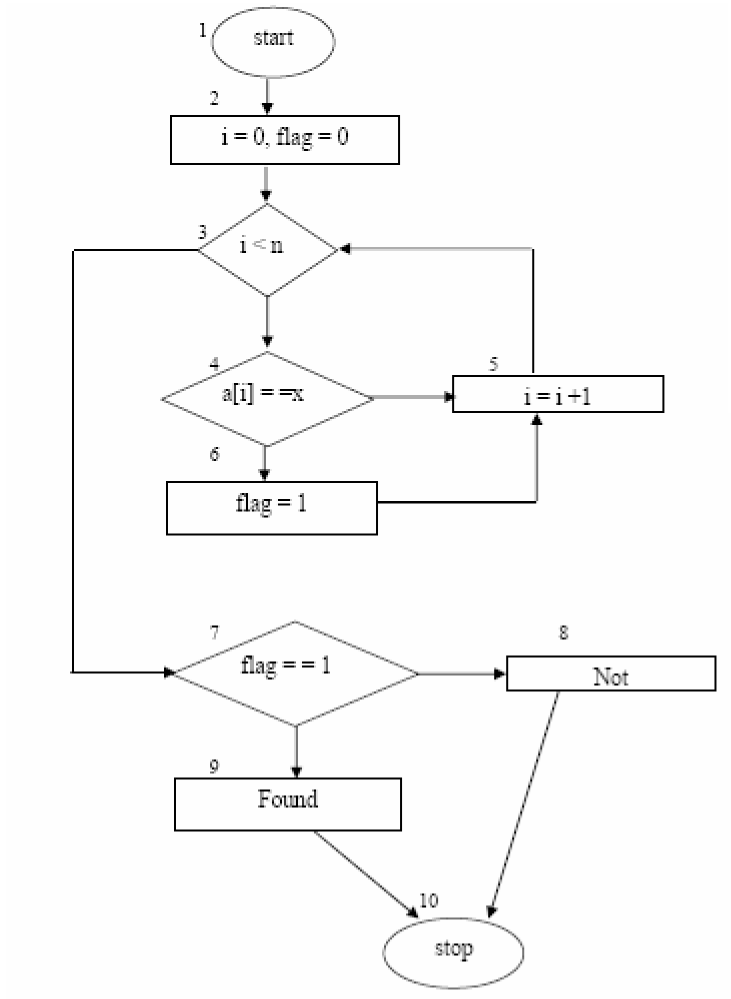
V (G) = 4

Cyclomatic Complexity V (G) = P + 1

Here, P = 3 (Nodes 3, 4 and 7 are predicate nodes)

So, V (G) = 3 + 1

V (G) = 4



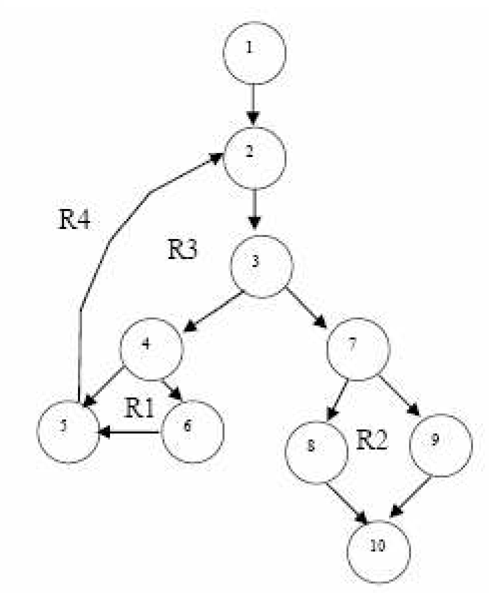


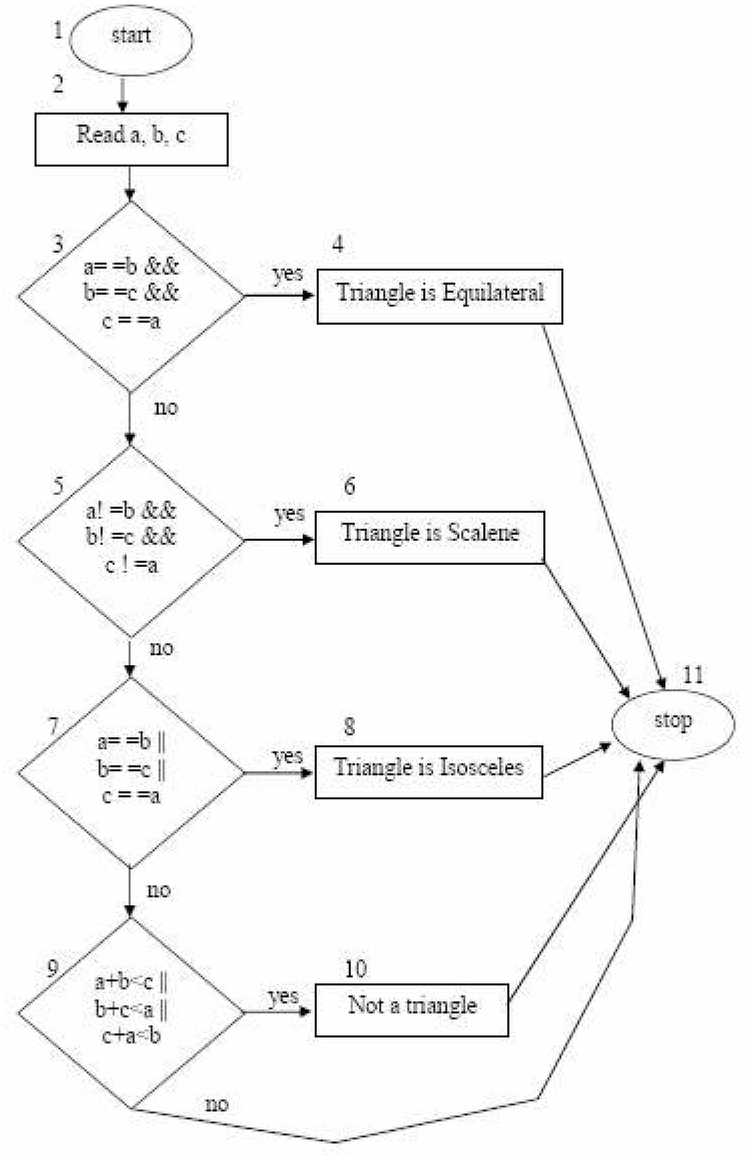
Fig Flow Chart Fig Flow Graph

Q. 3) Write a function to check whether a given triangle is scalene, isosceles, equilateral, not a triangle for which lengths of three sides are given to you. Draw its flow graph. Find its cyclomatic complexity.

Sol: The Function for finding the triangle is scalene, isosceles, equilateral, not a triangle is as follows:

```
void check (int a, int b, int c)
{
        if (a==b && b==c && c==a)
                printf("Triangle is equilateral");
        else if (a!=b && b!=c && c!=a)
                printf("Triangle is scalene");
        else if (a==b || b==c || c==a)
                printf("Triangle is isosceles");
        else if ((a+b)<c || (b+c)<a || (a+c)<b)
                printf("Not a triangle");
}
```

The flow chart and flow graph of this function is shown below:



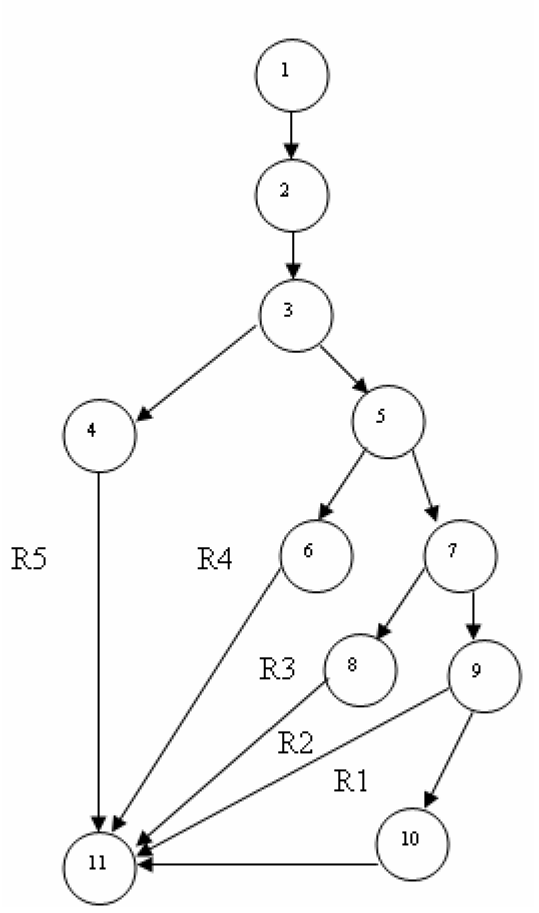


Fig Flow Chart Fig Flow Graph

Cyclomatic Complexity V (G) = No. of regions

V (G) = 5

Cyclomatic Complexity V (G) = P + 1

Here, P = 4 (Nodes 3, 5, 7 and 9 are predicate nodes)

So, V (G) = 4 + 1

V (G) = 5

Cyclomatic Complexity V (G) = E – N + 2

Here E = 14, N = 11

So, V (G) = 14 – 11 + 2

V (G) = 5

The independent paths are:

Path 1: 1 – 2 – 3 – 4 – 11

Path 2: 1 – 2 – 3 – 5 – 6 – 11

Path 3: 1 – 2 – 3 – 5 – 7 – 8 – 11

Path 4: 1 – 2 – 3 – 5 – 7 – 9 – 11

Path 5: 1 – 2 – 3 – 5 – 7 – 9 – 10 – 11

**Test cases are:**

**Path 1:** test case:

a, b, c = valid input

Expected results: If a = = b && b = =c && c = = a then message “Triangle is equilateral” is printed.

**Path 2:** test case:

a, b, c = valid input

Expected results: If a != b && b !=c && c != a then message “Triangle is Scalene” is printed.

**Path 3:** test case:

a, b, c = valid input

Expected results: If a = = b || b = =c || c = = a then message “Triangle is isosceles” is printed.

**Path 4:** test case:

a, b, c = valid input

Expected results: If (a+b)<c || (b+c)<a || (c+a)<b then message “Not a triangle” is printed.

**Path 5:** test case:

a, b, c = valid input

Expected results: If all the conditions are failed then execution of function is stopped.

Q. 4) Write a function to find the different roots of a quadratic equation:

(i) Give flow graph.

(ii) Find Cyclomatic Complexity.

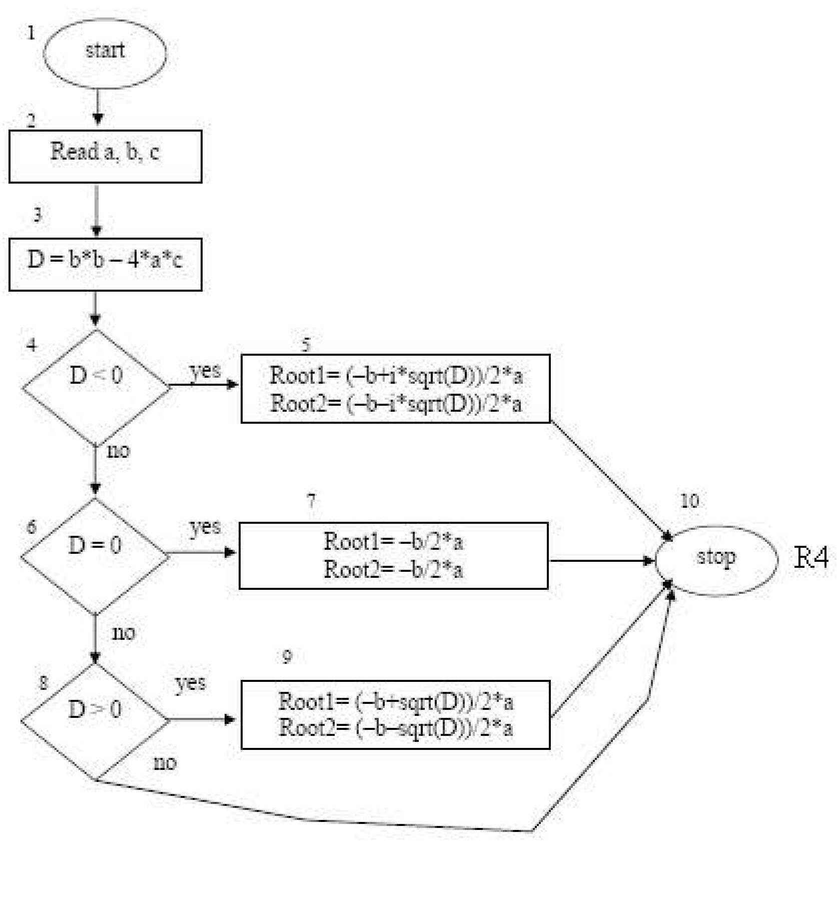
(iii) Find basic test paths.

(iv) Derive test cases.

Sol: The Function for finding the different roots for a quadratic equation is as follows:

```
void calc (int a, int b, int c)
{
        D = b*b – 4*a*c;
        if (D<0)
                Root1 = (–b+i*sqrt(D))/2*a;
                Root2 = (–b–i*sqrt(D))/2*a;
        else if (D ==0)
                Root1 = –b/2*a;
                Root2 = –b/2*a;
        else if (D>0)
                Root1 = (–b+sqrt(D))/2*a;
                Root2 = (–b+sqrt(D))/2*a;
}
```

The flow chart and flow graph of this function is shown below:



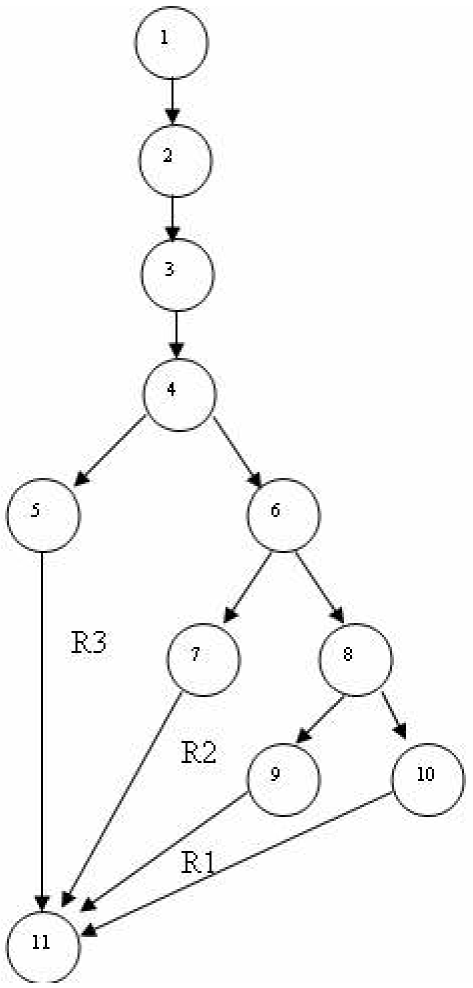


Fig Flow Chart

Fig Flow Graph

Cyclomatic Complexity V (G) = No. of regions

V (G) = 4

Cyclomatic Complexity V (G) = P + 1

Here, P = 3 (Nodes 4, 6, and 8 are predicate nodes)

So, V (G) = 3 + 1

V (G) = 4

Cyclomatic Complexity V (G) = E – N + 2

Here E = 13

N = 11

So, V (G) = 13 – 11 + 2

V (G) = 4

The independent paths are:

Path 1: 1 – 2 – 3 – 4 – 5 – 11

Path 2: 1 – 2 – 3 – 4 – 6 – 7 – 11

Path 3: 1 – 2 – 3 – 4 – 6 – 8 – 9 – 11

Path 4: 1 – 2 – 3 – 4 – 6 – 8 – 10 – 11

**Test cases are:**

Path 1: test case:

a, b, c = valid input

Expected results: If D < 0 then Root1= (–b+i*sqrt(D))/2*a and Root2= (–b–i*sqrt(D))/2*a are printed.

Path 2: test case:

a, b, c = valid input

Expected results: If D = 0 then Root1= –b/2*a and Root2= –b/2*a are printed.

Path 3: test case:

a, b, c = valid input

Expected results: If D > 0 then Root1= (–b+sqrt(D))/2*a and Root2= (–b–sqrt(D))/2*a are printed.

Path 4: test case:

a, b, c = valid input

Expected results: If all the conditions are failed then execution of function is stopped.

## Condition Testing

1. Condition testing is a test case design method that exercises the logical conditions contained in a program module.
2. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

   E1 <relational-operator> E2

   Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.
3. A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR ( | ), AND (&) and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.
4. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.
5. If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:
   - Boolean operator error (incorrect/missing/extra Boolean operators).
   - Boolean variable error.
   - Boolean parenthesis error.
   - Relational operator error.
   - Arithmetic expression error.
6. The condition testing method focuses on testing each condition in the program.
7. Condition testing strategies generally have two advantages.
   a) Measurement of test coverage of a condition is simple.
   b) The test coverage of conditions in a program provides guidance for the generation of additional tests for the program.
8. The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program *P* is effective for detecting errors in the conditions contained in *P,* it is likely that this test set is also effective for detecting other errors in *P.* In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

## Data Flow Testing

1. The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

2. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

   DEF(S) = {X | statement S contains a definition of X}

   USE(S) = {X | statement S contains a use of X}

   If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.
3. A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.
4. One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy.
5. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the then part has no definition of any variable and the else part does not exist. In this situation, the else branch of the if statement is not necessarily covered by DU testing.
6. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.
7. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
    B1;
    do while C1
        if C2
            then
                if C4
                    then B4;
                    else B5;
                endif;
            else
                if C3
                    then B2;
                    else B3;
                endif;
            endif;
        enddo;
        B6;
    end proc;
```

8. To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL. Assume that variable X is defined in the last statement of blocks B1, B2, B3, B4, and B5 and is used in the first statement of blocks B2, B3, B4, B5, and B6. The DU testing strategy requires an execution of the shortest path from each of Bi, $0 < i \leq 5$, to each of Bj, $1 < j \leq 6$. (Such testing also covers any use of variable X in conditions C1, C2, C3, and C4.) Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from Bi, $0 < i \leq 5$, to B6 and other DU chains can be covered by making these five paths contain iterations of the loop.

9. If we apply the branch testing strategy to select test paths of the PDL just noted, we do not need any additional information. To select paths of the diagram for BRO testing, we need to know the structure of each condition or block. (After the selection of a path of a program, we need to determine whether the path is feasible for the program; that is, whether at least one input exists that exercises the path.) Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection. However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

## Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

**Simple loops –** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n – 1$, $n$, $n + 1$ passes through the loop.

**Nested loops –** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

**Concatenated loops –** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

**Unstructured loops –** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

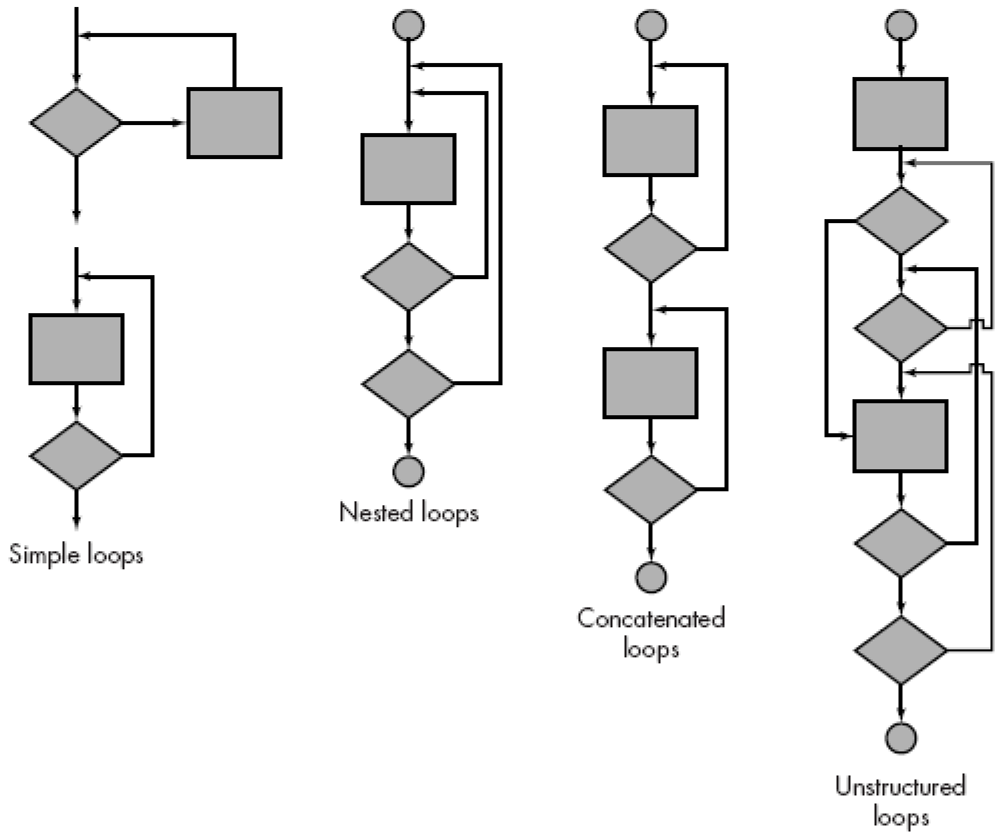


Fig Classes of Loops

**Equivalence Partitioning**

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either, a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

**1.** If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.

**2.** If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.

**3.** If an input condition specifies a member of a *set,* one valid and one invalid equivalence classes are defined.

**4.** If an Input condition is *Boolean,* one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

area code—blank or three-digit number

prefix—three-digit number not beginning with 0 or 1

suffix—four-digit number

password—six digit alphanumeric string

commands—check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as

area code: Input condition, *Boolean*—the area code may or may not be present.
Input condition, *range*—values defined between 200 and 999, with specific exceptions.

prefix: Input condition, *range*—specified value >200
Input condition, value—four-digit length

password: Input condition, *Boolean*—a password may or may not be present.
Input condition, *value*—six-character string.

command: Input condition, *set*—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest numbers of attributes of an equivalence class are exercised at once.

## Boundary Value Analysis

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that ***boundary value analysis*** (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

**1.** If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

**2.** If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

**3.** Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

**4.** If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

## A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure below. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

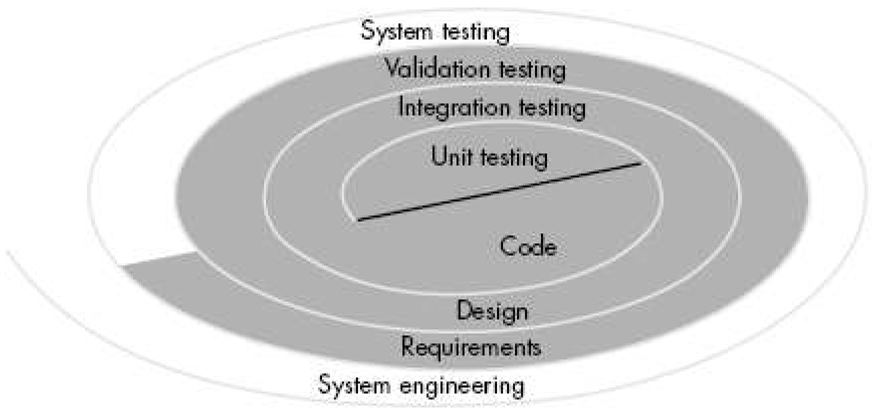


Fig A testing Strategy

A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to ***integration testing,*** where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter ***validation testing,*** where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at ***system testing,*** where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

**Various Levels of Testing**

If we focus the process of testing in context of software engineering, it is actually a series of four steps that are implemented sequentially. Each step known as level is meant for test different aspects of the system. The basic levels of testing are

- Unit Testing
- Validation Testing
- Integration Testing
- System Testing

The relation of the fault introduced indifferent phase and different levels of testing are shown below:

$$\begin{tikzcd}
\text{System engineering} \arrow[r, leftrightarrow] \arrow[d] & \text{System Testing} \arrow[d] \\
\text{Requirement} \arrow[r, leftrightarrow] \arrow[d] & \text{Validation Testing} \arrow[d] \\
\text{Design} \arrow[r, leftrightarrow] \arrow[d] & \text{Integration Testing} \arrow[d] \\
\text{Code} \arrow[r, leftrightarrow] & \text{Unit Testing}
\end{tikzcd}$$

1. **Unit Testing** – This testing is essentially for verification of the code produced during the code phase. The basic objective of this phase is to test the internal logic of the module. Unit testing makes heavy use of the white box testing technique, exercising specific path in a module control structure to ensure complete coverage and maximum error detection. In Unit testing the module interface is tested to ensure that information properly flows into and out of the program unit. Also the data structure is tested to ensure that data stored temporarily maintain its integrity during execution or not.
2. **Integration Testing** – This testing uses the address of verification and program construction. Black box testing technique is widely used in this testing strategy although limited amount of white box testing may be used to ensure coverage of control paths. The basic emphasis of this testing the interfaces between the modules.

3. **Validation Testing** – Criterion, which is established during requirement analysis, is tested and it provides final assurance that software meets all functional behavior, performance requirement etc.
4. **System Testing** – Here the entire software is tested. The last high order testing stage falls outside the boundary of software engineering. System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

## Guidelines for a Successful Testing Strategy

1. Specify product requirements in a quantifiable manner long before testing commences.
2. State testing objectives explicitly.
3. Understand the users of the software and develop a profile for each user category.
4. Develop a testing plan that emphasizes "rapid cycle testing."
5. Build "robust" software that is designed to test itself.
6. Use effective formal technical reviews as a filter prior to testing.
7. Conduct formal technical reviews to assess the test strategy and test cases themselves.
8. Develop a continuous improvement approach for the testing process.

## Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

### Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure below. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

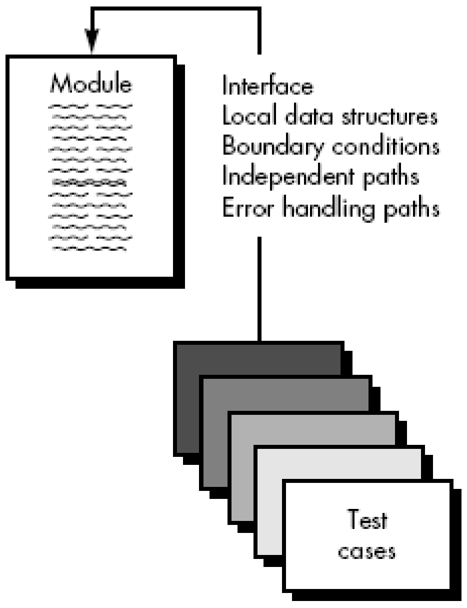


Fig Unit Test

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are

(1) Misunderstood or incorrect arithmetic precedence,

(2) Mixed mode operations,

(3) Incorrect initialization,

(4) Precision inaccuracy,

(5) Incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as

(1) Comparison of different data types,

(2) Incorrect logical operators or precedence,

(3) Expectation of equality when precision error makes equality unlikely,

(4) Incorrect comparison of variables,

(5) Improper or nonexistent loop termination,

(6) Failure to exit when divergent iteration is encountered, and

(7) Improperly modified loop variables.

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

**Unit Test Procedures**

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

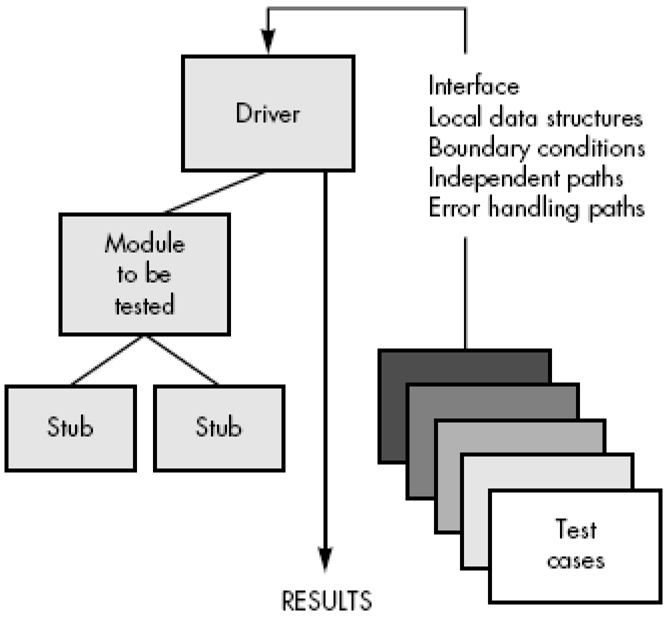


Fig Unit Test Environment

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure above. In most applications a ***driver*** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. ***Stubs*** serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing. Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

**Advantage of Unit Testing**

1. Can be applied directly to object code and does not require processing source code.
2. Performance profilers commonly implement this measure.

**Disadvantages of Unit Testing**

1. Insensitive to some control structures (number of iterations)
2. Does not report whether loops reach their termination condition
3. Statement coverage is completely insensitive to the logical operators (|| and &&).

**Integration Testing**

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered.

Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

**Top-down Integration**

***Top-down integration testing*** is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

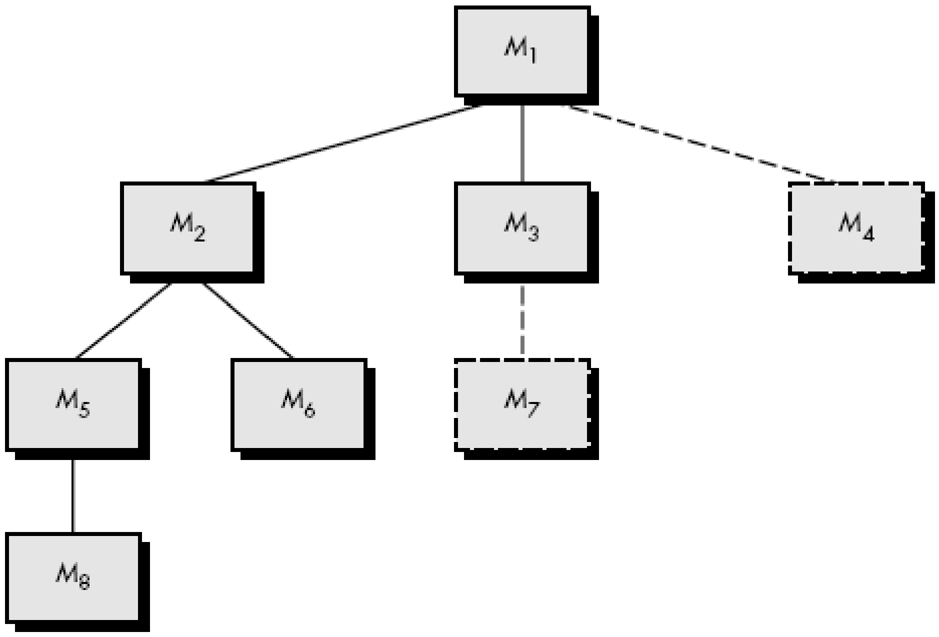


Fig Top Down Integration

Referring to Figure above, ***depth-first integration*** would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. ***Breadth-first integration*** incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

**1.** The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

**2.** Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

**3.** Tests are conducted as each component is integrated.

**4.** On completion of each set of tests, another stub is replaced with the real component.

**5.** Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path. The incoming path may be integrated in a top-down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of functional capability is a confidence builder for both the developer and the customer.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices:

(1) Delay many tests until stubs are replaced with actual modules,

(2) Develop stubs that perform limited functions that simulate the actual module, or

(3) Integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to loose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex.

## Bottom-up Integration

***Bottom-up integration testing***, as its name implies, begins construction and testing with ***atomic modules*** (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

**1.** Low-level components are combined into clusters (sometimes called ***builds***) that perform a specific software sub-function.

**2.** A driver (a control program for testing) is written to coordinate test case input and output.

**3.** The cluster is tested.

**4.** Drivers are removed and clusters are combined moving upward in the program structure.

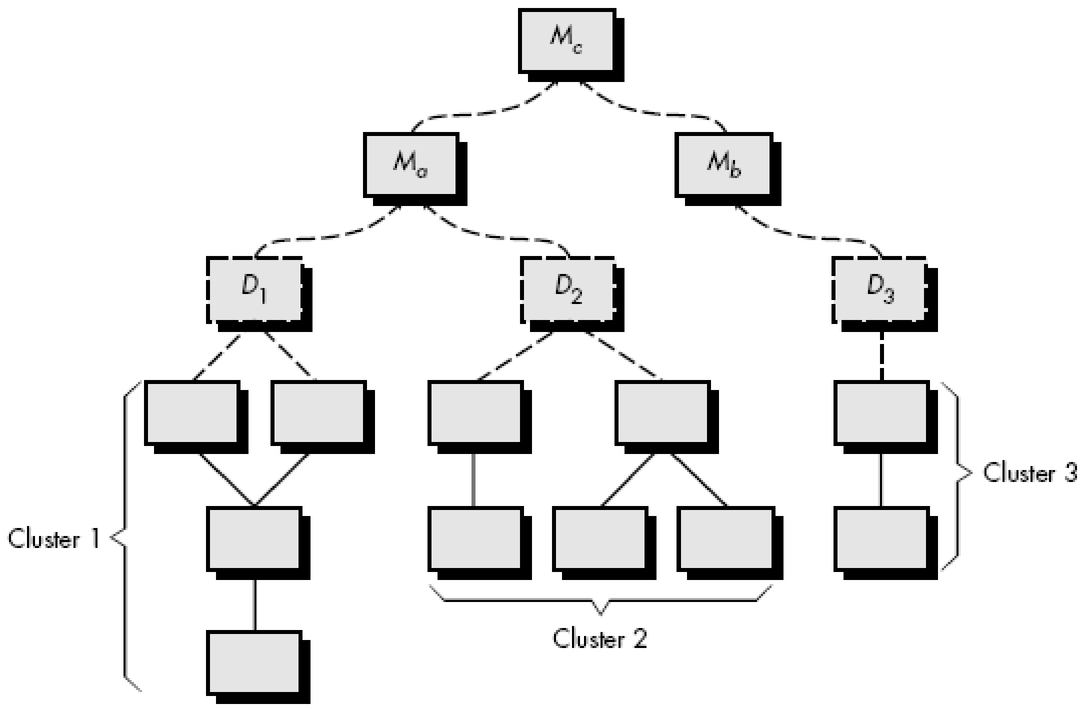


Fig Bottom up Integration

Integration follows the pattern illustrated in Figure above. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to $M_a$. Drivers $D_1$ and $D_2$ are removed and the clusters are interfaced directly to $M_a$. Similarly, driver $D_3$ for cluster 3 is removed prior to integration with module $M_b$. Both $M_a$ and $M_b$ will ultimately be integrated with component $M_c$, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

## Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, ***regression testing*** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

## Smoke Testing

***Smoke testing*** is an integration testing approach that is commonly used when "shrink – wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

**1.** Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

**2.** A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

**3.** The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell describes the smoke test in the following manner:

"The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly".

Smoke testing provides a number of benefits when it is applied on complex, time critical software engineering projects:

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- *The quality of the end-product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.
- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

## Validation Testing

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests—***validation testing***—may begin. Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the ***Software Requirements Specification***— a document that describes all user-visible attributes of the software. The specification contains a section called ***Validation Criteria***. Information contained in that section forms the basis for a validation testing approach.

**Validation Test Criteria**

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exists:

(1) The function or performance characteristics conform to specification and are accepted or

(2) A deviation from specification is uncovered and a ***deficiency list*** is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

**Configuration Review**

An important element of the validation process is a ***configuration review***. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle.

**Alpha Testing**

The ***alpha test*** is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

**Beta Testing**

The ***beta test*** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

**System Testing**

Software is the only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and

(1) Design error-handling paths that test all information coming from other elements of the system,

(2) Conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) Record the results of tests to use as "evidence" if finger-pointing does occur, and

(4) Participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

## Recovery Testing

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

***Recovery testing*** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

## Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

***Security testing*** attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack." During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

## Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

***Stress testing*** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

(1) Special tests may be designed that generate ten interrupts per second, when one or two is the average rate,

(2) Input data rates may be increased by an order of magnitude to determine how input functions will respond,

(3) Test cases that require maximum memory or other resources are executed,

(4) Test cases that may cause thrashing in a virtual operating system are designed,

(5) Test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called ***sensitivity testing***. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

## Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. ***Performance testing*** is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

**Debugging**

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

***Debugging*** occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

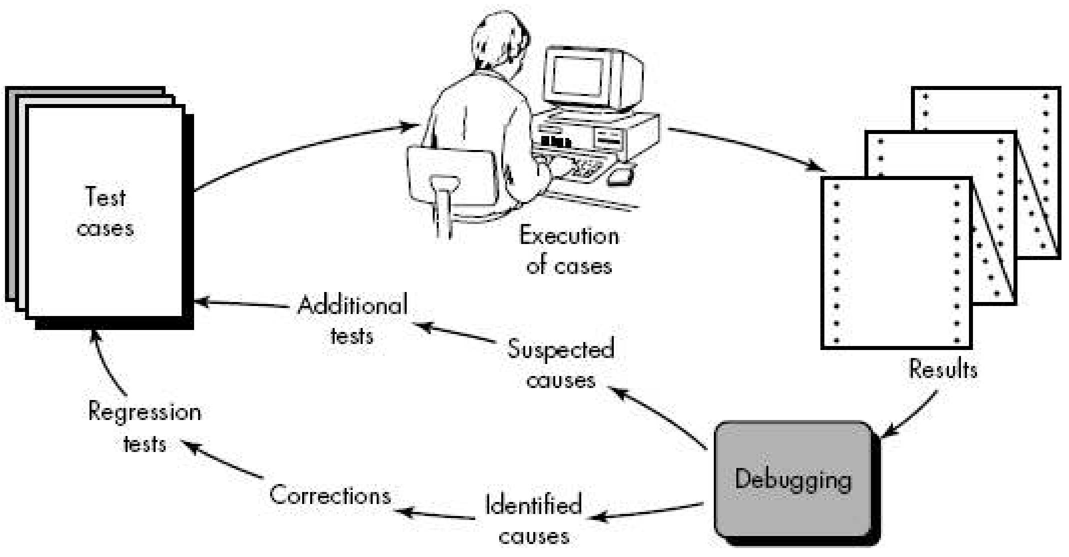


Fig Debugging Process

**The Debugging Process**

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure above, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes:

(1) The cause will be found and corrected, or

(2) The cause will not be found.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g. the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

## Debugging Approaches

In general, three categories for debugging approaches may be proposed:

(1) Brute force,

(2) Backtracking, and

(3) Cause elimination.

1. The ***brute force*** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!
2. ***Backtracking*** is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
3. The third approach to debugging – ***Cause elimination***—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

## Software Quality Assurance (SQA)

### Quality

It's a measurable quantity which can be compared with known standards for any product or process. It specifies the degree to which a product or process maintains the known standards. Quality can be classified in to two groups – a) Quality of design and b) Quality of conformance.

a) **Quality of design:** - It refers to the standards maintained during design processes which include the grade of - i) Materials, ii) Tolerances and iii) Performance specifications.

b) **Quality of conformance**: - It refers to the degree to which the design specifications are followed during manufacturing.

### Quality Control

a) It's a series of inspections, reviews and tests used throughout the development cycle to ensure that each work product meets the requirements placed up on it.

b) It includes a feedback loop to the process that created the work product. The feedback loop helps tune up the process during the creation of the product.

c) It should be a part of the manufacturing process.

**Quality Assurance**

It's the auditing and reporting functions of the management. If the data provided through quality assurance identify problems then it's the management's responsibility to solve the problems and apply the necessary resources to resolve the quality issues. The goal of quality assurance is to provide management to provide information and data related to product quality so that they can evaluate whether or not the manufacturing process is maintaining the product quality.

**Costs related to maintenance of Quality**

The Quality related costs are divided in to following categories: -

a) Prevention (from degraded quality) costs,

b) Appraisal (inspection) costs and

c) Failure (to maintain quality) costs.

a) **Prevention costs** :- It includes costs required to implement the following –
   - i) Quality planning,
   - ii) Formal technical reviews,
   - iii) Test equipment and
   - iv) Training.

b) **Appraisal costs**: - It includes activities to gain the quality standard of a product when going through the process for the first time. Examples include –
   - i) In-process and inter process inspections,
   - ii) Testing,
   - iii) Equipment calibration and maintenance.

c) **Failure costs**: - These costs are raised when a failure occurs other wise these costs get avoided. Failure costs can be further divided in to following –
   - I. **Internal failure costs**: - These are incurred when the defects are detected before the product gets delivered to the customer. The activities included here are –
     - 1) Rework,
     - 2) Repairs and
     - 3) Failure mode analysis.
   - II. **External failures**: - These are the costs incurred when the defects are detected after the product has reached the user. Examples are –
     - 1) Complaint resolution,
     - 2) Product return and replacement,
     - 3) Warranty support, etc.

**Software Quality Assurance (SQA) Activities**

The activities performed in software quality assurance involves two groups –

a) The software engineers doing the technical work: - Their work includes –
   - 1) Applying technical methods and measures for ensuring high quality,
   - 2) Conducting formal reviews and
   - 3) Performing well planned software testing.

b) The SQA group :- That has the responsibility for –
   - 1) Quality assurance planning,
   - 2) Oversight,

3) To assist the software engineering team in achieving high standards of production,

4) Record keeping,

5) Analysis and

6) Reporting.

The SQA activities performed by both SQA groups and Software engineering group are governed by an SQA plan for the project. An SQA plan consists of the following activities:-

1) **The Management section**: - It consists of the following standards –
   a) The organizational structure for the project,
   b) Tasks and activities to be maintained in the organizational structure,
   c) The organizational roles and responsibilities for maintaining good product quality.
2) **The Documentation section**: - It describes the standards to be maintained in the following elements –
   a) Project documents (e.g. project plan),
   b) Models (e.g. class hierarchies),
   c) Technical documents (e.g. test plans, specifications),
   d) User documents (e.g. help files).
3) **Standards, Practices and Conventions section**: - It contains the list of applicable standards and practices to be followed during the software process. It requires that all project, process and product matrices that are used in the project should be listed out.
4) **Reviews and Audits section**: - It describes the quality reviews and quality audits that are to be conducted by the software engineering group, the SQA group and the customer.
5) **The Test section**:-It refers to the software test plan and testing procedures. It also refers to the standards to be maintained for keeping the records of the tests performed.
6) **Problem reporting and corrective action section**: - It defines the procedures for reporting, tracking and resolving errors & defects. It also specifies the organizational responsibilities for these activities.

**Statistical Quality Assurance**

It implies the following steps –

1) Information about software defects is collected and categorized.
2) An attempt is made to trace the cause of each defect to its basis.
3) Using the Pareto principle to find out the vital view which states that 80 percent of the defects can be traced to 20 percent of all possible causes. Which means that if all the errors are found and all the possible causes for those errors are also found, then its observed that about 20 percent of the possible causes would justify about 80 percent of the total errors. This 20 percent is collectively called the "vital few".
4) Once the vital view causes are identified then the procedure of error correction should be carried out.

## Software Maintenance

Software Maintenance is a very broad activity that includes error, corrections, enhancement of capabilities, deletion of obsolete capabilities and optimization. Because change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications. So any work done to change the software after it is in operation is considered to be maintenance. The purpose is to preserve the value of software overtime. The value can be enhanced by expanding the customer base, meeting additional requirements, becoming easier to use, more efficient and employing newer technology. Maintenance may span for 500 years, whereas development may be 1–2 years.

**Types of Maintenance**

There are four major categories of maintenance

1. **Corrective Maintenance** – This refers to modifications initiated by defects in the software. A defect can result from, **design errors, logic errors** and **coding errors**. **Design errors** occur when, changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. **Logic errors** result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test data. **Coding errors** are caused by incorrect implementations of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors.

   In the event of system failure due to an error, actions are taken to restore operation of the software system. Due to pressure from management, maintenance personnel sometimes resort to emergency fixes known as patching. The adhoc nature of this approach often gives rise to a range of problems that include increased program complexity and unforeseen ripple effects. Unforseen ripple effects imply that a change to one part of a program may affect other sections in an unpredictable manner, thereby leading to distortion in the logic of the system. This is often due to lack of time to carry out a through "impact analysis" before effecting the change.
2. **Adaptive Maintenance** – It includes modifying the software to match changes in the over changing environment. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the software. For example, business rules, government policies, work patterns, software and hardware operating platforms. A change to the whole or part of this environment will require corresponding modifications of the software.
   Thus, this type of maintenance includes any work initiated as a consequence of moving the software to a different hardware or software platform – compiler, operating system or new processor. Any change in the government policy can have far reaching ramification on the software. When European countries had decided to for "single European currency", this change affected all banking system software and was modified accordingly.
3. **Perfective Maintenance** – It means improving processing efficiency or performance, or restructuring the software to improve changeability. When the software becomes useful, the user tends to experiment with new cases beyond the scope for which it was initially developed. Explosion in requirements can take form of enhancement of existing system functionality or improvement in computational efficiency; for example, providing a Management Information System with a data entry module or a new message handling facility.
4. **Preventive Maintenance** – There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflects deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance. This term is often used with hardware systems and implies such things as lubrication of parts before need occurs, or automatic replacement of banks of light bulbs before they start to individually burn out.