UNIT 4 SOFTWARE TESTING

Structure **Page Nos.**

4.0	Introduction	53
4.1	Objectives	54
4.2	Basic Terms used in Testing	54
	4.2.1 Input Domain	
	4.2.2 Black Box and White Box testing Strategies	
	4.2.3 Cyclomatic Complexity	
4.3	Testing Activities	64
4.4	Debugging	65
4.5	Testing Tools	67
4.6	Summary	68
4.7	Solutions/Answers	69
4.8	Further Readings	69

4.0 INTRODUCTION

Testing means executing a program in order to understand its behaviour, that is, whether or not the program exhibits a failure, its response time or throughput for certain data sets, its mean time to failure, or the speed and accuracy with which users complete their designated tasks. In other words, it is a process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. Testing can also be described as part of the process of Validation and Verification.

Validation is the process of evaluating a system or component during or at the end of the development process to determine if it satisfies the requirements of the system, or, in other words, are we building the correct system?

Verification is the process of evaluating a system or component at the end of a phase to determine if it satisfies the conditions imposed at the start of that phase, or, in other words, are we building the system correctly?

Software testing gives an important set of methods that can be used to evaluate and assure that a program or system meets its non-functional requirements.

To be more specific, software testing means that executing a program or its components in order to assure:

- The correctness of software with respect to requirements or intent;
- The performance of software under various conditions;
- The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;
- The usability of software under various conditions;
- The reliability, availability, survivability or other dependability measures of software; or
- Installability and other facets of a software release.

The purpose of testing is to show that the program has errors. The aim of most testing methods is to systematically and actively locate faults in the program and repair them. Debugging is the next stage of testing. Debugging is the activity of:

- Determining the exact nature and location of the suspected error within the program and
- Fixing the error. Usually, debugging begins with some indication of the existence of an error.

The purpose of debugging is to locate errors and fix them.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- know the basic terms using in testing terminology;
- black box and White box testing techniques;
- other testing techniques; and
- some testing tools.

4.2 BASIC TERMS USED IN TESTING

Failure: A failure occurs when there is a deviation of the observed behavior of a program, or system, from its specification. A failure can also occur if the observed behaviour of a system, or program, deviates from its intended behavior.

Fault: A fault is an incorrect step, process, or data definition in a computer program. Faults are the source of failures. In normal language, software faults are usually referred to as "bugs".

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Test Cases: Ultimately, testing comes down to selecting and executing test cases. A test case for a specific component consists of three essential pieces of information:

- A set of test inputs;
- The expected results when the inputs are executed; and
- The execution conditions or environments in which the inputs are to be executed.

Some Testing Laws

- Testing can only be used to show the presence of errors, but never the absence or errors.
- A combination of different verification & validation (V&V) methods outprform any single method alone.
- Developers are unsuited to test their own code.
- Approximately 80% of the errors are found in 20% of the code.
- Partition testing, that is, methods that partition the input domain or the program and test according to those partitions. This is better than random testing.
- The adequacy of a test suite for coverage criterion can only be defined intuitively.

4.2.1 Input Domain

To conduct an analysis of the input, the sets of values making up the input domain are required. There are essentially two sources for the input domain. They are:

1. The software requirements specification in the case of black box testing method; and
2. The design and externally accessible program variables in the case of white box testing.

In the case of white box testing, input domain can be constructed from the following sources.

- Inputs passed in as parameters; Variables that are inputs to function under test can be: (i) Structured data such as linked lists, files or trees, as well as atomic data such as integers and floating point numbers;

(ii) A reference or a value parameter as in the *C* function declaration
int P(int *power, int base) {
...}

- Inputs entered by the user via the program interface;
- Inputs that are read in from files;
- Inputs that are constants and precomputed values; Constants declared in an enclosing scope of function under test, for example,

```
#define PI 3.14159
double circumference(double radius)
{
return 2*PI*radius;
}
```

In general, the inputs to a program or a function are stored in program variables. A program variable may be:

- A variable declared in a program as in the *C* declarations

For example: int base; char s[];

- Resulting from a read statement or similar interaction with the environment, For example: scanf(‘‘%d\n’’, &x);

4.2.2 Black Box and White Box Test Case Selection Strategies

- **Black box Testing**: In this method, where test cases are derived from the functional specification of the system; and
- **White box Testing**: In this method, where test cases are derived from the internal design specifications or actual code (Sometimes referred to as Glass-box).

Black box test case selection can be done without any reference to the program design or the program code. Test case selection is only concerned with the functionality and features of the system but not with its internal operations.

- The real advantage of black box test case selection is that it can be done *before* the design or coding of a program. Black box test cases can also help to get the design and coding correct with respect to the specification. Black box testing methods are good at testing for missing functions or program behavior that deviates from the specification. Black box testing is ideal for evaluating products that you intend to use in your systems.

- The main disadvantage of black box testing is that black box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that need to be safe (additional code may interfere with the safety of the system) or secure (additional code may be used to break security).

White box test cases are selected using the specification, design and code of the program or functions under test. This means that the testing team needs access to the internal designs or code for the program.

- The chief advantage of white box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. White box testing can check additional functions or code that has been implemented, but not specified.

- The main disadvantage of white box testing is that you must wait until after design and coding of the programs of functions under test have been completed in order to select test cases.

Methods for Black box testing strategies

A number of test case selection methods exist within the broad classification of black box and white box testing.

For Black box testing strategies, the following are the methods:

- Boundary-value Analysis;
- Equivalence Partitioning.

We will also study State Based Testing, which can be classified as opaque box selection strategies that is somewhere between black box and white box selection strategies.

Boundary-value-analysis

The basic concept used in Boundary-value-analysis is that if the specific test cases are designed to check the boundaries of the input domain then the probability of detecting an error will increase. If we want to test a program written as a function F with two input variables x and y., then these input variables are defined with some boundaries like $a1 \leq x \leq a2$ and $b1 \leq y \leq b2$. It means that inputs x and y are bounded by two intervals [a1, a2] and [b1, b2].

Test Case Selection Guidelines for Boundary Value Analysis

The following set of guidelines is for the selection of test cases according to the principles of boundary value analysis. The guidelines do not constitute a firm set of rules for every case. You will need to develop some judgement in applying these guidelines.

1. If an *input* condition specifies a range of values, then construct valid test cases for the ends of the range, and invalid input test cases for input points just beyond the ends of the range.
2. If an *input* condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
3. If an *output* condition specifies a range of values, then construct valid test cases for the ends of the output range, and invalid input test cases for situations just beyond the ends of the output range.
4. If an *output* condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
5. If the input or output of a program is an ordered set (e.g., a sequential file, linear list, table), focus attention on the first and last elements of the set.

Example 1: Boundary Value Analysis for the Triangle Program

Consider a simple program to classify a triangle. Its input consists of three positive integers (say x, y, z) and the data types for input parameters ensures that these will be integers greater than zero and less than or equal to 100. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled.

Solution: Following possible boundary conditions are formed:

1. Given sides *(*A; B; C) for a scalene triangle, the sum of any two sides is greater than the third and so, we have boundary conditions $A + B > C$, $B + C > A$ and $A + C > B$.
2. Given sides *(*A; B; C) for an isosceles triangle two sides must be equal and so we have boundary conditions $A = B$, $B = C$ or $A = C$.
3. Continuing in the same way for an equilateral triangle the sides must all be of equal length and we have only one boundary where $A = B = C$.
4. For right-angled triangles, we must have $A^2+B^2 = C^2$.

On the basis of the above boundary conditions, test cases are designed as follows (*Table 4.1*):

Table 4.1: Test cases for Example-1

Test case	x	y	z	Expected Output
1	100	100	100	Equilatera*l* triangle
2	50	3	50	Isosceles triangle
3	40	50	40	Equilatera*l* triangle
4	3	4	5	Right-angled triangles
5	10	10	10	Equilatera*l* triangle
6	2	2	5	Isosceles triangle
7	100	50	100	Scalene triangle
8	1	2	3	Non-triangles
9	2	3	4	Scalene triangle
10	1	3	1	Isosceles triangle

Equivalence Partitioning

Equivalence Partitioning is a method for selecting test cases based on a partitioning of the input domain. The aim of equivalence partitioning is to divide the input domain of the program or module into classes (sets) of test cases that have a similar effect on the program. The classes are called Equivalence classes.

Equivalence Classes

An Equivalence *Class* is a set of inputs that the program treats identically when the program is tested. In other words, a test input taken from an equivalence class is representative of all of the test inputs taken from that class. Equivalence classes are determined from the specification of a program or module. Each equivalence class is used to represent certain conditions (or predicates) on the input domain. For equivalence partitioning it is usual to also consider valid and invalid inputs. The terms input condition, valid and invalid inputs, are not used consistently. But, the following definition spells out how we will use them in this subject. An input condition on the input domain is a predicate over the values of the input domain. A *Valid* input to a program or module is an element of the input domain that is expected to return a non-error value. An *Invalid* input is an input that is expected to return an error value. Equivalence partitioning is then a systematic method for identifying interesting input conditions to be tested. An input condition can be applied to a set of values of a specific input variable, or a set of input variables
as well.

A Method for Choosing Equivalence Classes

The aim is to minimize the number of test cases required to cover all of the identified equivalence classes. The following are two distinct steps in achieving this goal:

Step 1: Identify the equivalence classes

If an input condition specifies a range of values, then identify one valid equivalence class and two invalid equivalence classes.
For example, if an input condition specifies a range of values from 1 to 99, then, three equivalence classes can be identified:

- One valid equivalence class: $1 < X < 99$
- Two invalid equivalence classes $X < 1$ and $X > 99$

Step 2: Choose test cases

The next step is to generate test cases using the equivalence classes identified in the previous step. The guideline is to choose test cases on the boundaries of partitions and test cases close to the midpoint of the partition. In general, the idea is to select at least one element from each equivalence class.

Example 2: Selecting Test Cases for the Triangle Program

In this example, we will select a set of test cases for the following triangle program based on its specification. Consider the following informal specification for the Triangle Classification Program. The program reads three integer values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled. The specification of the triangle classification program lists a number of inputs for the program as well as the form of output. Further, we require that each of the inputs *"must be*" a positive integer. Now, we can determine valid and invalid equivalence classes for the input conditions. Here, we have a range of values. If the three integers we have called x, y and z are all greater than zero, then, they are valid and we have the equivalence class.

ECvalid = f(x,y, z) x > 0 and y > 0 and z > 0.

For the invalid classes, we need to consider the case where each of the three variables in turn can be negative and so we have the following equivalence classes:

ECInvalid1 = f(x, y, z) x < 0 and y > 0 and z > 0
ECInvalid2 = f(x, y, z) x > 0 and y <0 and z > 0
ECInvalid3 = f(x, y, z) x > 0 and y > 0 and z < 0

Note that we can combine the valid equivalence classes. But, we are not allowed to combine the invalid equivalence classes. The output domain consists of the text 'strings' 'isosceles', 'scalene', 'equilateral' and 'right-angled'. Now, different values in the input domain map to different elements of the output domain to get the equivalence classes in *Table 4.2*. According to the equivalence partitioning method we only need to choose one element from each of the classes above in order to test the triangle program.

Table 4.2: The equivalence classes for the triangle program

Equivalence class	Test Inputs	Expected Outputs
ECscalene	f(3, 5, 7), . . . g	"Scalene"
ECisosceles	f(2, 3, 3), . . . g f(2, 3, 3), . . g	"Isosceles"
ECequilateral	f(7, 7, 7), . . . g f(7, 7, 7), . . . g	"Equilateral"
ECright angled	f(3, 4, 5), . . .	"Right Angled"
ECnon _ triangle	f(1, 1, 3), . . .	"Not a Triangle"
ECinvalid1	f(-1, 2, 3), (0, 1, 3), . . .	"Error Value"
ECinavlid2	f(1, -2, 3), (1, 0, 3), . . .	"Error Value"
ECinavlid3	f(1, 2, -3), (1, 2, 0),. . .	"Error Value"

Methods for White box testing strategies

In this approach, complete knowledge about the internal structure of the source code is required. For *White-box testing strategies,* the methods are:

1. Coverage Based Testing
2. Cyclomatic Complexity
3. Mutation Testing

Coverage based testing

The aim of coverage based testing methods is to *'cover*' the program with test cases that satisfy some fixed coverage criteria. Put another way, we choose test cases to exercise as much of the program as possible according to some criteria.

Coverage Based Testing Criteria

Coverage based testing works by choosing test cases according to well-defined 'coverage' criteria. The more common coverage criteria are the following.

- **Statement Coverage or Node Coverage:** Every statement of the program should be exercised at least once.
- **Branch Coverage or Decision Coverage:** Every possible alternative in a branch or decision of the program should be exercised at least once. For *if* statements, this means that the branch must be made to take on the values *true* or *false*.
- **Decision/Condition Coverage**: Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false.
- **Multiple condition coverage**: All possible combinations of condition outcomes within each branch should be exercised at least once.
- **Path coverage**: Every execution path of the program should be exercised at least once.

In this section, we will use the control flow graph to choose white box test cases according to the criteria above. To motivate the selection of test cases, consider the simple program given in Program 4.1.

Example 3:

```
void main(void)
{
int x1, x2, x3;
scanf("%d %d %d", &x1, &x2, &x3);
if ((x1 > 1) && (x2 == 0))
x3 = x3 / x1;
if ((x1 == 2) || (x3 > 1))
x3 = x3 + 1;
while (x1 >= 2)
x1 = x1 - 2;
printf("%d %d %d", x1, x2, x3);
}
```

Program 4.1: A simple program for white box testing

The first step in the analysis is to generate the flow chart, which is given in Figure 4.1. Now what is needed for statement coverage? If all of the branches are true, at least once, we will have executed every statement in the flow chart. Put another way to execute every statement at least once, we must execute the path ABCDEFGF. Now, looking at the conditions inside each of the three *branches*, we derive a set of constraints on the values of x1, x2 and x3 such that all the three branches are extended. A test case of the form (x1; x2; x3) = (2; 0; 3) will execute all of the statements in the program.

Note that we need not make every branch evaluate to true and false, nor have we make every condition evaluate to true and false, nor we traverse every path in the program.

To make the first branch true, we have test input (2; 0; 3) that will make all of the branches true. We need a test input that will now make each one false. Again looking at all of the conditions, the test input (1; 1; 1) will make all of the branches false.

For any of the criteria involving condition coverage, we need to look at each of the five conditions in the program: C1 = (x1>1), C2 = (x2 == 0), C3 = (x1 == 2), C4 = (x3>1) and C5 = (x1 >= 2). The test input (1; 0; 3) will make C 1 false, C2 true, C3 false, C4 true and C5 false.

Examples of sets of test inputs and the criteria that they meet are given in Table 4.3. The set of test cases meeting the multiple condition criteria is given in Table 4.4. In the table, we let the branches B1 = C1&&C2, B2 = C3||C4 and B3 = C5.

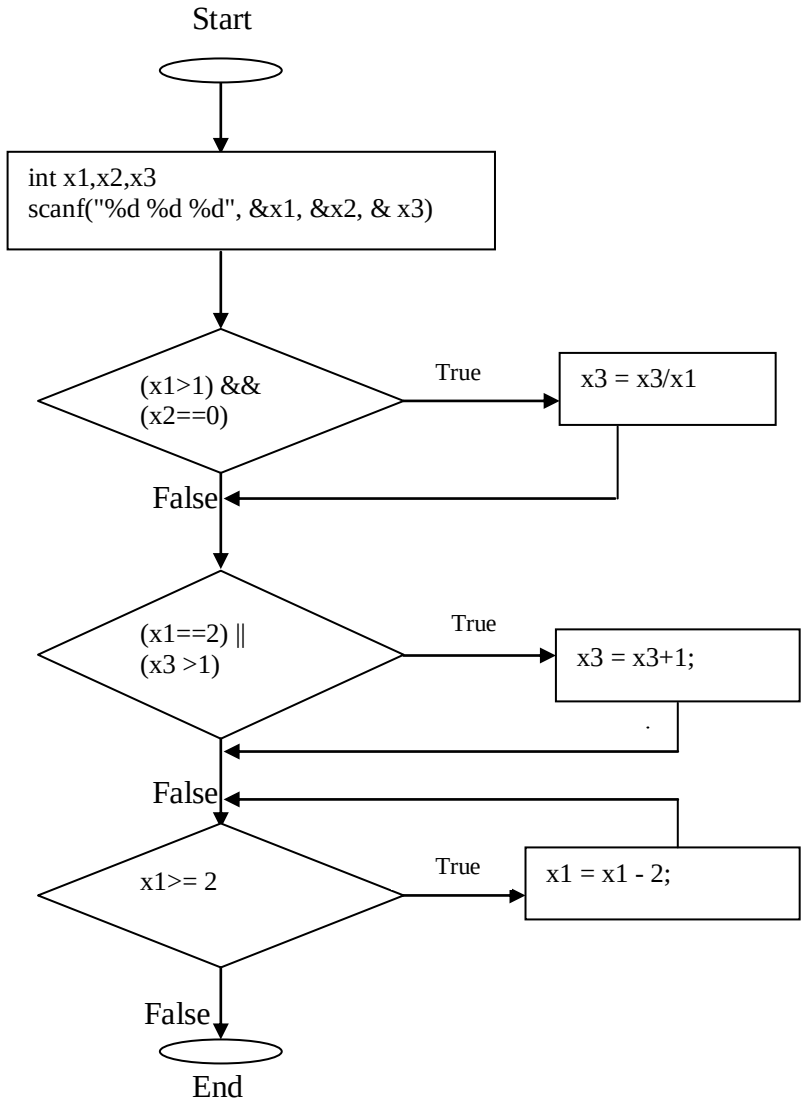

Figure 4.1: The flow chart for the program 4.1

Table 4.3: Test cases for the various coverage criteria for the program 4.1

Coverage Criteria	**Test Inputs (x1, x2, x3)**	**Execution Paths**
Statement	(2, 0, 3)	ABCDEFGF
Branch	(2, 0, 3), (1, 1, 1)	ABCDEFGF ABDF
Condition	(1, 0, 3), (2, 1, 1)	ABDEF ABDFGF
Decision/ Condition	(2, 0, 4), (1, 1, 1)	ABCDEFGF ABDF
Multiple Condition	(2, 0, 4), (2, 1, 1), (1, 0, 2), (1, 1, 1)	ABCDEFGF ABDEFGF ABDEF ABDF
Path	(2, 0, 4), (2, 1, 1), (1, 0, 2), (4, 0, 0),	ABCDEFGF ABDFGF ABDEF ABCDFGFGF

Test cases	C1 x1 > 1	C2 x2==0	B1	C3 x1==2	C4 x3 > 1	B2	B3 C5 x1 ≥ 2
(1,0,3)	F	T	F	F	T	T	F
(2,1,1)	T	F	F	T	F	F	T
(2,0,4)	T	T	T	T	T	T	T
(1,1,1)	F	F	F	F	F	F	F
(2,0,4)	T	T	T	T	T	T	T
(2,1,1)	T	F	F	T	F	T	T
(1,0,2)	F	T	F	F	T	T	F
(1,1,1)	F	F	F	F	F	F	F

4.2.3 Cyclomatic Complexity

Control flow graph (CFG)

A control flow graph describes the sequence in which different instructions of a program get executed. It also describes how the flow of control passes through the program. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. Following structured programming constructs are represented as CFG:

Figure 4.2 : sequence

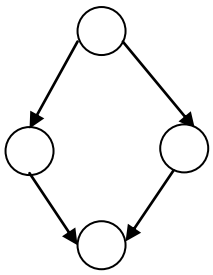

Figure 4.3 : if -else

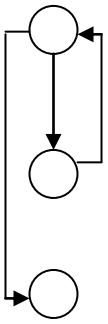

Figure 4.4 : while-loop

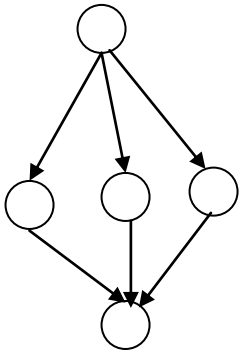

Figure 4.5: case

Example 4.4: Draw CFG for the program given below.

```
int sample (a,b)
int a,b;
{
1       while (a!= b) {
2       if (a > b)
3       a = a-b;
4       else b = b-a;}
5       return a;
}
```

Program 4.2: A program

In the above program, two control constructs are used, namely, while-loop and if-then-else. A complete CFG for the program of Program 4.2 is given below: (*Figure 4.6*).

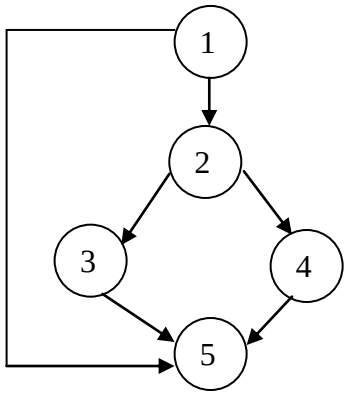

Figure 4.6: CFG for program 4.2

Cyclomatic Complexity: This technique is used to find the number of independent paths through a program. If CFG of a program is given, the Cyclomatic complexity V(G) can be computed as: V(G) = E – N + 2, where N is the number of nodes of the CFG and E is the number of edges in the CFG. For the example 4, the Cyclomatic Complexity = 6 – 5 + 2 = 3.

The following are the properties of Cyclomatic complexity:

- V(G) is the maximum number of independent paths in graph G
- Inserting and deleting functional statements to G does not affect V(G)
- G has only one path if and only if V(G) = 1.

Mutation Testing

Mutation Testing is a powerful method for finding errors in software programs. In this technique, multiple copies of programs are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program. The mutant that is detected by a test case is termed "killed" and the goal of the mutation procedure is to find a set of test cases that are able to kill groups of mutant programs. Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. It is essential that all mutants must be killed by the test cases or shown to be equivalent to the original expression. If we run a mutated program, there are two possibilities:

1. The results of the program were affected by the code change and the test suite detects it. We assumed that the test suite is perfect, which means that it must detect the change. If this happens, the mutant is called a killed mutant.

2. The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an equivalent mutant.

If we take the ratio of killed mutants to all the mutants that were created, we get a number that is smaller than 1. This number gives an indication of the sensitivity of program to the changes in code. In real life, we may not have a perfect program and we may not have a perfect test suite. Hence, we can have one more scenario:

3. The results of the program are different, but the test suite does not detect it because it does not have the right test case.

Consider the following program 4.3:

```
main(argc, argv)                                    /* line 1 */
int argc;                                         /* line 2 */
char *argv[];                                       /* line 3 */
{                                                   /* line 4 */
  int c=0;                                          /* line 5 */
                                                    /* line 6 */
    if(atoi(argv[1]) < 3){                        /* line 7 */
      printf("Got less than 3\n");                  /* line 8 */
      if(atoi(argv[2]) > 5)                         /* line 9 */
        c = 2;                                    /* line 10 */
    }                                               /* line 11 */
    else                                            /* line 12 */
      printf("Got more than 3\n");                /* line 13 */
   exit(0);                                         /* line 14 */
}                                                   /* line 15 */
```

Program 4.3: A program

The program reads its arguments and prints messages accordingly.

Now let us assume that we have the following test suite that tests the program:

Test case 1:
Input: 2 4
Output: Got less than 3
Test case 2:
Input: 4 4
Output: Got more than 3
Test case 3:
Input: 4 6
Output: Got more than 3
Test case 4:
Input: 2 6
Output: Got less than 3
Test case 5:
Input: 4
Output: Got more than 3

Now, let's mutate the program. We can start with the following simple changes:

Mutant 1: change line 9 to the form
`if(atoi(argv[2]) <= 5)`

Mutant 2: change line 7 to the form
`if(atoi(argv[1]) >= 3)`

Mutant 3: change line 5 to the form
`int c=3;`

If we take the ratio of all the killed mutants to all the mutants generated, we get a number smaller than 1 that also contains information about accuracy of the test suite. In practice, there is no way to separate the effect that is related to test suite inaccuracy and that which is related to equivalent mutants. In the absence of other possibilities, one can accept the ratio of killed mutants to all the mutants as the measure of the test suite accuracy. The manner by which a test suite is evaluated via mutation testing is as follows: For a specific test suite and a specific set of mutants, there will be three types of mutants in the code (i) killed or dead (ii) live (iii) equivalent. The score (evaluation of test suite) associated with a test suite T and mutants M is simply computed as follows:

$$\frac{\#\text{ killed Mutants}}{\#\text{ total mutants - \# equivalent mutants}} \times 100$$

☞ Check Your Progress 1

1) What is the use of Cyclomatic complexity in software development?

……

……

……

4.3 TESTING ACTIVITIES

Although testing varies between organisations, there is a cycle to testing:

Requirements Analysis: Testing should begin in the requirements phase of the software devlopment life cycle (SDLC).

Design Analysis: During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameters should the testers work.

Test Planning: Test Strategy, Test Plan(s).

Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.

Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

Retesting the Defects: Defects are once again tested to find whether they got eliminated or not.

Levels of Testing:

Mainly, Software goes through three levels of testing:

- Unit testing
- Integration testing
- System testing.

Unit Testing

Unit testing is a procedure used to verify that a particular segment of source code is working properly. The idea about unit tests is to write test cases for all functions or methods. Ideally, each test case is separate from the others. This type of testing is mostly done by developers and not by end users.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Unit testing provides a strict, written contract that the piece of code must satisfy. Unit testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, perfomance problems and any other system-wide issues. A unit test can only show the presence of errors; it cannot show the absence of errors.

Integration Testing

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing. Integration testing takes as its input, modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output, the integrated

system ready for system testing.The purpose of Integration testing is to verify functional, performance and reliability requirements placed on major design items.

System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). In system testing, the entire system can be tested as a whole against the software requirements specification (SRS). There are *rules* that describe the functionality that the vendor (developer) and a customer have agreed upon. System testing tends to be more of an *investigatory* testing phase, where the focus is to have a destructive attitude and test not only the design, but also the behavior and even the believed expectations of the customer. System testing is intended to test up to and beyond the bounds defined in the software requirements specifications.

Acceptance tests are conducted in case the software developed was a custom software and not product based. These tests are conducted by customer to check whether the software meets all requirements or not. These tests may range from a few weeks to several months.

4.4 DEBUGGING

Debugging occurs as a consequence of successful testing. Debugging refers to the process of identifying the cause for defective behavior of a system and addressing that problem. In less complex terms - fixing a bug. When a test case uncovers an error, debugging is the process that results in the removal of the error. The debugging process begins with the execution of a test case. The debugging process attempts to match symptoms with cause, thereby leading to error correction. The following are two alternative outcomes of the debugging:

1. The cause will be found and necessary action such as correction or removal will be taken.
2. The cause will not be found.

Characteristics of bugs

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Life Cycle of a Debugging Task

The following are various steps involved in debugging:

a) Defect Identification/Confirmation

- A problem is identified in a system and a defect report created
- Defect assigned to a software engineer
- The engineer analyzes the defect report, performing the following actions:
  - What is the expected/desired behaviour of the system?
  - What is the actual behaviour?
  - Is this really a defect in the system?
  - Can the defect be reproduced? (While many times, confirming a defect is straight forward. There will be defects that often exhibit quantum behaviour.)

b) Defect Analysis

Assuming that the software engineer concludes that the defect is genuine, the focus shifts to understanding the root cause of the problem. This is often the most challenging step in any debugging task, particularly when the software engineer is debugging complex software.

Many engineers debug by starting a debugging tool, generally a debugger and try to understand the root cause of the problem by following the execution of the program step-by-step. This approach may eventually yield success. However, in many situations, it takes too much time, and in some cases is not feasible, due to the complex nature of the program(s).

c) Defect Resolution

Once the root cause of a problem is identified, the defect can then be resolved by making an appropriate change to the system, which fixes the root cause.

Debugging Approaches

Three categories for debugging approaches are:

- Brute force
- Backtracking
- Cause elimination.

Brute force is probably the most popular despite being the least successful. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" technique, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. *Backtracking* is a common debugging method that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backwards till the error is found. In *Cause elimination*, a list of possible causes of an error are identified and tests are conducted until each one is eliminated.

☞ Check Your Progress 2

1) What are the different levels of testing and their goals? For each level specify which of the testing approaches are most suitable.

   ……..

   ……..

2) Mention the steps involved in the process of debugging.

   ……

   …….....

4.5 TESTING TOOLS

The following are different categories of tools that can be used for testing:

- **Data Acquisition**: Tools that acquire data to be used during testing.
- **Static Measurement**: Tools that analyse source code without executing test cases.
- **Dynamic Measurement**: Tools that analyse source code during execution.
- **Simulation**: Tools that simulate functions of hardware or other externals.
- **Test Management**: Tools that assist in planning, development and control of testing.
- **Cross-Functional tools**: Tools that cross the bounds of preceding categories.

The following are some of the examples of commercial software testing tools:

Rational Test Real Time Unit Testing

- **Kind of Tool**

  Rational Test RealTime's Unit Testing feature automates C, C++ software component testing.

- **Organisation**

  IBM Rational Software

- **Software Description**

Rational Test RealTime Unit Testing performs black-box/functional testing, i.e., verifies that all units behave according to their specifications without regard to how that functionality is implemented. The Unit Testing feature has the flexibility to naturally fit any development process by matching and automating developers' and testers' work patterns, allowing them to focus on value-added tasks. Rational Test RealTime is integrated with native development environments (Unix and Windows) as well as with a large variety of cross-development environments.

- **Platforms**

Rational Test RealTime is available for most development and target systems including Windows and Unix.

AQtest

- **Kind of Tool**

  Automated support for functional, unit, and regression testing

- **Organisation**

  AutomatedQA Corp.

- **Software Description**

AQtest automates and manages functional tests, unit tests and regression tests, for applications written with VC++, VB, Delphi, C++Builder, Java or VS.NET. It also supports white-box testing, down to private properties or methods. External tests can be recorded or written in three scripting languages (VBScript, JScript, DelphiScript). Using AQtest as an OLE server, unit-test drivers can also run it directly from application code. AQtest automatically integrates AQtime when it is on the machine. Entirely COM-based, AQtest is easily extended through plug-ins using the complete IDL libraries supplied. Plug-ins currently support Win32 API calls, direct ADO access, direct BDE access, etc.

- **Platforms**

Windows 95, 98, NT, or 2000.

csUnit

- **Kind of Tool**

  "Complete Solution Unit Testing" for Microsoft .NET (freeware)

- **Organisation**

  csUnit.org

- **Software Description**

  csUnit is a unit testing framework for the Microsoft .NET Framework. It targets test driven development using .NET languages such as C#, Visual Basic .NET, and managed C++.

- **Platforms**

  Microsoft Windows

Sahi

http://sahi.sourceforge.net/

Software Description

Sahi is an automation and testing tool for web applications, with the facility to record and playback scripts. Developed in Java and JavaScript, it uses simple JavaScript to execute events on the browser. Features include in-browser controls, text based scripts, Ant support for playback of suites of tests, and multi-threaded playback. It supports HTTP and HTTPS. Sahi runs as a proxy server and the browser needs to use the Sahi server as its proxy. Sahi then injects JavaScript so that it can access elements in the webpage. This makes the tool independant of the website/ web application.

- **Platforms**

OS independent. Needs at least JDK1.4

4.6 SUMMARY

The importance of software testing and its impact on software is explained in this unit. Software testing is a fundamental component of software development life cycle and represents a review of specification, design and coding. The objective of testing is to have the highest likelihood of finding most of the errors within a minimum amount of time and minimal effort. A large number of test case design methods have been developed that offer a systematic approach to testing to the developer.

Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational. A strategy for software testing may be to move upwards along the spiral. Unit testing happens at the vortex of the spiral and concentrates on each unit of the software as implemented by the source code. Testing happens upwards along the spiral to integration testing, where the focus is on design and production of the software architecture. Finally, we perform system testing, where software and other system elements are tested together.

Debugging is not testing, but always happens as a response of testing. The debugging process will have one of two outcomes:

1) The cause will be found, then corrected or removed, or

2) The cause will not be found. Regardless of the approach that is used, debugging has one main aim: to determine and correct errors. In general, three kinds of debugging approaches have been put forward: Brute force, Backtracking and Cause elimination.

4.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Cyclomatic Complexity is asoftware metric that provides a quantitative measure of the logical complexity of a program. When it is 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. It also provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Check Your Progress 2

1) The basic levels of testing are: unit testing, integration testing, system testing and acceptance testing.

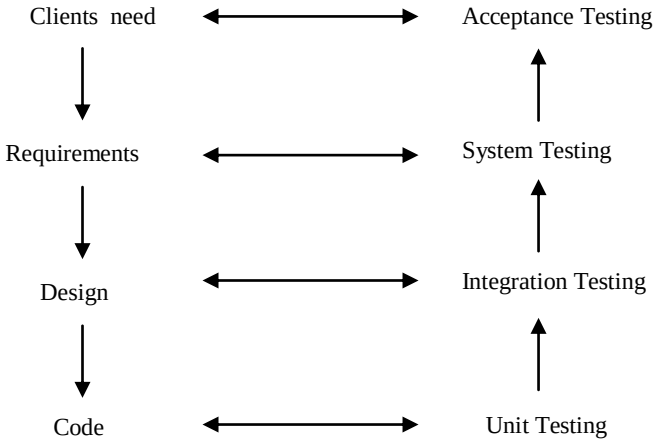

Figure 4.7: Testing levels

For unit testing, structural testing approach is best suited because the focus of testing is on testing the code. In fact, structural testing is not very suitable for large programs. It is used mostly at the unit testing level. The next level of testing is integration testing and the goal is to test interfaces between modules. With integration testing, we move slowly away from structural testing and towards functional testing. This testing activity can be considered for testing the design. The next levels are system and acceptance testing by which the entire software system is tested. These testing levels focus on the external behavior of the system. The internal logic of the program is not emphasized. Hence, mostly functional testing is performed at these levels.

2) The various steps involved in debugging are:

- Defect Identification/Confirmation
- Defect Analysis
- Defect Resolution

4.8 FURTHER READINGS

1) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.

2) *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

3) An *Integrated approach to Software Engineering*, Pankaj Jalote; Narcosis Publishing House.

Reference websites

http://www.rspa.com
http://www.ieee.org
http://standards.ieee.org
http://www.ibm.com
http://www.opensourcetesting.org