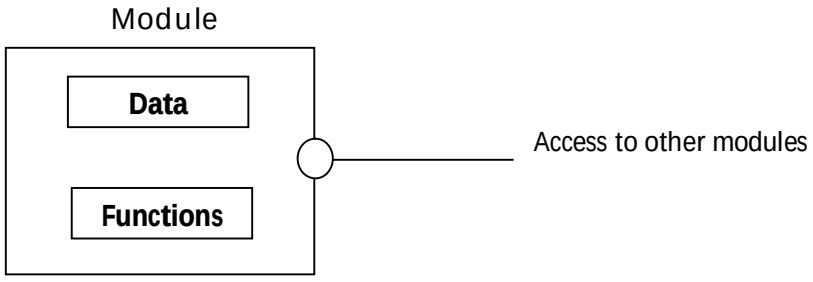
Coding and Testing

6.1 Programming Principles and Guidelines

- The software engineer translates the design into source code.
- The main goal of programming is to produce quality source codes that can reduce the cost of testing and maintenance.
- A clear, readable and understandable source code will make other tasks easier.
- The quality of source code depends on the skills and expertise of software engineers.
- The following programming principles can make a code clear, readable and understandable.
  - Information hiding
  - Structured programming
  - Coding standards

6.1.1 Information Hiding

- Information or data hiding is a programming concept which protects the data from direct modification by other parts of the program.
- It can be achieved using data encapsulation.
- Binding the data and operations that operate on the data into a single unit is called as data encapsulation. This is shown below.



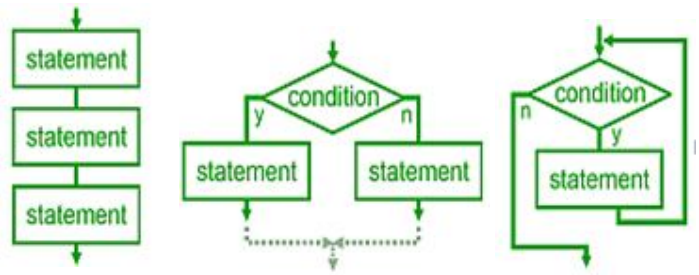
- The above figure shows that the functions within the module can only access the data in the module. Functions outside the module cannot

access the data because the data is not visible to these functions. However, the module may provide an interface through which other functions can access the data present in the module.

- In java, information hiding can be achieved by an object or class.

6.1.2 Structured Programming

- The basic objective of coding activity is to produce programs that are easy to understand.
- Structured programming helps to develop programs that are easier to understand.
- Every structured programming consists of the following three features
  - Sequence – statements in a program are executed sequentially one after another.
  - Selection – based on the result of the condition, statements are executed.
  - Iteration – based on the condition, group of statements are executed repeatedly.
- The following figure shows these features.



Sequence Selection Iteration

6.1.3 Coding Standards

- Programmers spend considerable amount of time reading the code during debugging and enhancement.
- Therefore, it is of prime importance to write code in a manner that is easy to read and understand.

- Coding standards provide rules and guidelines in order to make code easier to read.
- Most organizations that develop software regularly develop their own standards.
- In general, coding standards provide guidelines for programmers regarding naming, file organization, statements and declarations, and comments.

Naming

- Package names should be in lowercase (e.g., mypackage, edu.iitk.maths).
- Type names should be nouns and should start with uppercase (e.g., Day, DateOfBirth, EventHandler).
- Variable names should be nouns starting with lowercase (e.g., name, amount).
- Constant names should be all uppercase (e.g., PI, MAX ITERATIONS).
- Method names should be verbs starting with lowercase
- Loop iterators should be named i, j, k, etc.
- The prefix is should be used for Boolean variables and methods
- Exception classes should be suffixed with Exception (e.g., OutOfBoundException).

Files

- Java source files should have the extension .java
- File name should be same as the outer class name

Statements

- Variables should be initialized where declared
- Declare related variables together in a common statement.
- Class variables should never be declared public.
- Avoid the use of break and continue in a loop.

Comments

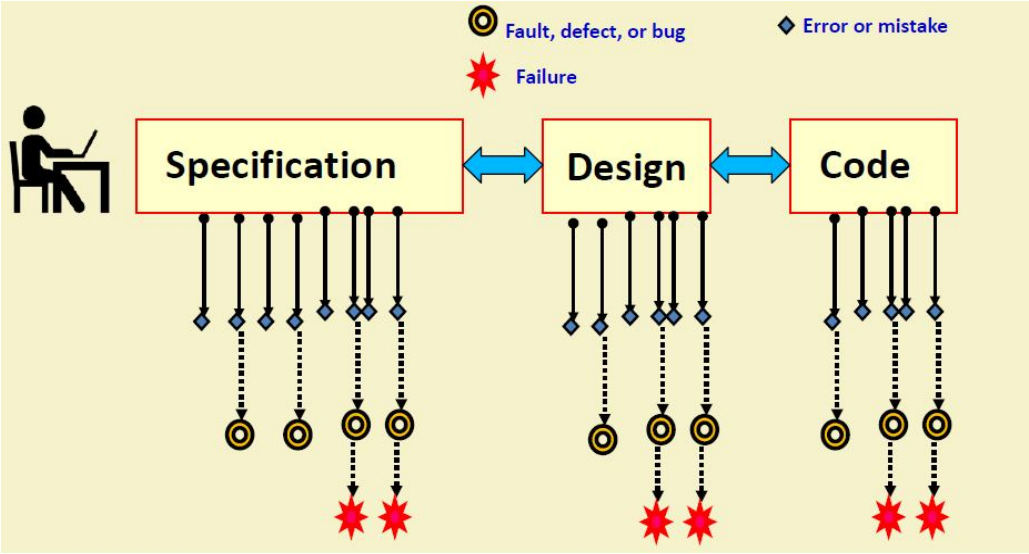
- Single line comments for a block of code.
- There should be comments for all major variables explaining what they represent.

6.2 Testing Concepts

- Testing is the process of executing a program with the intent of finding errors.
- In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

6.2.1 Error, Fault, and Failure

- **Error** - People make errors. Errors are also called as mistakes. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.
- **Fault** - is the representation of an error. That is errors may lead to faults. Fault is also called as defect or bug. Not all errors lead to bugs.
- **Failure** - A failure occurs when a fault executes.
- The following figure shows the relationship among error, fault and failure. It shows that some of the errors may lead to faults and some of the faults may lead to failures.



- For example consider the statement: **x<=50**. But, the programmer mistakenly written it as **x<=500**. Assuming that x never gets a value above 50. In such situation, though it is an error but, it will not lead to a fault.

6.2.2 Test Case, Test Suite and Test Harness

- A program is tested using a set of carefully designed test cases.
- **Test Case** - A test case is a triplet [I,S,O]
  - Where I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.
- For example, to test the functionality Renew Book in Library Management System, the test case would be
  - **Set the program in the required state:** Book record created, member record created, Book issued
  - **Give the defined input:** Select renew book option and request renew for a further 2 week period.
  - **Observe the output**: Compare it to the expected output.
- **Test Suite** - The set of all test cases is called the test suite

Why design test cases?

- Exhaustive testing of any non-trivial system is impractical because, input data domain is extremely large.
- Therefore design an optimal test suite of reasonable size that uncovers as many errors as possible.
- Consider the following example. The code has a simple programming error:

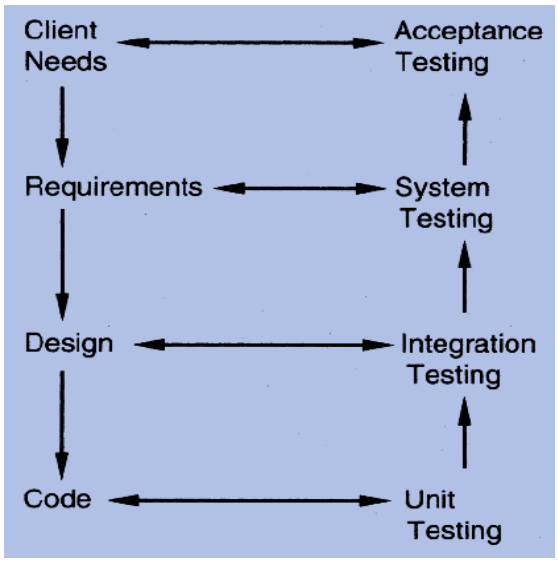
```
If (x>y) max = x;
    else max = x; / / should be max=y;
```

- Test suite {(x=3,y=2);(x=2,y=3)} can detect the bug
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)}does not detect the bug.
- **Test Harness**
  - To have a test suite executed automatically, we will need a framework in which test inputs can be defined representing test cases, the automated test script can be written, the SUT can be executed by this script, and the result of entire testing reported to the tester.

- Many testing frameworks now exist that permit all this to be done in a simple manner.
- A testing framework is called as **test harness**

6.2.3 Levels of Testing

- Software is tested at 4 levels of testing.
  1. Unit testing
  2. Integration testing
  3. System testing and
  4. Acceptance testing
- **Unit testing** - Test each module (unit, or component) independently. Mostly done by the developers of the modules.
- **Integration testing** - many unit tested modules are combined into subsystems, which are then tested. Here, the emphasis is on testing the interfaces between the modules.
- **System testing and Acceptance testing**
  - Here the entire software system is tested.
  - The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements.
  - Acceptance testing is often performed by the clients or customers to demonstrate that the software is working satisfactorily.
- The following figure shows levels of testing.



6.3 Testing Process

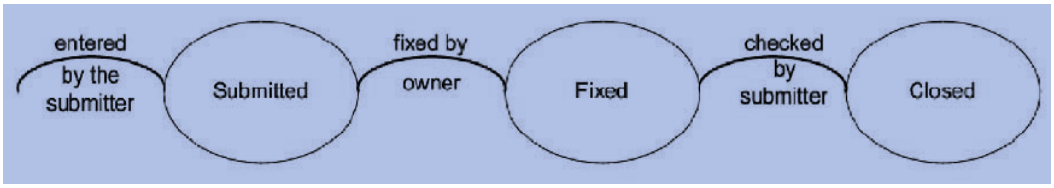
- The testing process for a project consists of three high-level tasks.
    1. Test planning
    2. Test case design and
    3. Test execution

1. **Test Plan -** Testing commences with a test plan. A test plan should contain the following:
    - Test unit specification
    - Features to be tested
    - Approach for testing
    - Test deliverables
    - Schedule and task allocation

- *Test unit specification* – A test unit is a set of one or more modules that form software under test (SUT).
- *Features to be tested* - include all software features and combinations of features that should be tested. They typically include functional requirements, performance requirements and design constraints.
- *Approach for testing* – specifies testing type for executing the test cases. It can be a block box or white box testing.

- *Test deliverables* - Could be a list of test cases that were used, detailed results of testing including the list of defects found, test summary report, and data about the code coverage.
- *Schedule and Task allocation* – specifies schedule for doing testing and the persons who are responsible for doing the testing.

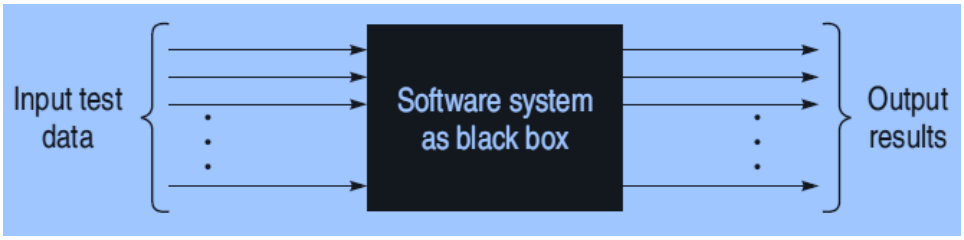
2. **Test Case Design** - In this set of test cases are designed for performing the testing. Test cases can be designed using black box or white box testing techniques.
3. **Test Case Execution** - With the specification of test cases, the next step in the testing process is to execute them. Testing tools can be used to execute test cases. During test case execution, defects are found. These defects are then fixed and testing is done again to verify the fix. The following figure shows the life cycle of a defect.



The figure shows that the defects found during the testing are submitted to the module developer at which stage the defect is said to be in *Submitted* state. Then, the developer debugs the program and fixes the bug in the program. At this time the defect is said to be in *Fixed* state. After this, the tester again tests the program to see whether bug fixation is over. If it is over then the bug is said to be in the *Closed* state.

6.4 Black-box Testing

- In black-box testing technique, test cases are designed based on functional specifications.
- Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software.
- In black-box testing, the tester has no idea of the internal structure of the software, so, it is called as black-box testing.



- Since, we test the functionalities of the software, so, black-box testing is also called as functional testing.
- Number of testing strategies is used to design the test cases in black-box testing.
  - Scenario coverage
  - **Boundary value analysis**
  - **Equivalence class partitioning**
  - **Cause-effect (Decision Table) testing**
  - Combinatorial testing
  - Orthogonal array testing

6.4.1 Boundary Value Analysis (BVA)

- Some typical programming errors occur at the boundaries of input values. Programmers often commit mistakes at the boundaries of input values.
- Programmers may improperly use <instead of <=
- Boundary value analysis selects test cases at the boundaries of input values. There are two variations in BVA.
  - Boundary Value Checking (BVC)
  - Robust Testing Method

Boundary Value Checking (BVC)

- In this, test cases are designed based on

  (a) Minimum value (Min)

  (b) Value just above the minimum value (Min+)

  (c) Maximum value (Max)

  (d) Value just below the maximum value (Max−)

  (e) Nominal value
- For example, for n variables, 4n+1 test cases can be designed

Example 1: **let there is a variable x with range [-3, 10]**

Test cases: **-3, -2, 10, 9, 0**

Example 1: let there is a variable y with range [5, 100]

Test cases: **5, 6, 100, 99, 50**

Robust Testing Method

- In this, test cases are designed based on

  (a) Minimum value (Min)

  (b) A value just less than Minimum value (Min−)

  (c) Value just above the minimum value (Min+)

  (d) Maximum value (Max)

  (e) Value just below the maximum value (Max−)

  (f) A value just greater than the Maximum value (Max+)

  (g) Nominal value
- In this, for n variables, 6n+1 test cases can be designed

  Example 1: **let there is a variable x with range [-3, 10]**

  Test cases: **-3, -4, -2, 10, 9, 11, 0**

  Example 1: let there is a variable y with range [5, 100]

  Test cases: **5, 4, 6, 100, 99, 101, 50**

6.4.2 Equivalence Class Partitioning

- The input values to a program are partitioned into equivalence classes
- Few guidelines for determining the equivalence classes can be given as
  - If an input is a range, one valid and two invalid equivalence classes are defined. Example: 1 to 100
  - If an input is a set, one valid and one invalid equivalence classes are defined. Example: {a, b, c}
  - If an input is a Boolean value, one valid and one invalid class are defined.

Example 1

A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class partitioning testing technique.

Solution

First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. This is shown below.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Example 2

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$1 \le mm \le 12$

$1 \le dd \le 31$

$1900 \le yyyy \le 2025$

Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

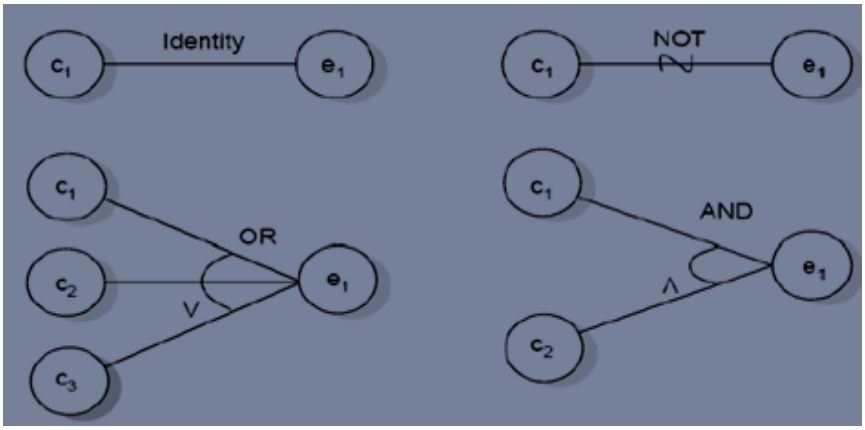
$$I_1 = \{<m, d, y> : 1 \le m \le 12\}$$
$$I_2 = \{<m, d, y> : 1 \le d \le 31\}$$
$$I_3 = \{<m, d, y> : 1900 \le y \le 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

6.4.3 Cause Effect Graphing Technique

- Causes & effects in the specifications are identified.
  - A cause is a distinct input condition.
  - An effect is an output condition or a system transformation.
- The semantic content of the specification is analysed and transformed into a Boolean graph linking the causes & effects.
- The graph is then converted to a decision table. Each row in the table represents a test case.
- The basic notations of the graph are shown below.



Example 1

The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message *x is issued. If* the character in column 2 is not a digit, message y is issued. Design the cause-effect graph for this scenario.

Solution

First identify the causes and effects in the problem statement.

The causes are

c_1: character in column 1 is A

c_2: character in column 1 is B

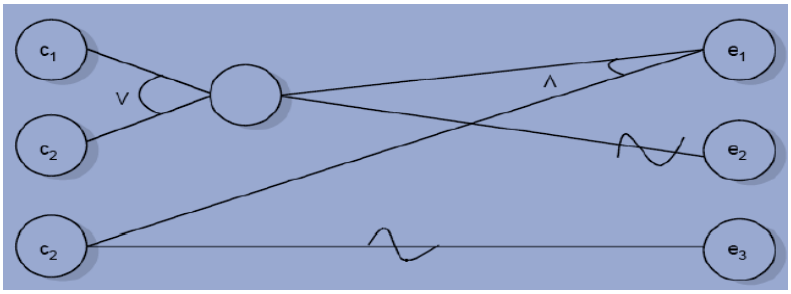
c_3: character in column 2 is a digit

and the effects are

e_1: update made

e_2: message *x* is issued

e_3: message *y* is issued



Example 2

Consider the following scenario.

If depositing less than Rs. 1 Lakh, rate of interest:

- 6% for deposit upto 1 year
- 7% for deposit over 1 year but less than 3 yrs
- 8% for deposit 3 years and above

If depositing more than Rs. 1 Lakh, rate of interest:

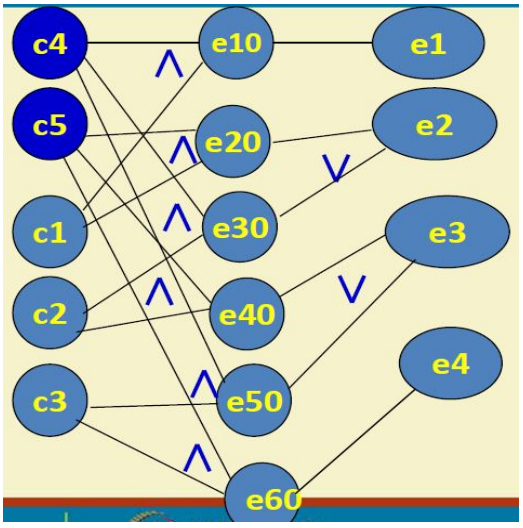
- 7% for deposit upto 1 year
- 8% for deposit over 1 year but less than 3 yrs
- 9% for deposit 3 years and above

Design the test cases for this scenario using cause-effect graphing technique.

Solution

Identify the causes and effects and draw the graph.

Causes	Effects
C1: Deposit<1yr	e1: Rate 6%
C2: 1yr<Deposit<3yrs	e2: Rate 7%
C3: Deposit>3yrs	e3: Rate 8%
C4:Deposit <1 Lakh	e4: Rate 9%
C5: Deposit >=1Lakh	



C1	C2	C3	C4	C5	e1	e2	e3	e4

1	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	0	0	1	0	0	1	0
0	0	1	1	0	0	0	1	0
0	0	1	0	1	0	0	0	1

Each row in the table is considered as a test case.

6.5 White-box Testing

- In white box testing, test cases are designed based on the code structure of the program. Because of this it is also called as structural testing.
- In this, test cases are designed to cover the entire program.
- Several white-box testing strategies have become very popular.
  - Statement coverage
  - Branch coverage
  - **Path coverage**
  - Condition coverage
  - MC/DC coverage
  - **Mutation testing**
  - **Data flow-based testing**

6.5.1 Control Flow Testing (or Path Testing)

- In this, design test cases such that:
  - ✓ All linearly independent paths in the program are executed at least once.
- Defined in terms of Control flow graph (CFG) of a program.
- To understand the path coverage based testing: We need to learn how to draw control flow graph of a program.

- A control flow graph (CFG) describes: The sequence in which different instructions of a program get executed and the way control flows through the program.

How to Draw Control Flow Graph?

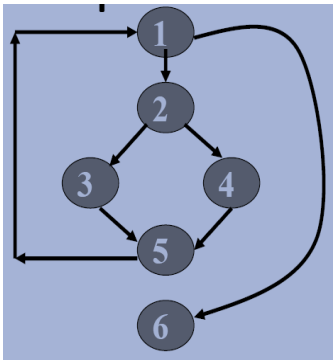
- Number all statements of a program.
- Numbered statements: Represent nodes of control flow graph.
- Draw an edge from one node to another node: If execution of the statement representing the first node can result in transfer of control to the other node.

Example

Consider the following program.

```
int f1(int x,int y){
1 while (x != y){
2   if (x>y) then
3       x=x-y;
4   else y=y-x;
5 }
6 return x;      }
```

Program



Control-flow graph

Once the control-flow graph is drawn, then we can find the number of independent paths in it. The number of independent paths present in the above graph is 3 as given below.

- ✓ **1,6** test case (**x=1, y=1**)
- ✓ **1,2,3,5,1,6** test case(**x=1, y=2**)
- ✓ **1,2,4,5,1,6** test case(**x=2, y=1**)

McCab's Cyclomatic Complexity

- It is straight forward to identify linearly independent paths of simple programs. However, for complicated programs it is not easy to determine the number of independent paths.
- It provides a practical way of determining the number of linearly independent paths of a program.
- Given a control flow graph G,

Cyclomatic complexity V(G) is given as

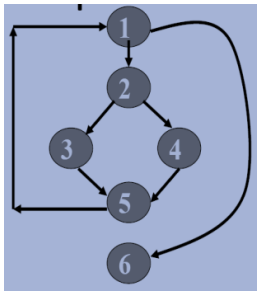
V(G)= E-N+2

Where N is the number of nodes in G

E is the number of edges in G

Example

Consider the following control-flow graph.



Cyclomatic complexity V(G) is

V(G) = 7-6+2

=3

Another way of computing cyclomatic complexity is based on number of bounded areas in the graph.

V(G) = Total number of bounded areas + 1

For the above graph V(G) = 2+1 = 3

Another way of computing cyclomatic complexity is based on number of predicates (or decisions) in the graph.

V(G) = Number of predicates + 1

For the above graph V(G) = 2+1 = 3

6.5.2 Data Flow Testing

- Data flow testing uses the control flow graph to explore the unreasonable things that can happen to data (data flow anomalies).
- Data flow anomalies are detected based on the associations between values and variables. Some anomalies are
  - o Variables are used without being initialized
  - o Initialized variables are not used once

Data Object Categories

(d) Defined, Created, Initialized

(k) Killed, Undefined, Released

(u) Used:

– (c) Used in a calculation

– (p) Used in a predicate

Example

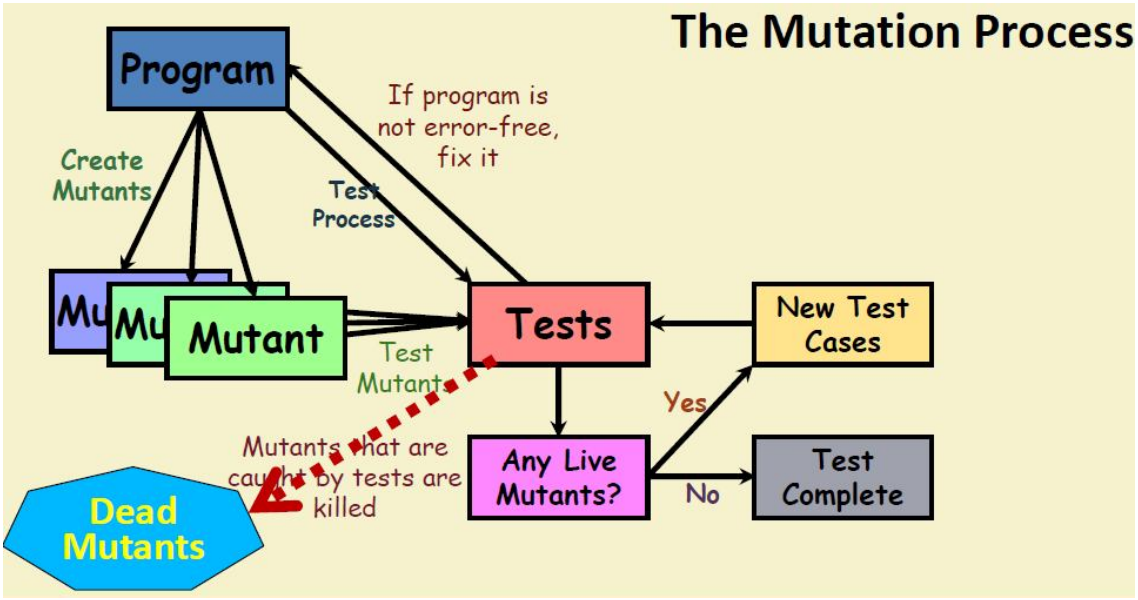
Consider the following sample program.

	Def	C-use	P-use
1. read (x, y);	x, y		
2. z = x + 2;	z	x	
3. if (z < y)			z, y
4 w = x + 1;	w	x	
else			
5. y = y + 1;	y	y	
6. print (x, y, w, z);		x, y, w, z	

6.5.3 Mutation Testing

- In this, software is first tested: Using an initial test suite designed using white box strategies we already discussed.
- After the initial testing is complete, mutation testing is taken up.
- The idea is insert faults into a program and then check whether the test suite is able to detect these.
- Each time the program is changed: It is called a mutated program and the change is called a **mutant.**
- Example primitive changes to a program are:
  - –Deleting a statement
  - –Altering an arithmetic operator,
  - –Changing the value of a constant,
  - –Changing a data type, etc.
- A mutated program is tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - –A mutant gives an incorrect result,
  - –Then the mutant is said to be dead
- If a mutant remains alive even after all test cases have been exhausted,
  - –The test suite is enhanced to kill the mutant.

- The mutation testing is shown below.



6.6. Metrics for Testing

- A few natural questions arise while testing:
  - How good is the testing that has been done?
  - What is the quality or reliability of software after testing is completed?

6.6.1 Coverage Analysis

- The coverage requirement at unit level can be 90% to 100%
- Besides the coverage of program constructs, coverage of requirements is also often examined.

6.6.2 Reliability

- As reliability of software depends considerably on the quality of testing, by assessing reliability we can also judge the quality of testing.
- Reliability estimation can be used to decide whether enough testing has been done.
- Let X be the random variable that represents the life of a system.
- Reliability of a system is the probability that the system has not failed by time t.
- In other words

$$R(t) = P(X > t).$$

6.6.3 Defect Removal Efficiency

- Defect Removal Efficiency is the ability of the system to remove defects prior to release.
- It is calculated as ratio of defects resolved to no of defects found.

*****

UNIT –VI

Assignment-Cum-Tutorial Questions

Objective Questions

1. White-box testing, sometimes called _____.
2. The testing technique that requires preparing test cases to exercise the internal logic of a software module is [ ]

a) Behavioural Software Testing b) Black-box Testing
c) Grey-box Testing d) White-box Testing

3. White-box testing uses the _____ structure of the procedural design to derive test cases. [ ]

a) Behaviour b) Control
c) Ariel d) None of the mentioned

4. Which one of the following testing techniques is effective in testing whether a developed software meets its non-functional requirements? [ ]

a) Path testing b) Dataflow testing
c) Robust boundary-value testing d) Performance testing

5. Which one of the following is a fault-based testing technique? [ ]

a) Pair wise testing b) Dataflow testing
c) Path testing d) Mutation testing

6. Suppose a certain function takes 5 Integer input parameters. The valid values for each parameter takes an integer value in the range 1..100. What is the minimum number of test cases required for robust boundary value testing? [ ]

a) 20 b) 21 c) 30 d) 31

7. Scenario coverage testing can be considered to be which one of the following types of testing strategies? [ ]

a) Pair-wise testing
b) Decision table-based testing
c) Equivalence partitioning-based testing
d) Boundary value-based testing

8. Which one of the following types of bugs may not get detected in black-box testing, but are very likely to be get detected by white-box testing? [ ]

a) Syntax errors b) Behavioral errors
c) Logic errors d) Performance errors

9. Cause-effect test cases are, in effect, designed using which one of the following types of testing techniques? [ ]

a) Decision-table based testing b) Coverage-based testing

c) Fault-based testing d) Path-based testing

10. If a user interface has three checkboxes, at least how many test cases are required to achieve pair-wise coverage? [ ]
a) 3 b) 4 c) 5 d) 6

11. Among the following test techniques, which one of the following is the strogest? [ ]
a) All path coverage testing b) Decision coverage testing
c) Basic condition coverage testing d) MC/DC testing

SECTION-B

Descriptive Questions

1) Explain different programming principles and guidelines on publicly available standards.
2) (a) Differentiate Error, Fault and Failure.
(b) What is Test Case and Test Criteria?
3) Explain Cause-Effect Graph technique with decision table.
4) Explain about Mutation Testing and write the steps for mutation testing.
5) Analyze boundary value Analysis with formulas.
6) Apply state based testing for any example and draw state model and state table.
7) Identify def, C-use and P-use in data-flow based testing and draw data-flow graph for any example.
8) A program takes an angle as input within the range [0,360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.
9) What would be the Cyclomatic complexity of the following program?

```
int find-maximum(int i, int j, int k){
int max;
if(i>j) then
if(i>k) then max=I;
else max=k;
else if(j>k) max=j
else max=k;
return(max);
}
```

10) Sketch Reliability Model for failure intensity and also with respect to time.

C) Previous GATE Questions:

1) The following is the comment written for a C function.

```
/* This function computes the roots of a quadratic equation
        a.x^2 + b.x + c = . The function stores two real roots
        in *root1 and *root2 and returns the status of validity
```

of roots. It handles four different kinds of cases.
(i) When coefficient a is zero irrespective of discriminant
(ii) When discreminant is positive
(iii) When discriminant is zero
(iv) When discriminant is negative.
Only in case (ii) and (iii) the stored roots are valid.
Otherwise 0 is stored in roots. The function returns
0 when the roots are valid and -1 otherwise.
The function also ensures root1 >= root2

```
int get_QuadRoots( float a, float b, float c,
   float *root1, float *root2);
```

*/

A software test engineer is assigned the job of doing black box testing. He comes up with the following test cases, many of which are redundant.

Test Case	Input Set			Expected Output Set		
	a	b	c	root1	root2	Return Value
T1	0	0	7	0	0	-1
T2	0	1	3	0	0	-1
T3	1	2	1	-1	-1	0
T4	4	-12	9	1.5	1.5	0
T5	1	-2	-3	3	-1	0
T6	1	1	4	0	0	-1

Which one of the following option provide the set of non-redundant tests using equivalence class partitioning approach from input perspective for black box testing?

A) T1, T2, T3, T6 **(GATE 2011)**
B) T1, T3, T4, T5
C) T2, T4, T5, T6
D) T2, T3, T4, T5

2) The following program is to be tested for statement coverage:

```
begin
  if (a== b) {S1; exit;}
  else if (c== d) {S2;]
       else {S3; exit;}
  S4;
end
```

The test cases T1, T2, T3 and T4 given below are expressed in terms of the properties satisfied by the values of variables a, b, c and d. The exact values are not given. T1 : a, b, c and d are all equal T2 : a, b, c and d are all

distinct T3 : a = b and c != d T4 : a != b and c = d Which of the test suites given below ensures coverage of statements S1, S2, S3 and S4?

A) T1, T2, T3 (**GATE 2010)**

B) T2, T4

C) T3, T4

D) T1, T2, T4

3) Match the following:

List-I	List-II
a. Condition coverage	1. Black-box testing
b. Equivalence class partitioning	2. System testing
c. Volume testing	3. White-box testing
d. Alpha testing	4. Performance testing

A) a - 2 b - 3 c - 1 d - 4 (**GATE 2015)**

B) a - 3 b - 4 c - 2 d - 1

C) a - 3 b - 1 c - 4 d - 2

D) a - 3 b - 1 c - 2 d – 4