UNIT 2- Types Of Testing

Testing Category	Techniques
Dynamic testing:	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision
Black-Box	Table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing:	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
White-Box	
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation
	Testing.
Regression testing	Selective retest technique, Test prioritization.

Black-box testing:

Black-box technique is one of the major technique in dynamic testing for designing effective test cases.

This techniques considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered.

Therefore, this is also known as functional testing.

The system is considered as a black box, taking no notice of its internal structure; so it is called black-box testing.
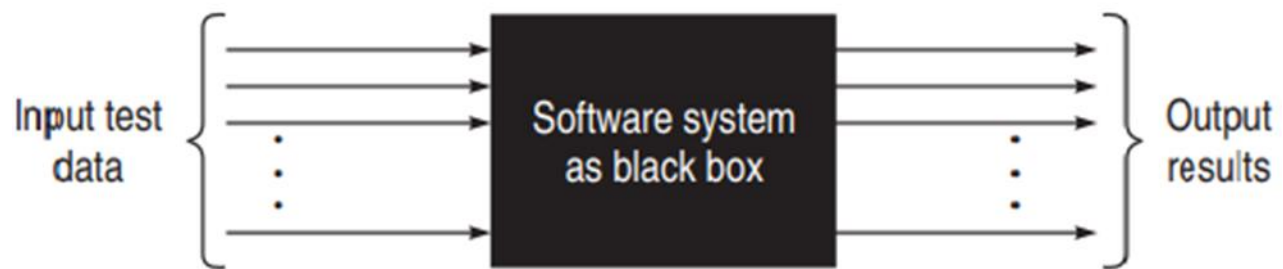
Figure 4.1   Black-box testing

- Black-box testing attempts to find errors in the following categories:

1. To test modules independently.

2. To test functional validity of the software so that incorrect or missing functions can be recognized.

3. To look for interface errors.

4. To test the system behaviour and check its performance.

5. To test the maximum load or stress on the system.

6. To test the software such that the user/customer accepts the system within defined acceptable limits.

[1] Boundary value analysis [BVA]

- An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

- BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain.

For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9, 10, 11) and on 255(256,255,254).

Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9, 10, 11) and 100(99,100,101), as shown in Fig. 4.2.
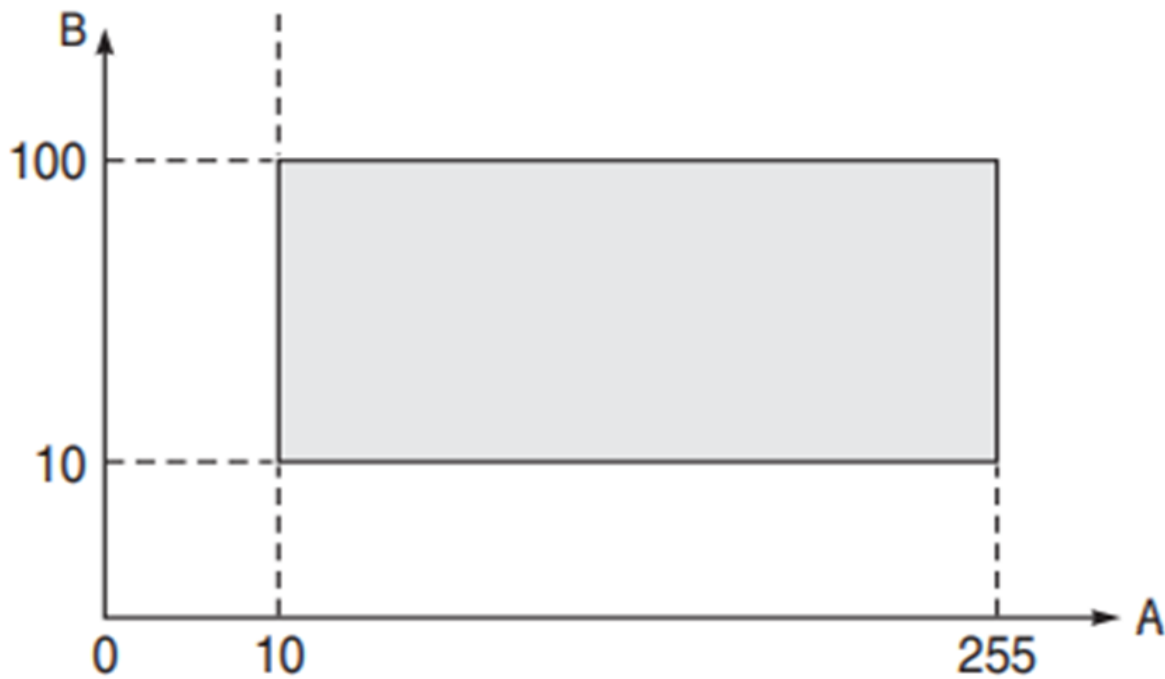

Figure 4.2 Boundary value analysis

An exam has a pass boundary at 50 percent, merit at 75 percent and distinction at 85 percent. The Valid Boundary values for this scenario will be as follows:

49, 50 - for pass

74, 75 - for merit

84, 85 - for distinction

The Invalid Boundary Cases for the above example can be given as follows:

0 - for lower limit boundary value

101 - for upper limit boundary value

4.1.1 BOUNDARY VALUE CHECKING (BVC)

- In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

- The variable at its extreme value can be selected at:

(a) Minimum value (Min)

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

(c) Maximum value (Max)

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

EXAMPLE: BOUNDARY VALUE CHECKING (BVC)

- Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following (4n+1) test cases (see Fig. 4.3) can be designed:

1. A_{nom}, B_{min}

2. A_{nom}, B_{min+}

3. A_{nom}, B_{max}

4. A_{nom}, B_{max-}

5. A_{min}, B_{nom}

6. A_{min+}, B_{nom}

7. A_{max}, B_{nom}

8. A_{max-}, B_{nom}
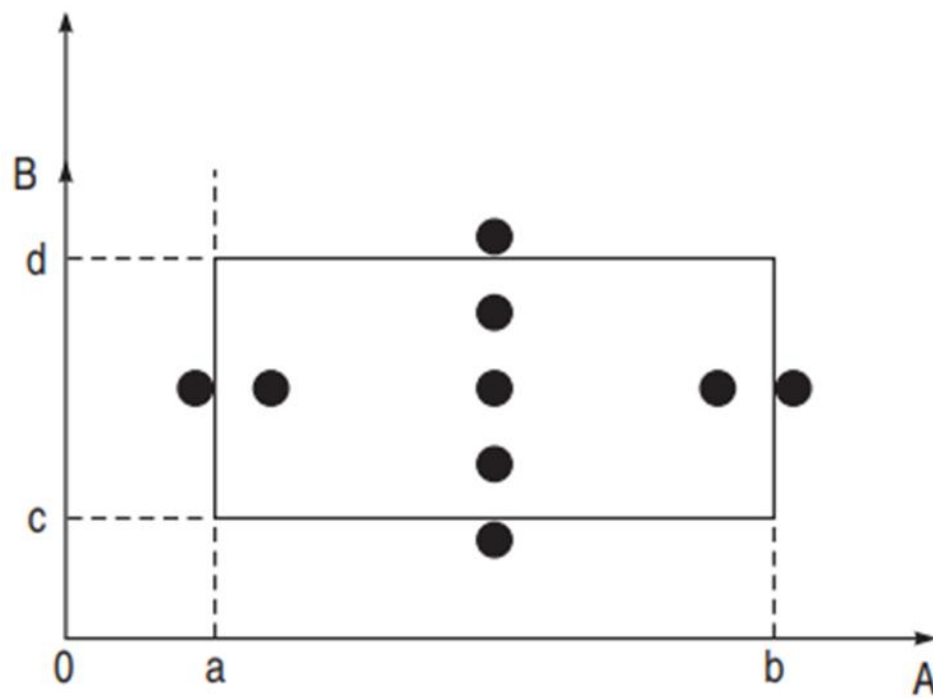
9. A_{nom}, B_{nom}

Figure 4.3 Boundary value checking

4.1.2 ROBUSTNESS TESTING METHOD

- The idea of BVC can be extended such that boundary values are exceeded as:

1. A value just greater than the Maximum value (Max+)

2. A value just less than Minimum value (Min−)

- When test cases are designed considering the above points in addition to BVC, it is called robustness testing.

- Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

- It can be generalized that for n input variables in a module, 6n + 1 test cases can be designed with robustness testing.

10. A_{max+}, B_{nom} 11. A_{min-}, B_{nom}

12. A_{nom}, B_{max+} 13. A_{nom}, B_{min-}

4.1.3 WORST-CASE TESTING METHOD

- We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called worst-case testing method. Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

- It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

- BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

10. A_{min}, B_{min}

11. A_{min+}, B_{min}

12. A_{min}, B_{min+}

13. A_{min+}, B_{min+}

14. A_{max}, B_{min}

15. A_{max-}, B_{min}

16. A_{max}, B_{min+}

17. A_{max-}, B_{min+}

18. A_{min}, B_{max}

19. A_{min+}, B_{max}

20. A_{min}, B_{max-}

21. A_{min+}, B_{max-}

22. A_{max}, B_{max}

23. A_{max-}, B_{max}

24. A_{max}, B_{max-}

25. A_{max-}, B_{max-}

4.1.4 ROBUST WORST-CASE TESTING METHOD:

In Robust Worst-Case Testing Method, we consider the all possible combinations. Total 7 values are there for one variable. Therefore, the total number of cases can be generated in Robust Worst-Case Methods are 7n. In the case of 2 variables, total number of test cases are 49.

Example 4.1

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

Test cases using BVC Since there is one variable, the total number of test cases will be $4n + 1 = 5$.

In our example, the set of minimum and maximum values is shown below:

Min value = 1

Min+ value = 2

Max value = 100

Max– value = 99

Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Test cases using robust testing since there is one variable, the total number of test cases will be 6n + 1 = 7. The set of boundary values is shown below:

Min value = 1

Min– value = 0

Min+ value = 2

Max value = 100

Max− value = 99

Max+ value = 101

Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c) Test cases using worst-case testing since there is one variable, the total number of test cases will be 5n = 5. Therefore, the number of test cases will be same as BVC.

4.2 EQUIVALENCE CLASS TESTING

- It is a black-box testing technique used in software testing as a major step in the Software development life cycle (SDLC).

- The input domain for testing is too large to test every input. So, we can divide or partition the input domain based on a common feature or a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur.

- Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors.

- If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class.

- This testing technique is better than many of the testing techniques like boundary value analysis, worst case testing, robust case testing and many more in terms of time consumption. Since testing is done to identify possible risks, equivalence class testing performs better than the other techniques as the test cases generated using it are logically identified with partitions in between to create different input and output class

- Equivalence partitioning method for designing test cases has the following goals:

1. **Completeness** Without executing all the test cases, we strive to touch the completeness of testing domain.

2. **Non-redundancy** When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug.

- To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes

2. Design test cases

- Two types of classes can always be identified as discussed below: Valid equivalence classes:-These classes consider valid inputs to the program.

- Invalid equivalence classes:-One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behavior of the program, as shown in Fig. 4.4.

- There are no well-defined rules for identifying equivalence classes, as it is a examining process. However, some guidelines are defined for forming equivalence classes:
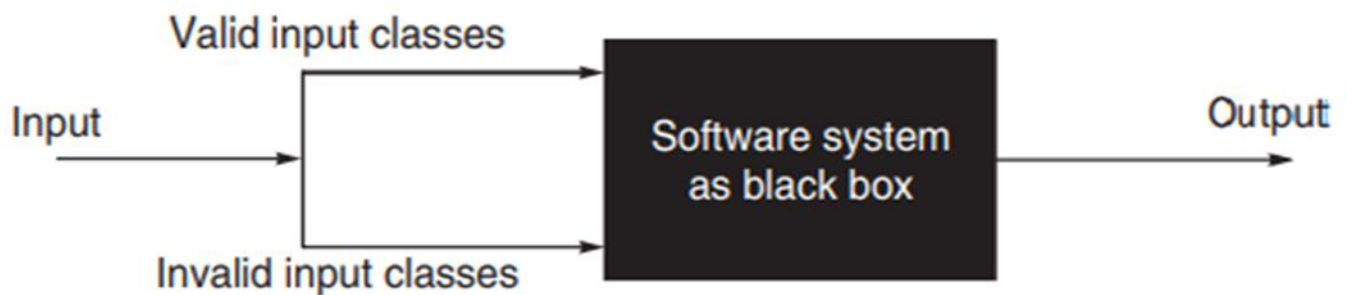

Figure 4.4 Equivalence classes

4.2.1 IDENTIFICATION OF EQUIVALENT CLASSES

- If there is no reason to believe that the entire range of an input will be treated in the same manner, then the range should be split into two or more equivalence classes.

- If a program handles each valid input differently, then define one valid equivalence class per valid input.

- Boundary value analysis can help in identifying the classes.

- If an input variable can identify more than one category, then for each category, we can make equivalent classes.

- If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are very few inputs, and one invalid class where there are too many inputs.

- If an input condition specifies a 'must be' situation, identify a valid equivalence class and an invalid equivalence class.

- Equivalence classes can be of the output desired in the program.

4.2.2 IDENTIFYING THE TEST CASES

- A few guidelines are given below to identify test cases through generated equivalence classes:

1. Assign a unique identification number to each equivalence class.

2. Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.

3. Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases.

Example-1:
Let us consider an example of any college admission process. There is a college that gives admissions to students based upon their percentage.

Consider percentage field that will accept percentage only between 50 to 90 %, more and even less than not be accepted, and application will redirect user to an error page. If percentage entered by user is less than 50 %or more than 90 %, that equivalence partitioning method will show an invalid percentage. If percentage entered is between 50 to 90 %, then equivalence partitioning method will show valid percentage.

Percentage [   ] *Accepts Percentage value between 50 to 90

Equivalence Partitioning		
Invalid	Valid	Invalid
<=50	50-90	>=90

Example: 2

Enter OTP [   ] *Must include six digits

Equivalence Partioning			
Invalid	Invalid	Valid	Valid
Digits>=7	Digits<=5	Digits=6	Digits=6
67545678	9754	654757	213309

4.4 DECISION TABLE-BASED TESTING

- Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions.

- Decision table is another useful method to represent the information in a tabular method. It has the specialty to consider complex combinations of input conditions and resulting actions. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

4.4.1 FORMATION OF DECISION TABLE

- **Condition stub**: It is a list of input conditions for which the complex combination is made.

- **Action stub**: It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

- **Condition entry**: It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE for all input conditions for a particular combination, then it is called a Rule.

  When the condition entry takes only two values—TRUE or FALSE, then it is called Limited Entry Decision Table. When the condition entry takes several values, then it is called Extended Entry Decision Table. In limited entry decision table, condition entry, which has no effect whether it is True or False, is called a Don't-Care state or immaterial state (represented by I). The state of a don't-care condition does not affect the resulting action.

- **Action entry:** It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

  Action Entry:

  - List all actions that can be associated with a specific procedure (or module).
  - List all conditions (or decision made) during execution of the procedure.

- o Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- o Define rules by indicating what action occurs for a set of conditions.

4.4.1 FORMATION OF DECISION TABLE

- The guidelines to develop a decision table for a problem are discussed below:

- Action Entry:

1. List all actions that can be associated with a specific procedure (or module).
2. List all conditions (or decision made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what action occurs for a set of conditions.

4.4.2 TEST CASE DESIGN USING DECISION TABLE

For designing test cases from a decision table, following interpretations should be done:

„ Interpret condition stubs as the inputs for the test case.

„ Interpret action stubs as the expected output for the test case.

„ Rule, which is the combination of input conditions, becomes the test case itself.

„ If there are k rules over n binary conditions, there are at least k test cases and at the most 2n test cases

Table 4.3 Decision table structure

ENTRY

		Rule 1	Rule 2	Rule 3	Rule 4	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Example 1: How to make Decision Base Table for Login Screen

Let's create a decision table for a login screen.

The condition is simple if the user provides the correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

- **T** – Correct username/password

- **F** – Wrong username/password

- **E** – Error message is displayed

- **H** – Home screen is displayed

Interpretation:

- **Case 1** – Username and password both were wrong. The user is shown an error message.

- **Case 2** – Username was correct, but the password was wrong. The user is shown an error message.

- **Case 3** – Username was wrong, but the password was correct. The user is shown an error message.

- **Case 4** – Username and password both were correct, and the user navigated to the homepage

Example:2

A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution

The decision table for the program is shown below:

ENTRY

		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

White-Box Testing:

White-box testing is another effective testing technique in dynamic testing. It is also known as glass-box testing, as everything that is required to implement the software is visible.

The entire design, structure, and code of the software have to be studied for this type of testing.

 It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code.

This testing is also known as structural or development testing.

NEED OF WHITE-BOX TESTING:

- Is white-box testing really necessary? Can't we write the code and simply test the software using black-box testing techniques? The supporting reasons for white-box testing are given below:

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.

2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).

4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.

5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

Difference between Black-box and White-box Testing:

Black-box Testing	White-box Testing
This is also called as internal structure testing technique	It is also called as glass-box testing technique
It is a testing technique in which everything i.e. the code, internal structure of the program being tested is not known to the tester	It is a testing technique in which the code and internal structure of the program being tested must be known to the tester
It is done by the independent Software Testers	It is done by the Software Developers
The technique requires no programming knowledge	This technique requires programming knowledge

WHITE BOX TESTING TECHNIQUES

LOGIC COVERAGE CRITERIA

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. Discussed below are the basic forms of logic coverage.

[1] Statement coverage:

- The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
        if (x > y)
                x = x - y;
        else
                y = y - x;
}
printf ("x = ", x);
printf ("y = ", y);
```

Figure 5.1   Sample code

- The following Test-cases needs to be design to (test) cover all the statements of the code given in the Figure 5.1.

- Test case 1: x = y = n, where n is any number

- Test case 2: x = n, y = n', where n and n' are different numbers.

Test Case 1 just skips the while loop and all loop statements are not executed.

Considering test case 2, the loop is also executed.

However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: x > y

Test case 4: x < y

These test cases will cover every statement in the code segment; however statement coverage is poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, statement coverage is a necessary but not sufficient criteria for logic coverage.

Example:

1. intmain()
2. {
3. intA, B, C;

4. printf("Enter three numbers: ");
5. scanf("%d %d %d", &A, &B, &C);

6. if(A >= B && A >= C)
7. printf("%d is the largest number.", A);

8. elseif(B >= A && B >= C)
9. printf("%d is the largest number.", B);

10.   else

11.   printf("%d is the largest number.", C);

12.   return0;

13.   }

[2] Decision OR branch coverage:

Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in Figure 5.1, while and if statements have two outcomes: True and False. So test cases must be designed such that both outcomes for while and if statements are tested. The test cases are designed as:

Test case 1: x = y

Test case 2: x! = y

Test case 3: x < y

Test case 4: x > y

[3] Condition coverage:

- Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following example:

while ((I <=5) && (J < COUNT))

- In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

- Test case 1: **I <= 5, J < COUNT**

- Test case 2: **I < 5, J > COUNT**

- Example: If (A && B) is being tested, the condition coverage would allow one to write two test cases:

- Test case 1: A is True, B is False.

- Test case 2: A is False, B is True.

[4] Multiple condition coverage:

- In case of multiple conditions, even decision/ condition coverage fails to exercise all outcomes of all conditions.

- The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions.

- For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated.

- Similarly, if an OR condition is true, none of the subsequent conditions will be evaluated.

- Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

- Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once.

- The following test cases can be there:

Test case 1: A = True, B = True

Test case 2: A = True, B = False

Test case 3: A = False, B = True

Test case 4: A = False, B = False

5.3 BASIS PATH TESTING

- Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program.

- Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing.

- Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors.

- The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.

2. Path testing requires complete knowledge of the program's structure.

3. Path testing is closer to the developer and used by him to test his module.

4. The effectiveness of path testing gets reduced with the increase in size of software under test.

5. Choose enough paths in a program such that maximum logic coverage is achieved.

5.3.1 CONTROL FLOW GRAPH

- The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed

graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V.

1. **Node:** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.

2. **Edges or links:** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.

3. **Decision node:** A node with more than one arrow leaving it is called a decision node.

4. **Junction node:** A node with more than one arrow entering it is called a junction.

5. **Regions:** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

5.3.3 PATH TESTING TERMINOLOGY

- **Path:** A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.

- **Segment:** Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision).

- **Length of a path:** The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path.

- **Independent path:** An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.
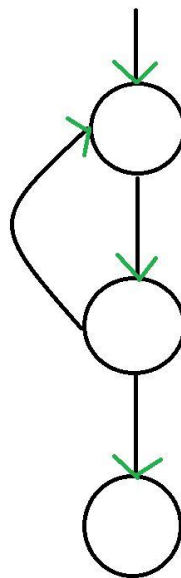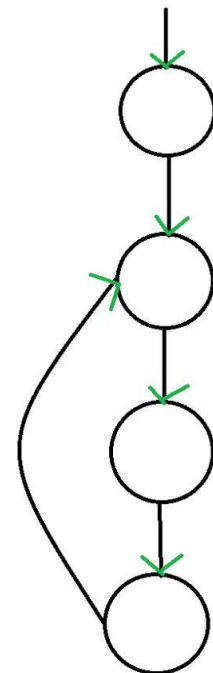
FLOW GRAPH NOTATIONS
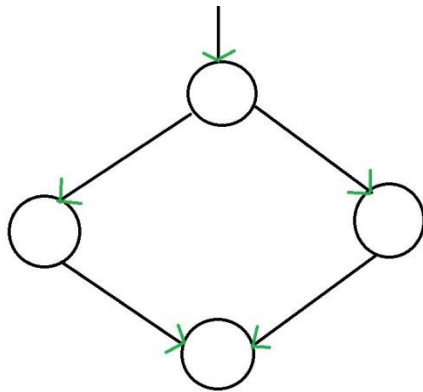
If-then-else
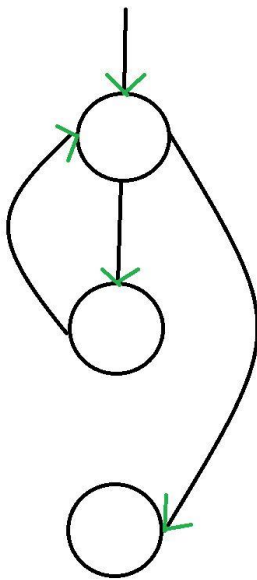
while   do-while   for

CYCLOMATIC COMPLEXITY

- In the graph shown in Figure 5.3, there are six possible paths: acei, acgh, acfj, bdei, bdgh, bdfj. In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4.
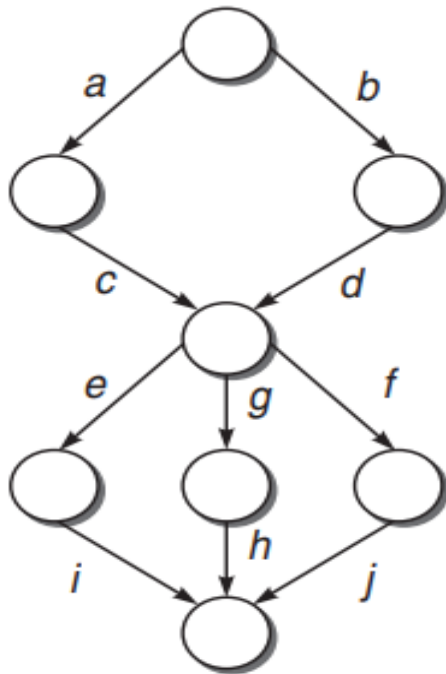
- $V(G) = e - n + 2$

Figure 5.3 Sample graph

Formulae Based on Cyclomatic Complexity

- Cyclomatic complexity number can be derived through any of the following three formulae

- 1. **V(G) = e − n + 2p** where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.

- 2. **V(G) = d + p** where d is the number of decision nodes in the graph.

- 3. **V(G) = number of regions in the graph.**

Guidelines for Basis Path Testing

- Draw the flow graph using the code provided for which we have to write test cases.

- Determine the cyclomatic complexity of the flow graph.

- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.

- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

6.STATIC TESTING (why?):

- Dynamic testing uncovers the bugs at a later stage of SDLC and hence is costly to debug.

- Dynamic testing is expensive and time-consuming, as it needs to create, run, validate, and maintain test cases.

- The efficiency of code coverage decreases with the increase in size of the system.

- Dynamic testing provides information about bugs. However, debugging is not always easy. It is difficult and time-consuming to trace a failure from a test case back to its root cause.

- Dynamic testing cannot detect all the potential bugs.
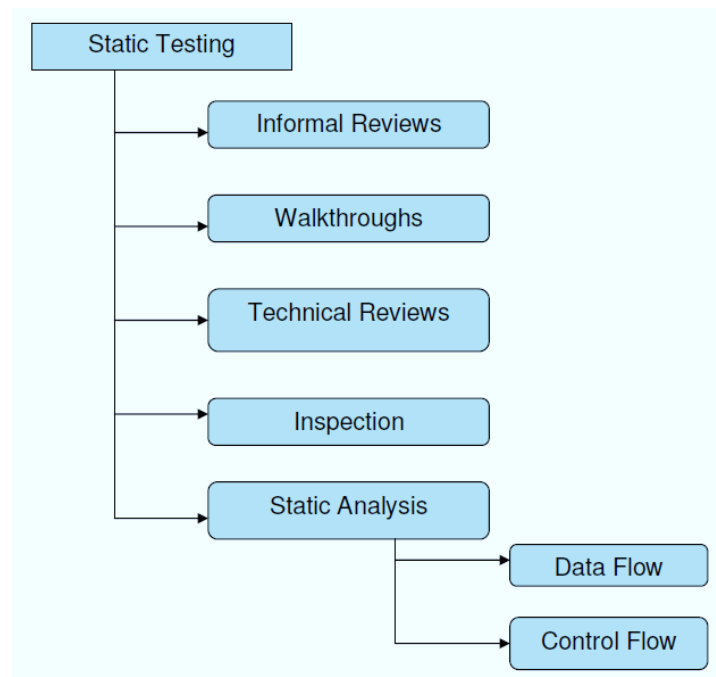
6.1 INSPECTIONS:

- Software inspections were first introduced at IBM by Fagan in the early 1970s.

- These can be used to tackle software quality problems because they allow the detection and removal of defects after each phase of the software development process.

- Inspection process is an in-process manual examination of an item to detect bug.

- It may be applied to any product or partial product of the software development process, including requirements, design and code, project management plan, SQA plan, software configuration plan (SCM plan), risk management plan, test cases, user manual, etc.

- This process does not require executable code or test cases.

- The inspection process is carried out by a group of peers.

6.1.1 INSPECTION TEAM

- **Moderator:** A team member who manages the whole inspection process. He schedules, leads, and controls the inspection session. He is the key person with the responsibility of planning and successful execution of the inspection.

- **Author/Owner/Producer:** A programmer or designer responsible for producing the program or document. He is also responsible for fixing defects discovered during the inspection process.

- **Inspector:** A peer member of the team, i.e. he is not a manager or supervisor. He is not directly related to the product under inspection and may be concerned with some other product. He finds errors, omissions, and inconsistencies in programs and documents.

- **Recorder:** One who records all the results of the inspection meeting.

6.1.2 INSPECTION PROCESS

- Inspection process has following stages:

1. Planning
2. Overview
3. Individual Preparation
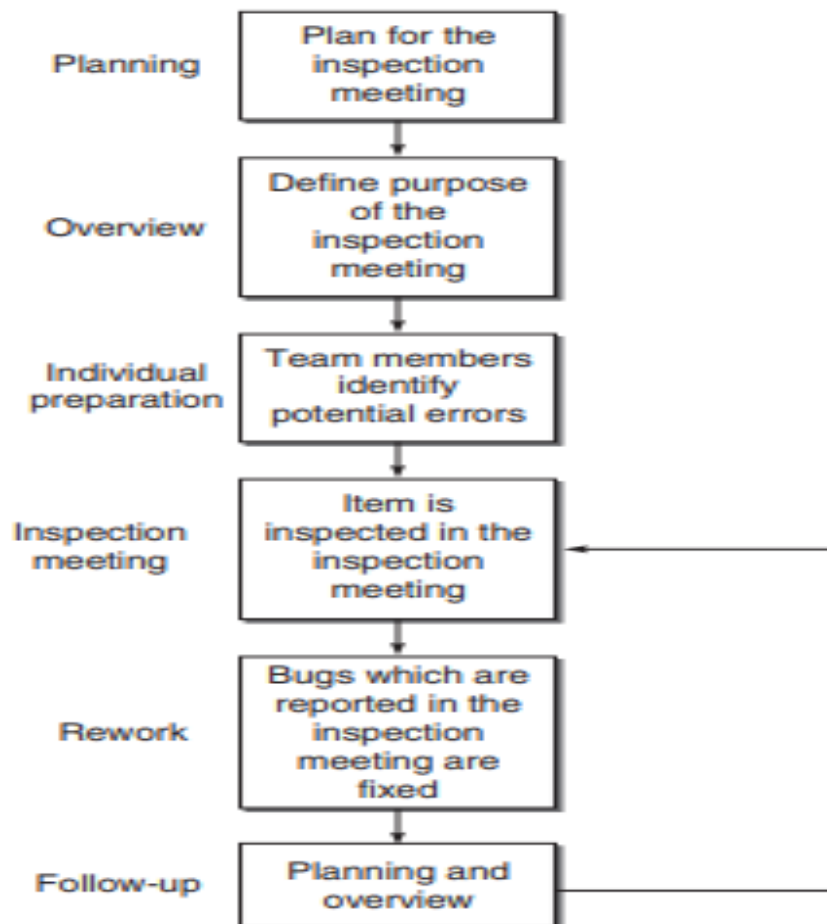4. Inspection Meeting
5. Rework
6. Follow-up

Figure 6.1 Inspection process

6.1.3 BENEFITS OF INSPECTION PROCESS

- Bug reduction
- Bug prevention
- Productivity
- Real-time feedback to software engineers
- Reduction in development resource
- Quality improvement
- Project management
- Checking coupling and cohesion
- Learning through inspection
- Process improvement

6.2 STRUCTURED WALKTHROUGHS:

- The idea of structured walkthroughs was proposed by Yourdon. It is a less formal and less rigorous technique as compared to inspection. The common term used for static testing is inspection but it is a very formal process.
- If you want to go for a less formal process having no bars of organized meeting, then walkthroughs are a good option.
- A typical structured walkthrough team consists of the following members:
- **Coordinator:** Organizes, moderates, and follows up the walkthrough activities.
- **Presenter/Developer:** Introduces the item to be inspected. This member is optional.
- **Scribe/Recorder:** Notes down the defects found and suggestion proposed by the members.
- **Reviewer/Tester:** Finds the defects in the item.

- **Maintenance Oracle:** Focuses on long-term implications and future maintenance of the project.

- **Standards Bearer:** Assesses adherence to standards.

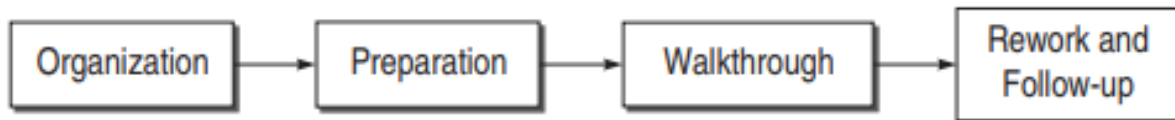- **User Representative/Accreditation:** Agent Reflects the needs and concerns of the user.

Organization → Preparation → Walkthrough → Rework and Follow-up

Figure 6.7   Walkthrough process

6.3 TECHNICAL REVIEWS:

- A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives.

- A review is similar to an inspection or walkthrough, except that the review team also includes management.

- Therefore, it is considered a higher-level technique as compared to inspection or Walkthrough.

- A technical review team is generally comprised of management-level representatives and project management.

- Review agendas should focus less on technical issues and more on oversight than an inspection.

- The purpose is to evaluate the system relative to specifi cations and standards, recording defects and deficiencies.

- The moderator should also prepare a set of indicators to measure the following points:

1. Appropriateness of the problem definition and requirements

2. Adequacy of all underlying assumptions

3. Adherence to standards

4. Consistency

5. Completeness

6. Documentation

- The moderator may also prepare a checklist to help the team focus on the key points.

Assignment:2   Submission Date:21/02/2023

1. What is black box testing? List its different techniques. Explain any one in detail.

2. List methods of BVA and explain any two in detail.

3. How equivalence class testing is used in black box testing? Explain it in detail.

4. Explain decision table in detail.

5. What is white box testing? Why we need it?

6. Explain Logic coverage criteria with its various forms.

7. Write a short note on Basis Path Testing.

8. Explain Inspection process. Explain members of the inspection team with their role.

9. Explain structured walkthrough and technical review in detail.

MCQs:

4	4.1 to 4.11 (All)
5	5.1 to 5.9
6	6.1 to 6.9 , 6.19

Unit-2