Unit 4 – Chapter 11

Testing

Prerequisites:

Software development life cycle, verification, and validation, testing methods

Unit Outcomes:

The objective of this chapter is to introduce the concept of software testing. At the end of this chapter, students will be able to:

- Design the test stubs and drivers
- Discuss the test cases and suites
- Analyze the white box and black box testing
- Explain static testing strategies

11.1 Test Stubs and Test Drivers

Test stubs and drivers provide the complete environment to check the performance and the results. Test stubs and drivers are used as a short-term alternative for the original testing process. The test stubs and rivers are shown in Figure 11.1.

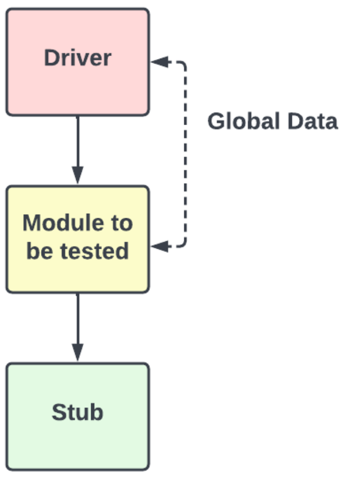

Figure 11.1 Use of Test Stubs and Drivers in Testing

A stub is a simple dummy procedure with the same input-output parameters as the original procedure. If the module is not developed or is in progress, then a test stub is used. Test stubs are developed by developers to improve testing speed, especially for lower components. Testing stubs consist of the following:

- Messages to trace the data
- Input values for components in a module
- Return value to check the values managed by the component or the module
- Specific return vale to perform the unit testing

Test stubs are mainly used in top-down integration testing. Test drivers include the code with nonlocal data structures. Different modules access Data structures and the code in the test drivers to test the functions with appropriate values. They are like stubs and contain additional information such as result estimation, communication links, and summary reports. A case study of stubs and drivers is given below.
Example: Consider a web application used for online payment where there are four modules as given below.

Module Name	Module Activity
Login()	Allows to login into the application
Order()	Performs order placing
Payment()	Allows payment with the use of the card, internet banking, UPI, cash, etc.
Report()	Generate the bill and receipt with the tracking number

Table 11.1 Module details in Online Web Application

In practice, the coding and testing of the modules can be done in a parallel fashion. Each module is developed, integrated with other modules, and tested based on the interfaces or dependencies. In the above example a user can log in to the web page, after selecting the option, in order processing he/she must go through the Payment () option, which is not yet ready and still in the developing process. The Payment () module is intern linked to the Report () and again it is not yet programmed. In such situations, stubs and drivers can be used in software testing. The Payment () module is a generic module having different payment options. These can be replaced by stubs and drivers.

In summary stubs and rivers are used in top-down and bottom-up software testing respectively. Test stubs are used as an alternative for lower small components, whereas test drivers are used for main modules. Test drivers call test stubs and use them in development testing.

11.2 Test Case Design Approaches: White Box and Black box functional testing

11.2.1 Test cases and suites

A document based on requirement specification that contains sets of test data expected output, conditions, and assumptions are known as a test case. It is used in program execution at the beginning with sample input. The test case consists of a unique ID, description of the scenario, preconditions, steps, input test data, expected output, actual output, information about environment settings, and comments. A login and password verification test case is as shown below:

Test Case#	Test Case Description	Test Data	Expected Results	Actual Results	Pass/Fail
1	Check response after valid login and password are entered	Email: vaishali@geu.ac.in Password: *******	Successful Login	Login was successful	Pass

Table 11.2 Test Case Example of Login and Password

If there is no match between actual results and expected output, then the error is reported, and the defect is fixed. Test cases are designed by the following methods.

- Structure of the components
- Using requirement specifications in black box testing that includes

- Based on the tester’s knowledge and experience with the subject

The test suite consists of multiple test cases that help in test execution and reporting test status. The status can be either active, in progress, or completed. Test suites are based on functional or non-functional.

For designing test cases, the internal structure of the software should be well known. This is done using structural testing, commonly known as **white box testing**. The term white box is used to see through the software being tested. In functional testing, only the functional specification of the software is tested, and it is known as **black-box testing**. The term black box indicates that the focus is on the functionality and other details are ignored. Some of the non-functional requirements such as performance, usability, scalability, etc are also performed in black box testing. A detailed discussion of both approaches is given below.

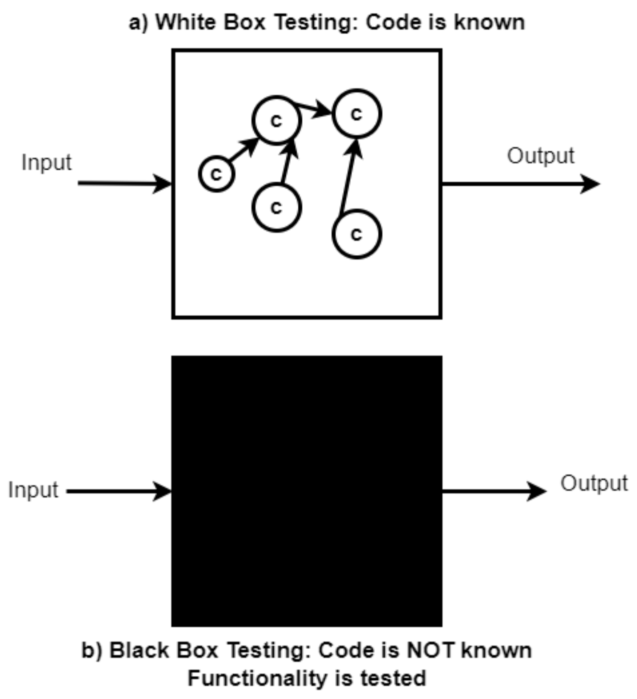
a) White Box Testing: Code is known
Input
Output
c
Input
Output
b) Black Box Testing: Code is NOT known
Functionality is tested

Figure 11.2 White Box and Black Box Testing

11.2.3 White Box Testing

This is performed by the software developer and not the test engineer. The structure, design, and coding of the developed software are tested. The white box testing includes verifying expected output for specific inputs, unstructured control and looping statements, security loopholes, objects and classes, data, control flow, etc. The different tests carried out in the white box testing are shown in Figure 11.3.

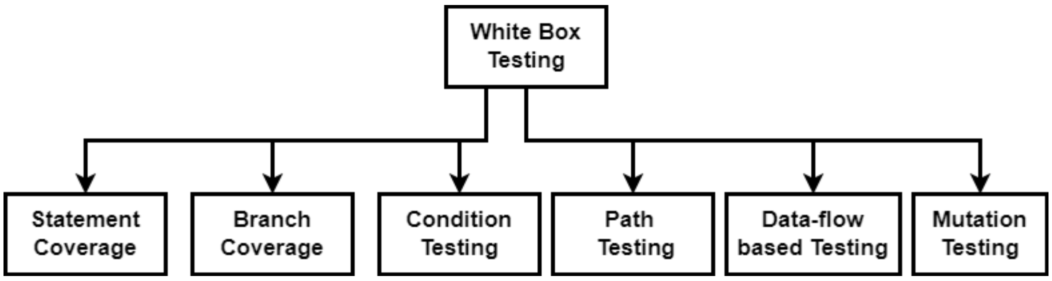

Figure 11.3 White Box Testing

1) **Statement Coverage Testing:** The testing process aims at designing test cases so that every statement in a program is executed at least once. The main idea is to check whether there is a possibility of a software failure due to some mistakes in computation, memory allocation, wrong arithmetic calculations, etc. Whether the developed software is working for all the possible test cases is checked. For example, in the program given in Table 11.3, for testing all the statements we require inputs as: {1,2,3}, {1,3,2} {3,2,1}.

```
int main()
{  double a, b, c;
  printf("Enter three different numbers: ");
  scanf("%lf %lf %lf", &a, &b, &c);
  if (a >= b && a >= c)
  printf("%.2f is the largest number.", a);
  if (b >= a && b >= c)
   printf("%.2f is the largest number.", b);
  if (c >= a&& c >= b)
    printf("%.2f is the largest number.", c);
  return 0;
}
```

Table 11.3 Statement Coverage in white box testing

2) **Branch Coverage Testing:** Branch testing is also known as edge testing where each conditional branching is tested for true and false values. It guarantees that all the test cases are fulfilled, and it is stronger than the statement coverage. Consider the program given in Table 11.4, here the test cases for checking the branch coverage can be be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

```
#include <stdio.h>
int gcd_calculate(int x, int y)
{
if (y == 0) {
return x;
} else if (x >= y && y > 0) {
return gcd_algorithm(y, (x % y));
}
```

Table 11.4 Branch testing

3) **Condition Testing:** When there is more than one condition or composite condition, with different branch statements, then a condition testing method is used. For example, consider a statement ((X OR B) NOT C). Here the test cases with all the true and false values must be used. In this structural testing, test cases are designed to make each component of composite conditions is tested. If there are n expressions in a composite conditional statement, then, 2^n test cases are required. The number of test cases increases exponentially in condition testing. Thus, condition testing is used when several conditions are small. It is stronger than the statement coverage and branch testing.

4) **Path Testing:** Path testing uses test cases so that the paths in a program are tested. The paths may be linearly independent. A linearly independent path can be defined using the control flow graph (CFG).

   CFG: The CFG presents the sequence of execution of different program statements. The statements are numbered, and each numbered statement serves as a node of CFG. Consider two statements 1 and 2. If the execution of one statement results in the transfer of control to another statement, then the two nodes are connected by an edge. The different operations in a program such as a sequence, selection, and loops or iterations can be shown in Figure 11.4.

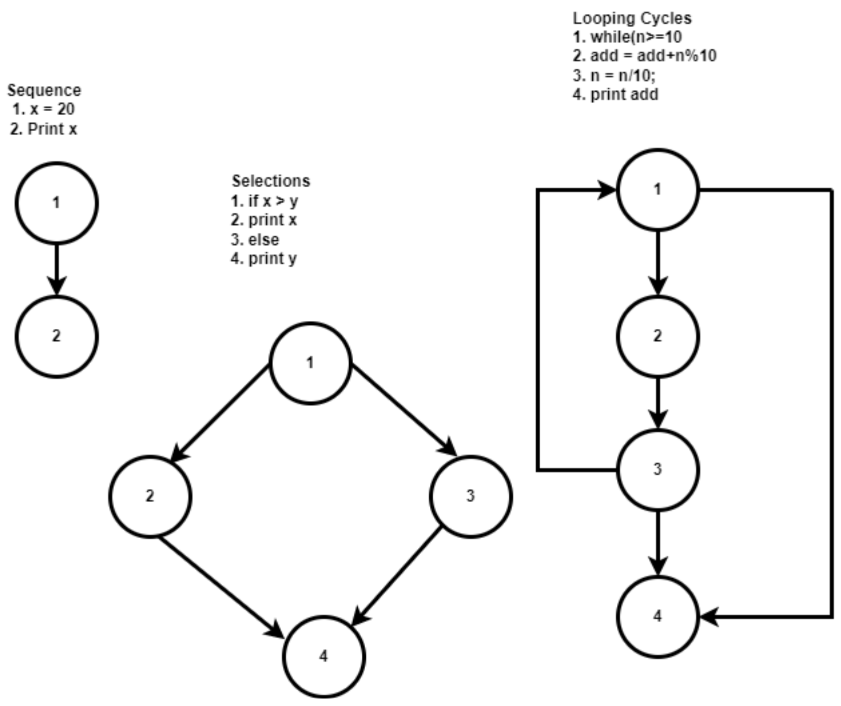

Figure 11.4 Use of CFG for sequence, selections, and Loops

In a CFG, a path is considered as a sequence of nodes and edges from a starting node to the ending node. It is not practical to write test cases that can cover all the paths. Therefore, only linearly independent paths must be covered. A path that is leading to at least one new edge is known as a linearly independent path. A linearly independent path automatically implies that there is a new test that needs to be followed.

Cyclomatic Complexity: It is complicated to find the number of independent paths of a program. The maximum number of linearly independent paths throughout the program can be identified using McCabe's cyclomatic complexity. It may not give the exact number of paths, but approximately it determines how many paths are to be tested. There are different methods to compute the cyclomatic complexity.

1. Method 1: Consider a CFG. The cyclomatic complexity C(G) can be computed as
   C(G) = E – V + 2
   where V is the number of nodes of the CFG, and E is the number of edges in the CFG.
2. Method 2: The cyclomatic complexity of a program can be computed by inspecting the CFG as follows:
   C(G) = Total number of bounded areas + 1

A bounded area in a CFG is any region enclosed by nodes and edges. The bounded area contains many decisions sequence paths and iterations, which make the testing measurable and reliable. In such situations, it is easy to determine McCabe's cyclomatic complexity. But, if the graph is having two or more intersecting edges, then McCabe's cyclomatic complexity cannot be used.

Usually, for structured programs, there are no intersecting edges for graphs (these graphs are known as planar graphs). If there are unconditional branching statements like GOTO, the CFG is with intersecting edges. Thus, GOTOs must be avoided and for such non-structured programs, McCabe's cyclomatic complexity method cannot be used. CFGs can be just viewed to determine the cyclomatic complexity. Also, it can be programmed and used to determine the cyclomatic complexities of arbitrary CFGs.

3. Method 3: The number of decision statements of a program can be used to determine the cyclomatic complexity of a program. Let us consider a program P with the number of decision statements of the program as N, then McCabe's metric C(G) = N+1.

   The details of each method are given by the example of computing the GCD of two numbers. The function to calculate the GCD of two numbers and the corresponding CFG is given in Figure 11.5.

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

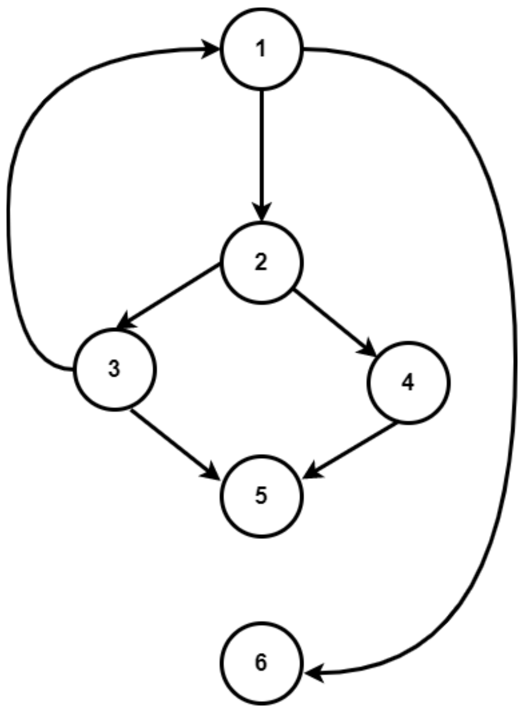

Figure 11.5 CFG for a function to calculate GCD of two numbers

The cyclomatic complexity calculation for all the three methods is given as follows:

Method 1: C(G) = E – N + 2

= 7 – 6 + 2

= 3

Method 2: C(G) = Total number of bounded areas + 1

= 2 + 1

= 3

Method 3: C(G) = N + 1

= 2 + 1

= 3

5) **Data flow-based testing:** The definitions and uses of different variables n a program can be used as test paths. Data flow-based testing methods make use of such paths. The issues such as undeclared variables, deleting the memory of a variable before its use, or multiple declarations of variables are pointed out in data flow testing.

Consider a statement with the number M, then

DEF(M) = {A/statement M contains a definition of A}, and

USES(M) = {A/statement M contains a use of A}

Consider the statement A = B + C.

The DEF(M) = {A} and USES(M) = {B, C}.

Here is the definition of A at statement M. Suppose there is a path from statement X to statement Y and statement Y does not contain a definition of A, then even statement Y can use A as if it is live. This is known as a definition-use chain (DU chain). The DU chain can be given as follows: A is the variable, and X and Y are the statements. X contains

[A, X, Y]. A $\in$ DEF(X) and A $\in$ USES(Y)

This indicates that the definition of A is live in X and Y statements. In data flow testing, every DU chain is checked whether it is covered at least once.

6) **Mutation Testing:** In this testing, some statements of the source program are changed, also known as mutated to verify whether the test cases can locate errors. The changes in the program are very small and random. The changed program is called a mutant. The objective is to build the test cases robust so that they fail the mutated source program. This testing is used in unit testing to create a fault on purpose and perform white box testing. If the output of the mutant is different than the original, then it is killed. The different faults to be introduced may be the change of data type,

altering the constant value, changing the order of the operators in an expression, etc. The pictorial presentation of mutation testing is shown in Figure 11.6.

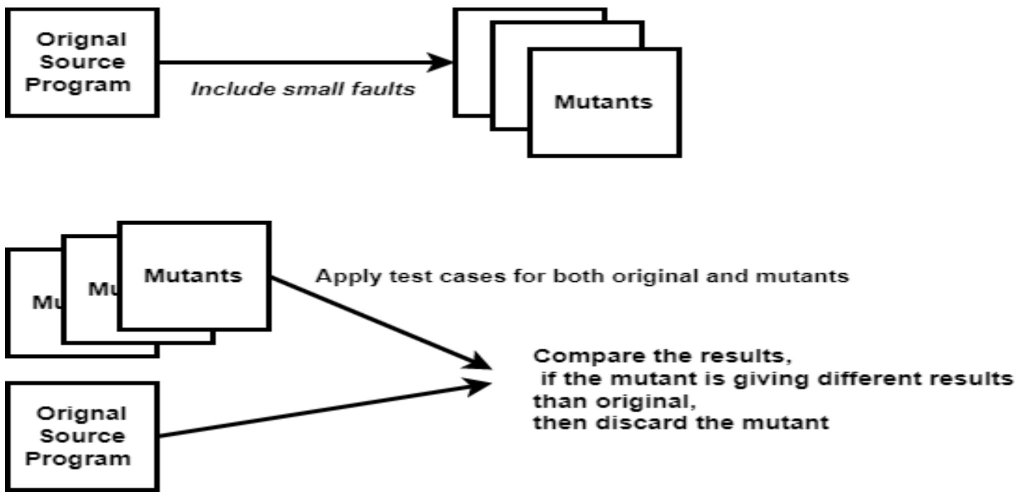
Orignal Source Program
Include small faults
Mutants
Mu
Mu
Mutants
Apply test cases for both original and mutants
Compare the results, if the mutant is giving different results than original, then discard the mutant
Orignal Source Program

Figure 11.6 Mutation Testing

Mutation testing is expensive as there is a need to generate multiple mutants and it is time-consuming.

11.2.4 Black Box Testing

Black box testing involves testing functionalities of the software using requirement specifications and without the knowledge of design, or code. Black box testing is also known as behavioral testing. The black box testing can be performed by the following steps.

- Examine the requirement specifications of the system.
- Select a set of proper inputs and determine the possible output
- Prepare test cases for the inputs and execute the test cases
- Compare the results with expected outcomes.
- If there are any undesired results or errors, then correct the errors and repeat the testing.

As the black box testing is related to verifying the functionality of the system, it concentrates on preparing test cases. The two approaches to designing black-box test cases are equivalence class portioning and boundary value analysis.

a) Equivalence Class Partitioning

The equivalence class partitioning method is used to reduce the building of possible test cases. The set of input values is divided into different classes. The test cases are built in such a way that testing one class is like testing other classes.

For example, consider a program that computes the square root of numbers in the range 0 to 100. Then the test cases can be divided into three categories as follows:

1) Integers less than 0
2) Integers equal to 9
3) Integers between 1 and 100
4) Integers greater than 100

The test cases can be designed so that the sample data can be representative of each class. Now {-2,0,3,118} can be a equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. Generally, one valid and two invalid equivalence test classes are designed. In case there is a range of input values then one equivalence class for valid inputs and another for invalid inputs is defined.

Consider a range for age groups, here the different inputs can be tested against the equivalent partitions.

Age valid	Age Invalid	Age suitable for Voting	Age not suitable for voting
1 to 110	0 and <0	Above 18	Below 18

Table 11.5 Equivalence Class Partitioning

b) **Boundary Value Analysis**: Here the software is tested for the input boundaries. The general practice is to test the value from the middle value in the input range. The programmers often miss the less than or greater than levels, so this technique is useful in such cases.

Example: In an online banking application, more than 10 fund transfers are not allowed. While testing this application, you may start with middle values such as 5, 8 and test up to 10. When the value 10 is crossed the system should generate the exception error message.

Static Testing Strategies

Static testing strategies are used to check the system without executing a code. It is an economical process, and it is done at the early stages of software development. Software is verified either manually or using automated tools. Doing static testing helps reduce the development cost and time. The defects can be identified at the early stages, and it requires less effort in fixing the bugs. The missing requirements, inconsistencies in the interface design, and deviation from the target problems are identified in the static testing.

Static testing strategies are classified into the following two types: review and static analysis. Some of the review and inspection techniques are already discussed in Chapter 9 in unit 3 of design and development.

11.4.1 Review

In the review process, the code is reviewed using software requirement specifications and other software design-related documents. The review is conducted by different people to identify any errors, duplicate or confusing statements if any. There are four different types of review.

a) Informal: Informal meeting where the developer presents the code in front of the team and the defects are identified by everyone at the early stage of development.
b) Walk-through: A subject expert is assigned to go through the code to avoid any errors in the future development stage.
c) Peer Review: Team members test and verify each other's documents to identify and correct the defects. There are four categories of participants in the peer review.
d) Inspection: In the inspection process, the SRS is used to verify the code by the higher authorities

11.4.2 Static Analysis

In the static analysis strategy, the quality of the developed code is evaluated using different standard tools. The defects such as syntax errors ignored in compilation, variables that are not serving any use, unused or unwanted code, infinite loops, and variables without initialization are identified in this method. Static analysis is done using three types. Already these techniques are discussed in white box testing.

1) Data flow for checking the processing of data streams at various stages.
2) Control flow of different statements during execution
3) Identification of linearly independent paths using cyclomatic complexity.

11.5 Compliance with Design and Coding Standards

In the development phase, different modules are coded as per the requirement specifications and design. The main objective of the coding phase is to transform the design into a program and perform testing. The quality and reliable software development firm always expects the developed code to be well-defined and with a standard coding style. To pass through the code review, the coding standards must be maintained.

As there is more than one member in the team, following a standard coding style gives a uniform appearance. The other non-functional requirements such as maintainability and readability of the code can be improved by using standard coding styles. In general, no coding standard is perfect for any application. The general coding standards are related to the following.

- Structure of a program
- Naming rules and conventions
- Formatting
- Code Documentation
- Use of features in a programming language

The coding guidelines and documentation are already discussed in sections 9.1 and 9.2 in chapter 9 of unit 3.
The standards related to the different aspects of software development are used in requirement gathering, design, coding, verification and validation, documentation, the interaction between computer and human, quality control, and ethics.

Standards are written by well-known national and international standards bodies. There are about 80 international and national standards. Some familiar names that create standards are given as follows.

1.	ISO	International Organization for Standardization
2.	BSI	British Standards Institute
3.	ANSI	American National Standards Institute
4.	IEEE	Institute for Electronic and Electrical Engineers

11.6 Automated Testing

Automated testing is used for the automatic review and validation of a software product using a test case suite. Manual testing can be slow. The quality of a software product is tested to check whether it fulfills the standard coding style, experience, and business of a user.

Automation Testing is a software testing technique that performs using special automated testing software tools to execute a test case suite. Manual testing can be time-consuming and may not cover all the test cases. Using automated testing a user need not be physically present in front of a computer. Automation can perform the tests, compare the expected and actual results, and generate detailed test reports. The drawback of automated testing is that it is not cost-effective in some cases.
The different criteria for using automated testing techniques are:

- Critical software with high-risk business
- Repeated execution of test cases
- Lengthy code with a high volume of data

Some of the examples used in automated testing are:

- Compare two images (e.g., photos, signature, captcha, etc)
- Testing a database and web-based application for more than 1000000 users (e.g., irctc website in Indian railway reservation online tickets)
- Testing a developed software on different operating system platforms concurrently.

Automated testing can be used for unit testing, integration testing, functional testing regression testing, black-box testing, etc. Testing tools are selected based on the software environment, database, objects, image, etc. The selection of testing tools depends on the type of application used. Some of the automated testing tools are:

1. LambdaTest: This tool is used for testing the web application for different web browsers, operating systems, and devices.
2. BrowserStack: This testing tool is used to test the application on the web and mobile application
3. TestProject: This tool provides a free test automation platform for web and mobile applications
4. WorkSoft: This tool is used to give a platform for test automation, documentation support, and business process.

In summary, the testing techniques can be summarized in Table 11.6.

Testing Category	**Techniques**
Black Box Testing Related to dynamic Testing.	Equivalence Class Partitioning, Boundary Value Analysis
White Box Testing Related to dynamic testing.	Statement Coverage Testing, Branch Coverage Testing, Condition Testing, Path Testing, Data flow-based Testing, Mutation Testing, Function/Performance Testing
Static Testing	Informal Review, Walkthrough, Inspection
Testing related to validation	Unit testing, System Testing, Acceptance Testing, Integration Testing
Testing during the maintenance phase	Regression Testing

Table 11.6 Summary of Software Testing

11.7 Multiple Choice Questions (MCQs)

1. A testing method with different versions with introducing fault is:
    a. Unit Testing
    b. Regression Testing
    c. Mutation Testing
    d. Beta Testing
2. The white box technique is performed after:
    a. Deployment of software
    b. Finalizing design
    c. Coding
    d. Requirement's finalization
3. Cyclomatic complexity is used in
    a. Alpha testing
    b. Grey box testing
    c. Black box testing
    d. White-box testing
4. The usability of software is tested using
    a. Black box testing
    b. White box testing
    c. Module testing
    d. Security testing
5. White box technique is also known as
    a. Structural testing
    b. Nonfunctional testing
    c. Practical testing
    d. Error detection testing
6. Boundary value analysis is used in
    a. Unit Testing
    b. Regression Testing
    c. Black box Testing
    d. Beta Testing
7. A black box tester cannot understand
    a. Faults in the system

    b. Requirement document
    c. Functional requirements
    d. Source code

8. The usability testing is followed in
    a. White Box testing
    b. Black box testing
    c. Unit testing
    d. None of the above
9. The structural testing does not belong to
    a. Acceptance testing
    b. Stress testing
    c. Parallel testing
    d. Regression testing
10. The static analysis does not belong to
    a. Code walkthrough
    b. Code inspection
    c. Analysis of data flow
    d. Error guess

11.8 REVIEW QUESTIONS

1. Describe the meaning of test stubs and test drivers with a suitable example.
2. How is path coverage testing performed using CFG?
3. Determine the cyclomatic complexity for an iterative C function to find the factorial of a number.
4. What is white box testing? explain with a suitable example.
5. What is automated testing? Explain its advantages and uses in suitable scenarios.
6. Write the advantages of black-box testing.
7. Compare white box and black box testing methods.
8. Write a note on different methods used in determining cyclomatic complexity.
9. Determine test cases necessary to test an ATM's functioning.
10. Determine the test cases and explain how the white box test can be performed on the following code.

1. READ X & Y
2. Z = X + Y
3. IF Z>1000
4. PRINT “JOB IS OVER”

11.9 KEYS TO MCQ QUESTIONS

1:c	2:c	3: d	4: a	5: a	6: c	7: d	8: b	9: a	10: d