BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 5

DCA3103

SOFTWARE ENGINEERING

Unit 8

Software Testing Techniques

Table of Contents

1. INTRODUCTION

Testing is a process of identifying the program behaviour for the set of sample test inputs. If it behaves as expected, then it is fine otherwise user need to note the condition under which failure occurs and later debug and correct the error. For this purpose, user need to define different test case design methods and testing techniques. Low-level testing is used to test the modules for correct implementation whereas high-level testing is for testing the system against the customer's requirements.

Software testing is an important phase in the creation of software that involves evaluating and validating a software programme or system to find any flaws, faults, or inconsistencies. It is essential to ensure the software's effectiveness, dependability, and efficiency of the software.

The main goal of software testing is to detect and address defects or bugs that may exist in the software, aiming to deliver a high-quality product that meets the requirements and expectations of its users.

1.1 Learning Objectives:

After studying this unit, students should be able to:

- ❖ *Discuss the generic characteristics of software testing.*
- ❖ *Describe testing strategies for conventional software.*
- ❖ *Differentiate white box and black box testing.*
- ❖ *Explain Boundary Analysis and GUIs.*

2. SOFTWARE TESTING FUNDAMENTALS

The purpose of testing is to identify errors, and an effective test has an elevated probability of doing so. As a result, "testability" should be considered while designing and implementing computer-based systems or products. In addition, the tests themselves must have a set of characteristics that allow them to detect most errors with the least amount of work.

James Bach defines **testability** as *"Software testability is simply how easily [a computer program] can be tested."*

*The following **characteristics** lead to testable software.*

i. ***Operability:*** The concept of operability in software engineering emphasizes the importance of designing and implementing a system with quality in mind. When a system is well-designed and implemented, it tends to have fewer bugs or issues that could delay the execution of tests. This, in turn, allows testing to proceed smoothly and efficiently.

ii. ***Observability:*** Observability is an important aspect of software testing that highlights the visibility and accessibility of system states, variables, inputs, outputs, errors, and source code during the execution of tests. It enables testers to effectively validate the behaviour of the system and identify any discrepancies or issues.

iii. ***Controllability:*** Controllability is a fundamental aspect of software testing that emphasizes the ability to control and manipulate the software system and its components during the testing process. It enables testers to automate and optimize testing activities effectively.

iv. ***Decomposability:*** Decomposability is a concept in software testing that highlights the ability to break down a software system into independent modules or components that can be tested individually. By isolating and testing these modules independently, testers can more quickly identify and isolate problems and perform targeted retesting.

v. ***Simplicity:*** Simplicity is a key principle in software testing that emphasizes the benefits of keeping the software system, its features, architecture, and codebase simple and streamlined. By promoting simplicity, testing can be conducted more quickly and effectively.

vi. ***Stability:*** It is an important factor in software testing that points to the need for a stable and robust software system. When a system is stable, it experiences fewer disruptions to the testing process.

vii. ***Understandability:*** It highlights the importance of having comprehensive information and documentation about the software system. When testers have a clear understanding of the system's architectural design, dependencies, and changes, they can perform smarter and more effective testing.

2.1 Testing Principles

The testing principles focus on the characteristics of a good test. A few principles related to the quality of tests are:

- *A good test has a high probability of finding an error:* A good test should have a high probability of uncovering defects or errors in the software. This means that the test case should be designed to target potential areas of weakness, boundary conditions, or critical functionalities where errors are more likely to occur. Tests that have a higher chance of finding errors provide valuable feedback to improve the quality of the software.
- *A good test is not redundant:* Redundant tests, which have the same purpose as existing tests, do not provide any additional value and unnecessarily consume testing time and resources.
- *A good test should be "best of breed" [Kan93]:* "best of breed" test suggests that when multiple tests have a similar objective or intent, and there are limitations on time and resources, it is essential to prioritize and execute the test that has the highest likelihood of uncovering a whole class of errors.
- *A good test should be neither too simple nor too complex:* The principle of balancing the complexity of tests emphasizes that a good test should neither be too simple nor too complex. Each test should be designed and executed separately to avoid masking errors and ensure clear visibility of test results.

3. TESTING STRATEGIES FOR CONVENTION SOFTWARE

The software engineering process may be viewed as a spiral, illustrated in Figure 1, initially system engineering defines the role of software and leads to software requirements and analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, user come to design and finally coding.

To develop computer software, the user spiral along streamlines that decrease the level of Abstraction on each turn.

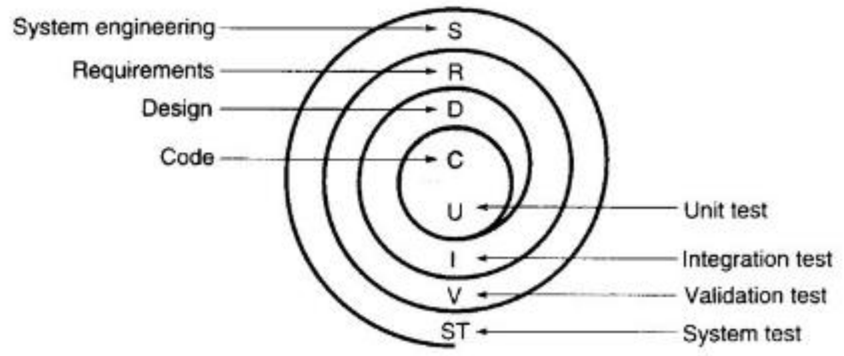


Figure 1: Testing Strategy

The strategy for software testing may also be viewed in the context of the spiral.

Unit testing begins at the start of the spiral and concentrates on each unit of the software as implemented in the source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on the design and the construction of the software architecture.

Validation testing is where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, the user aarrivesat system testing where the software and other system elements are tested.

To test computer software, the user spiral out along streamlines that broaden the scope of testing with each turn. Considering the process from a procedural point of view testing within the context of software engineering is a series of four steps that are implemented sequentially.

The steps are shown in Figure 2 initially tests focus on each module individually, assuring that it functions as a unit hence the name unit testing.

Unit testing makes heavy use of white box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, modules must be assembled or integrated to form the complete software package.

Integration testing addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), sets of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software needs all functional, behavioural and performance requirements. Black box testing techniques are used exclusively during validation.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering.

Software once validated must be combined with other system elements (e.g., hardware, people, and databases).

System testing verifies the tall elements mesh properly and that overall system function/performance is achieved.

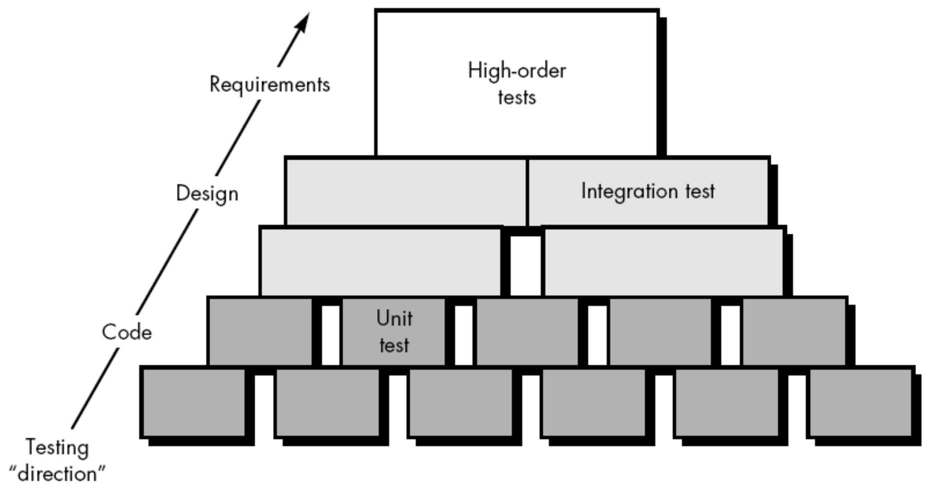


Figure 2: Software testing Steps

4. WHITE BOX TESTING:

White box testing is a testing technique that takes the internal logic and structure of the code into account. White box testing is also called structural testing or glass box testing or open box testing or unit testing. To perform white box testing on an application, the tester needs to possess knowledge of the internal working of the code. The tester needs to have a look inside the source code and find out which part of the code is behaving inappropriately. White box testing is often used for verification.

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases.

The white box testing technique does the following,

i. Ensures that all the different path the code execution can take place is tested at least once,
ii. Executes all the possible combinations of logical decision-making in the code,
iii. Execute all loops at their boundaries and within their operational bounds, and
iv. Ensures the validity of internal data structure by exercising them

Now let's discuss various white box testing techniques:

4.1 Basis Path Testing

Basis path testing is a white box testing technique first proposed by Tom McCabe. This method helps the test case designer to compute the measure of the logical complexity of the design and use this measure as the basis for fixing the set of execution paths. The test cases developed through the basis path testing method are guaranteed to execute each statement of the program at least once from the coverage point of view,

4.1.1 Flow Graph Notation

Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow graph node represents one or more procedural statements. The edges between nodes represent the flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements. A region in a flow graph is an area bounded by edges and nodes. Each node that contains a condition is called a predicate node. Cyclomatic complexity is a metric that provides a quantitative measure of the logical complexity of a program. It defines the number of independent paths in the basis set and thus provides an upper bound for the number of tests that must be performed.

Notation for representing control flow is shown in Figure 3.

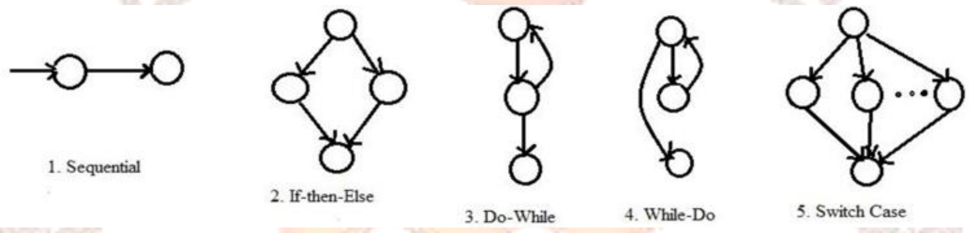


Figure 3: Flow Graph Notations

On a flow graph:

- Arrows called edges represent the flow of control.
- Circles called nodes represent one or more actions.
- Areas bounded by edges and nodes are called regions.
- A predicate node is a node containing a condition.

Any procedural design can be translated into a flow graph. Note that compound Boolean expressions at tests generate at least two predicate nodes and additional arcs.

4.1.2 Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. If the cyclomatic complexity is referred to in terms of basis path testing, the complexity derived provides the number of independent tests to be carried out to visit all statements at least once. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Cyclomatic complexity provides a measure of independent paths possible in the software which is a direct measure of the complexity of the application. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

For example, a set of independent paths for the flow graph illustrated in Figure 4 is given below:

Path 1: 1-11.

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

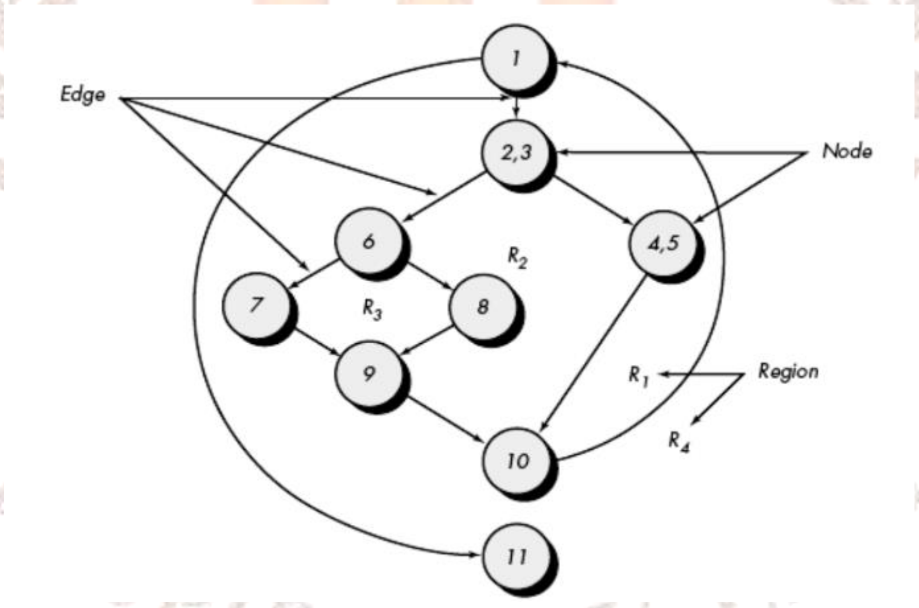


Figure 4: An Example of Flow Graph

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be independent because it is simply a combination of already specified paths and does not traverse any new edges.

Cyclomatic Complexity is computed in one of three ways:

1. *The number of regions of the flow graph corresponds to the cyclomatic complexity.*
2. *Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E –N + 2. Where E is the number of flow graph edges, N is the number of flow graph nodes.*
3. *Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1. Where P is the number of predicate nodes contained in the flow graph G.*

Referring again to the flow graph in Figure 4, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. V (G) = 11 edges – 9 nodes + 2 = 4.
3. V (G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 4 is 4.

Advantages of White Box testing

i) Since the tester has the pre-requisite knowledge of the software, the testing becomes more effective as a tester can define a meaningful set of test data and test conditions.
ii) White box testing helps in code optimization
iii) It helps in the removal of unnecessary code from the application as these can cause some unknown defects in the long run of the application.

Disadvantages of White Box testing:

i) For performing white box testing the tester should know the code and internal structure of the application. This prerequisite increases the cost of testing.
ii) Though white box testing intends to test the internal structure of the application, it is impossible to look at every line of code and various code combinations that execution can take place.

5. CONTROL STRUCTURE TESTING

Control structure testing as the name suggests focuses on testing the control statements or loops in the software. The different types of control structure testing are listed as follows.

Loop testing:

Loops are fundamental to many algorithms and need thorough testing.

There are four different classes of loops: simple, concatenated, nested, and unstructured as shown in Figure 5.

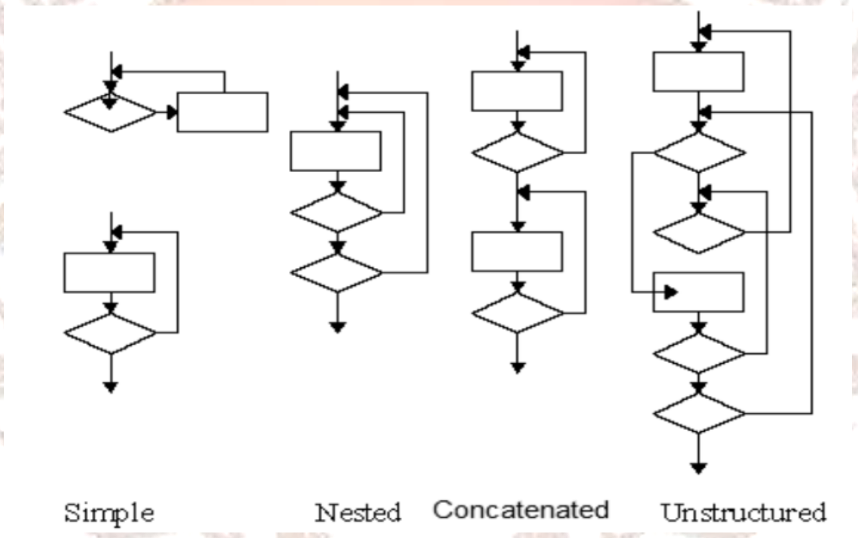


Figure 5: Loop Structures

- **Simple loops,** suppose n is the number of passes allowed through the loop, the testing would involve.
  - Skip the loop entirely.
  - test with only one pass
  - test with the passes
  - try with m passes where m<n.
  - (n-1), n, and (n+1) passes through the loop.
- **Nested loops:** For nested loops test as follows,
  - Start with the inner loop. Set all other outer loops with terminal conditions attaching minimum value.

  - Conduct simple loop testing on the inner loop.
  - extends towards outer loops
  - Complete the testing continuing in the same fashion covering all the loops.
- **Concatenated loops**
  - If the loops are not inter-dependent execute the test as simple loops.
  - If the loops are dependent, then carry out the testing as nested loops.
- **Unstructured loops**
  - Do not test this code, instead redesign the solution.

6. BLACK-BOX TESTING

Black Box Testing is also called functional testing. In this testing, the tester just focuses on the inputs and outputs of the software system without having any internal knowledge of the program. Black box testing is often used for validation.

In this method, the software's functional requirements are focused. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black box testing is opposite to white box testing, which can be used to uncover the errors which are not detected during white box testing.

In the following categories, attempt to find the errors made by the black box testing methods:

i. Errors because of missing or incorrect functions,
ii. Errors at the interface,
iii. data structure errors or database access, which is external,
iv. errors in behaviour or performance, and
v. errors in initialization and termination of the software.

Equivalence partitioning is a black box testing technique that segregates the input data into differenttestingpos different testing possibilitiesses. The number of test cases can be decreased by an ideal test case that can identify a variety of problems.

Test case design for equivalence partition is relevant to the evaluation of the input equivalence. An equivalence class would be present if the set of objects in the preceding section can be conjugated/associated in relations that are symmetric, transitive, and reflexive. An equivalence class represents the validity state for input conditions.

Typically, an input condition is a specific numeric value, a range of values, a set of related values, or a Boolean condition. According to the following guidelines, one can define the equivalence classes:

i. One valid and two invalid equivalence classes are defined if a range is specified by the input condition.

ii. One valid and two invalid equivalence classes are defined if a specific value is required by the input condition
iii. One valid and one invalid equivalence class are defined, if the member set is specified by the input condition
iv. One valid and one invalid equivalence class are defined, if Boolean is the input condition.

As an example, consider an automated banking application where data is maintained as part of it. By using the personal computer, the user can access the bank by entering the password which is of six digits and then follow the commands that trigger the different banking functions. During the

log-on sequence, the software supplied for the banking applications data in the form.

Area code – blank or three-digit number

Prefix – three-digit number not beginning with 0 or 1.

The suffix – a four-digit number

Password – a six-digit alphanumeric string

Commands – check, deposit, bill pay, and the like

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

Area code: Input condition, Boolean-the area code may or may not be present.

Input condition, range-values defined between 200 and 999, with specific exceptions.

Prefix: Input condition, range-specified value >200

Input condition, value-four-digit length

Password: Input condition, Boolean-a password may or may not be present.

Input condition, value-six-character string.

Command: Input condition, set-containing commands noted previously.

The test cases for each data item can be developed by applying the above guidelines and then executed. The test cases are selected in such a way that at once the largest number of equivalence class attributes should be exercised.

7. BOUNDARY VALUE ANALYSIS

BVA focuses on the boundaries of the input domain because it has been observed that a significant number of errors tend to occur at these boundaries rather than in the centre of the range. By targeting these boundary values, BVA aims to uncover potential issues and defects that may arise due to incorrect handling or processing of these critical values.

BVA is a complementary technique to equivalence partitioning, which is another test-case design technique. While equivalence partitioning divides input values into groups or classes, BVA specifically selects test cases at the edges or boundaries of these classes. Instead of testing every element within an equivalence class, BVA focuses on values that fall on the boundaries of the classes. This approach helps identify issues related to boundary conditions, off-by-one errors, and other edge cases that may not be adequately covered by testing within the equivalence class itself.

BVA not only considers input conditions but also derives test cases from the output domain. This means that it considers the expected output or result of the software based on the given input values. By considering the output domain, BVA ensures that the software generates correct outputs and behaves as expected for various boundary values of the input.

The guidelines for conducting Boundary Value Analysis are like those for equivalence partitioning:

a) *Identify Boundary Values:* Identify the lower and upper boundaries for each input parameter or condition being tested. These boundaries define the edge values or limits that separate one equivalence class from another as shown in Figure 6.

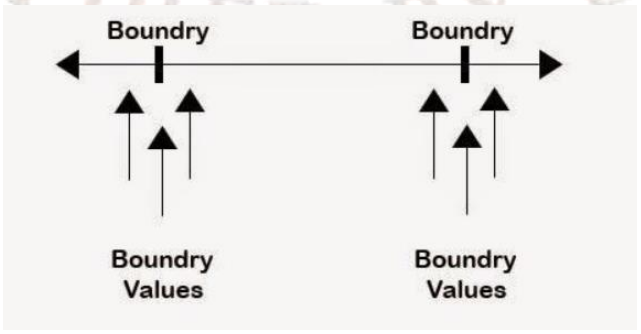


Figure 6: Boundary Value Analysis

b) *Design Test Cases:* Create test cases that exercise the boundaries and edge values identified in the previous step. Test cases should include values at the lower boundary, values just above the lower boundary, values at the upper boundary, and values just below the upper boundary. These values are considered critical because errors are more likely to occur around these boundaries.

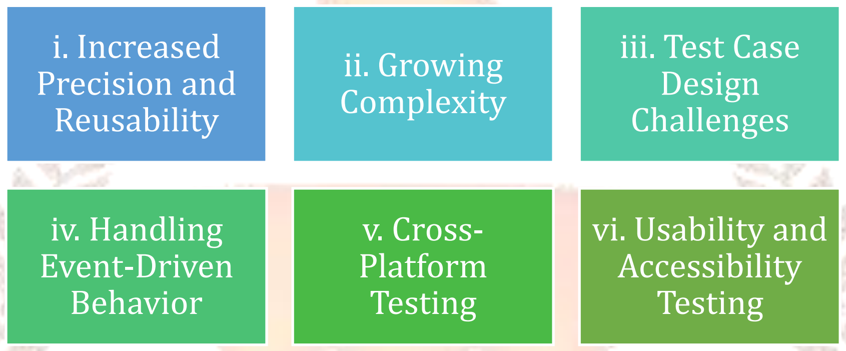
c) *Include Invalid Values:* Test cases should also include invalid values that lie outside the valid range. For example, if an input field only accepts values from 1 to 100, test cases should include values less than 1 and greater than 100 to verify that the software handles such invalid inputs correctly.

d) *Verify Expected Behaviour:* Execute the test cases and observe the software's behaviour. Verify that the software handles the boundary values correctly and produces the expected results. Pay close attention to how the software handles values at the boundaries, as errors are more likely to occur in these scenarios.

e) *Consider Output Domain:* In addition to testing input values, BVA also considers the output domain. Test cases should cover a range of output values to ensure that the software generates correct outputs based on the given input range.

8. GUIS TESTING

Testing graphical user interfaces (GUIs) presents unique challenges due to the increasing complexity and the use of reusable components in GUI development environments. Here are some key points regarding testing GUIs:


i. Increased Precision and Reusability
ii. Growing Complexity
iii. Test Case Design Challenges
iv. Handling Event-Driven Behavior
v. Cross-Platform Testing
vi. Usability and Accessibility Testing

i. *Increased Precision and Reusability:* GUI development environments now offer reusable components, which streamline the creation of user interfaces and improve precision. This means that GUIs can be constructed more efficiently, with less manual coding required. However, this also means that testing must ensure the correct integration and behaviour of these reusable components, as any issues could affect multiple parts of the GUI.

ii. *Growing Complexity:* GUIs have become more complex over time. Modern GUIs often feature dynamic content, multiple input methods (keyboard, mouse, touch), various screen resolutions, and diverse user interactions. Testing such complex GUIs requires a thorough understanding of the underlying functionality, proper handling of different input methods, and the ability to simulate various user scenarios.

iii. *Test Case Design Challenges:* Designing effective test cases for GUIs can be challenging due to the wide range of possible user interactions, screens, and input combinations. Testers need to consider all possible user actions, validate input handling, and verify that the GUI responds correctly to different scenarios. Test case design techniques like equivalence partitioning, boundary value analysis, and exploratory testing can be applied to create comprehensive test cases.

iv. *Handling Event-Driven Behaviour:* GUIs often rely on event-driven programming, where user actions trigger various events and responses. Testing GUIs involves verifying that the appropriate events are triggered and that the GUI reacts as expected. Testers need to consider both positive and negative scenarios to validate the GUI's responsiveness, error handling, and event synchronization.

v. *Cross-Platform Testing:* GUIs are typically developed for multiple platforms, such as desktop, web, and mobile. Each platform may have its specific nuances and requirements. Testing should cover different platforms, operating systems, browsers, and screen resolutions to ensure consistent functionality and user experience across various environments.

vi. *Usability and Accessibility Testing:* GUI testing goes beyond functional validation and includes assessing the usability and accessibility aspects of the interface. Testers need to evaluate the GUI's ease of use, responsiveness, visual consistency, error messaging, and compliance with accessibility standards. Usability testing techniques like user surveys, user feedback, and heuristic evaluation can provide valuable insights.

To effectively test GUIs, testers need a combination of technical skills, knowledge of GUI development principles, and an understanding of user expectations. Automation tools and frameworks specifically designed for GUI testing can also assist in accelerating and enhancing the testing process. By addressing the unique challenges of GUI testing, software teams can ensure the reliability, usability, and quality of the graphical user interfaces they develop.

9. SUMMARY

By incorporating the characteristics into the design and implementation of computer-based systems or products, the testability of the software is improved. This, in turn, facilitates the detection of errors and contributes to the overall quality of the system.

By adhering to the testing principles, testers can design and execute tests that are more likely to uncover errors, avoid redundancy, prioritize strategically, and strike the right balance between simplicity and complexity. This helps in maximizing the efficiency and effectiveness of the testing process and ultimately contributes to the overall quality of the software.

White box testing is a testing technique that takes the internal logic and structure of the code into account. White box testing is also called structural testing or glass box testing or open box testing or unit testing. In white box testing, the tester needs to have a look inside the source code and find out which part of the code is behaving inappropriately. White box testing is often used for verification.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

Loops are fundamental to many algorithms and need thorough testing and there are four different classes of loops: simple, concatenated, nested, and unstructured.

Validation can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Most software product builders use a process called alpha-beta testing to uncover errors that only the end user seems able to find.

System testing is testing to make sure that by keeping the software in different environments (for example, Operating Systems) it still works.

Black Box Testing is also called functional testing. In this testing, the tester just focuses on the inputs and outputs of the software system without having any internal knowledge of the program. Black box testing is often used for validation.

Boundary Value Analysis is a valuable technique as it targets areas where errors are likely to occur, providing effective test coverage. By focusing on boundary conditions, it helps identify and address potential issues related to how the software handles extreme values and limits.

10. SELF–ASSESSMENT QUESTIONS

1. __________ activity can be planned and conducted systematically.
2. Testing and debugging are the same activities (True/False).
3. The software engineering process may be viewed as a ________.
4. ________ testing provides final assurance that software needs all functional, Behavioral and performance requirements.
5. _____________Testing is also called as functional testing.
6. Black box testing is often used for validation. (True/False).
7. _____________ is a black box testing technique that segregates the input data into different testing possibilities/cases.
8. _________ testing is a testing technique that takes internal logic and
9. structure of the code into account.
10. __________ testing is also called structural testing or glass box
11. testing or open box testing or unit testing.
12. White box testing is often used for ___________.
13. _______ testing is a white box testing technique first proposed by Tom McCabe.
14. ____________ is a metric that provides a quantitative measure of the logical complexity of a program.
15. Cyclomatic complexity, V(G), for a flow graph, G, is defined as________ Where E is the number of flow graph edges, and N is the number of flow graph nodes.
16. Mention any two kinds of loops.

11. SELF-ASSESSMENT ANSWERS

1. Testing
2. False
3. Spiral
4. Validation
5. Black Box
6. True
7. Equivalence partitioning
8. White box
9. White box
10. Verification
11. Basis path
12. Cyclomatic complexity
13. V(G) = E –N + 2
14. Simple and nested

12. TERMINAL QUESTIONS

1. Briefly explain the characteristics of software testing.
2. Elucidate the testing principles.
3. Explicate the strategies for convention software.
4. Write a short note on
    a. White Box Testing
    b. Black Box Testing
5. Explain the different flow graph notations.
6. With an example explain Cyclomatic Complexity.
7. Define Loop and explain its types. (8.5)
8. Briefly explain the guidelines for conducting Boundary Value Analysis for equivalence partitioning.

13. TERMINAL ANSWERS

1. Refer to Section 8.2
2. Refer to Section 8.2.1
3. Refer to Section 8.3
4. a. Refer to Section 8.4

   b. Refer to Section 8.6
5. Refer to Section 8.4.1.1
6. Refer to Section 8.4.1.2
7. Refer to Section 8.5
8. Refer to Section 8.7