BACHELOR OF COMPUTER APPLICATIONS

SEMESTER *5*

DCA3103

SOFTWARE ENGINEERING

Unit 10

Software Testing Strategies

Table of Contents

1. INTRODUCTION

Software testing is a crucial process that helps ensure the quality, reliability, and correctness of software systems. Software testing strategies are systematic approaches employed to design and execute tests that identify defects, errors, or vulnerabilities in software applications. These strategies are aimed at uncovering issues in the software early in the development cycle, which helps in delivering a higher-quality product to end users. Some, of the testing strategies are Unit Testing, Integration Testing, System Testing, Acceptance Testing, Performance Testing, Security Testing etc.

software testing is a crucial process that helps ensure the quality, reliability, and correctness of software systems. Software testing strategies are systematic approaches employed to design and execute tests that identify defects, errors, or vulnerabilities in software applications. These strategies are aimed at uncovering issues in the software early in the development cycle, which helps in delivering a higher-quality product to end users.

1.1 Learning Objectives

At the end of the unit, students should be able to,

- *Recall the characteristics and goals of different software testing strategies.*
- *Describe the relationship between different testing strategies and their role in ensuring software quality.*
- *Apply unit testing techniques to write effective test cases for individual software components.*
- *Design integration test cases to validate the interactions between different software modules.*

2. A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is an important part of software development, and it involves planning and conducting activities systematically. To make this process more efficient, it's beneficial to have a software testing template—a predefined set of steps that can accommodate various test case design techniques and testing methods.

In the literature, several software testing strategies have been proposed, each offering a template for testing with certain common characteristics. Here are some key points to consider:

i. Conducting effective technical reviews before testing starts can greatly reduce the number of errors that need to be addressed during testing.
ii. Testing begins at the component level and gradually moves towards integrating the entire computer-based system.
iii. Different testing techniques are suitable for different software engineering approaches and stages of development.
iv. Testing is typically carried out by both the software developer and, for larger projects, an independent test group.
v. It's important to note that testing and debugging are distinct activities, but any testing strategy should account for the need to address debugging.

A well-defined software testing strategy should accommodate both low-level tests, which verify the correct implementation of small source code segments, and high-level tests which validate major system functions against customer requirements. It should guide practitioners and establish milestones for managers. Since testing usually occurs when deadline pressure increases, it's crucial to have measurable progress and identify problems as early as possible.

By adopting an effective software testing strategy and following a predefined set of steps, the software development team can streamline their testing efforts, identify, and resolve issues, and ensure that the software meets both technical and customer requirements.

2.1 Organizing for Software Testing

During software testing, a conflict of interest arises as developers are tasked with testing the software they built. While developers possess extensive knowledge of the program, they also have a vested interest in proving that it is error-free, meets customer requirements, and adheres to the project's timeline and budget. Unfortunately, these interests may make it difficult to conduct thorough tests.

The **software engineer** goes through the process of analysing, modelling, and creating a computer program along with its documentation. When testing starts, there is an underlying motivation to protect the software engineer's creation. Testing may appear harmful conceptually from their point of view. As a result, the engineer takes testing cautiously and concentrates on proving that the programme works rather than finding mistakes. But mistakes happen, and if the engineer misses them, the consumer will experience them.

The **software developer** is always responsible for testing each unit (component) of the programme to make sure that it operates as intended or behaves in the desired manner. The developer frequently also does integration testing, which is a testing phase that precedes the building (and testing) of the entire software architecture. An independent test team doesn't get engaged until the software architecture is finished.

An **independent test group (ITG)** plays a vital role in addressing the challenges that arise when the software builder tests their creation. Independent testing eliminates any conflicts of interest that could arise in such situations. After all, ITG members are specifically tasked with finding errors and issues in the software.

However, it's important not to simply hand over the program to the ITG and step away. The developer and the ITG work closely together throughout the software project to ensure thorough testing. While testing takes place, the developer needs to be readily available to address and fix any errors that are uncovered.

The ITG is considered part of the software development project team. They get involved early on during the analysis and design phases and remain involved throughout the project, particularly in planning and specifying test procedures. In many cases, the ITG reports to the

software quality assurance organization, which allows them to maintain a level of independence that might be difficult to achieve if they were part of the software engineering organization.

3. SOFTWARE TESTING STRATEGY

The software process may be viewed as a spiral, as shown in Figure 1.

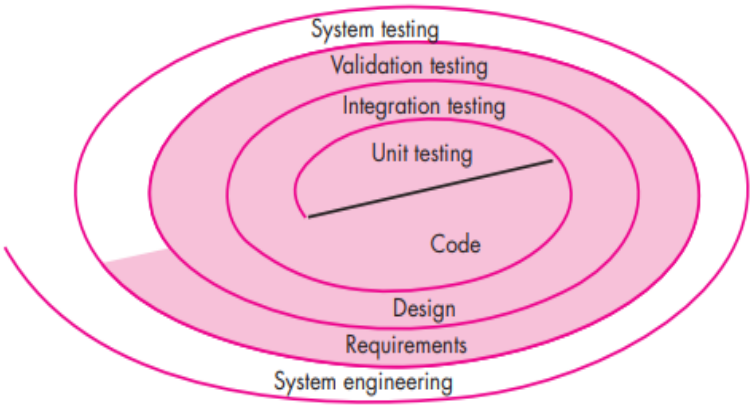

Figure 1: Testing Strategy

In the beginning, system engineering sets the stage by defining what the software should do. The next step is software requirements analysis, where we go deeper into the details of the tasks the programme should complete, its behaviour, and its performance. We also consider any constraints or limitations it should adhere to. It's like building a roadmap for the software. As we move further inward on our development journey, we reach the design phase, where we plan how the software will be structured and organized. Finally, we arrive at the coding stage, where we write the lines of code that bring the software to life. It's like unravelling the layers of abstraction, getting closer and closer to the actual implementation of the software. It's an iterative process, and with each cycle, we refine and improve the software.

As shown in Figure 1, Unit testing is like starting at the centre of a spiral. It zooms in on each unit of the software, such as components, classes, or WebApp content objects, by examining the source code. As you move outward along the spiral, you reach integration testing, which focuses on the design and construction of the software architecture. Continuing along the spiral, you come across validation testing, where the requirements established during requirements modelling are validated against the constructed software. Finally, you arrive at system testing, where the entire software and other system elements are tested. Think of

the process of testing computer software as a clockwise tour along streamlines, where each turn broadens the scope of the test.

The testing process in software engineering can be broken down into **four sequential steps** which are illustrated in Figure 2.

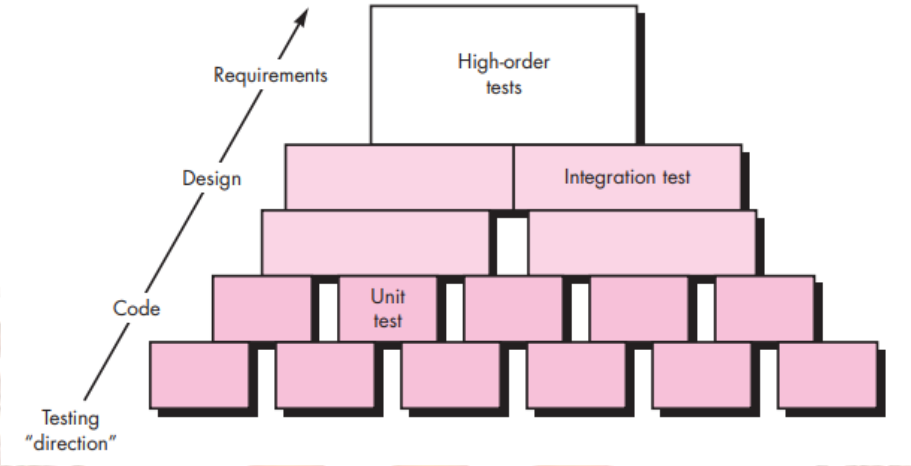

Figure 2: Software Testing Steps

First, start with unit testing. This step focuses on testing each component individually to ensure they function correctly on their own. That's why it's called unit testing. During unit testing, we use testing techniques that explore specific paths in a component's control structure to achieve comprehensive coverage and detect as many errors as possible.

Once the components are tested, we move on to integration testing. Here, the focus shifts to assembling or integrating the components to create the complete software package. Integration testing addresses the challenges of both verification and program construction. While we still consider input and output during this phase, techniques that examine specific program paths may also be used to ensure major control paths are covered.

After integration, we move to the next step: validation testing. This step is crucial as it evaluates whether the software meets all the requirements established during the requirements analysis phase. We look at informational, functional, behavioural, and performance requirements to ensure the software aligns with the intended goals.

The final high-level testing step goes beyond software engineering and enters the realm of computer system engineering. It's system testing. At this stage, the validated software is combined with other system elements such as hardware, people, and databases. System testing verifies that all these elements work together seamlessly and achieve the desired overall system function and performance.

So, *these four steps—unit testing, integration testing, validation testing, and system testing*—form a logical sequence in the testing process of software engineering. They help ensure that the software functions properly at various levels, meet requirements, and operates smoothly within the broader system context.

3.1 Strategic Issues

A software testing strategy will succeed when software testers:

i. Specify product requirements in a quantifiable manner long before testing commences.
ii. State testing objectives explicitly.
iii. Understand the users of the software and develop a profile for each user category.
iv. Develop a testing plan that emphasizes “rapid cycle testing”.
v. Build “robust” software that is designed to test itself.
vi. Use effective technical reviews as a filter before testing.
vii. Conduct technical reviews to assess the test strategy and test cases themselves.
viii. Develop a continuous improvement approach for the testing process.

4. UNIT TESTING

Unit testing targets the smallest unit of software design, such as a component or module. It tests important control paths within the module to identify errors. Unit testing has a limited scope, which constrains the complexity of tests and the errors they can uncover. It focuses on the internal processing logic and data structures within a component. Multiple components can undergo unit testing simultaneously.

Unit tests are illustrated schematically in Figure 3.

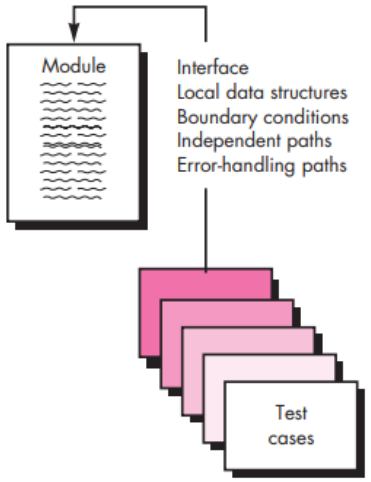

Figure 3: Unit Test

Information flow into and out of the programme unit under test is monitored by testing the module interface. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

The module's functionality at boundaries set up to restrict or limit processing is evaluated to ensure that boundary constraints are met. All error-handling paths are then tested.

Before starting any additional testing, a component interface's data flow is tested. All other tests are meaningless if data cannot enter and exit properly. Additionally, local data

structures should be tested, and during unit testing, it is advisable to determine (if possible) how local changes would affect global data.

Selective testing of execution paths is a crucial aspect of unit testing. Test cases should be designed to uncover errors resulting from incorrect computations, improper comparisons, or incorrect control flow.

Boundary testing holds significant importance in unit testing. Software often encounters failures at its boundaries, such as when processing the nth element of an n-dimensional array, invoking the ith repetition of a loop with I passes, or encountering the maximum or minimum allowable values. Test cases that encompass data structures, control flow, and data values just below, at, and just above maximum and minimum values are highly likely to reveal errors.

4.1 Unit – Test Procedures

Unit testing is often seen as a complementary step to coding. The design of unit tests can be done either before coding or after the source code has been generated. Reviewing the design information helps in creating test cases that are likely to detect errors in the discussed categories. Each test case should be accompanied by a set of expected results.

Since a component is not a standalone program, driver and/or stub software are often required for each unit test. The unit test environment is depicted in Figure 4. In most cases, a driver acts as a "main program" that accepts test case data, passes it to the component being tested, and prints relevant results. Stubs, on the other hand, replace subordinate modules invoked by the component under test. Stubs simulate the behaviour of the subordinate module's interface, perform minimal data manipulation, print entry verification, and return control to the module being tested.

Drivers and stubs contribute to testing overhead. They are software components that need to be developed (without formal design in most cases) but are not included in the final software product. If drivers and stubs are kept simple, the actual overhead is relatively low. However, some components may require more complex overhead software, making it

challenging to perform adequate unit testing. In such cases, complete testing may be postponed until the integration testing phase, where drivers or stubs are also utilized.

Unit testing becomes easier when components with high cohesion are designed. When a component focuses on a single function, the number of test cases is reduced, and it becomes easier to predict and uncover errors.

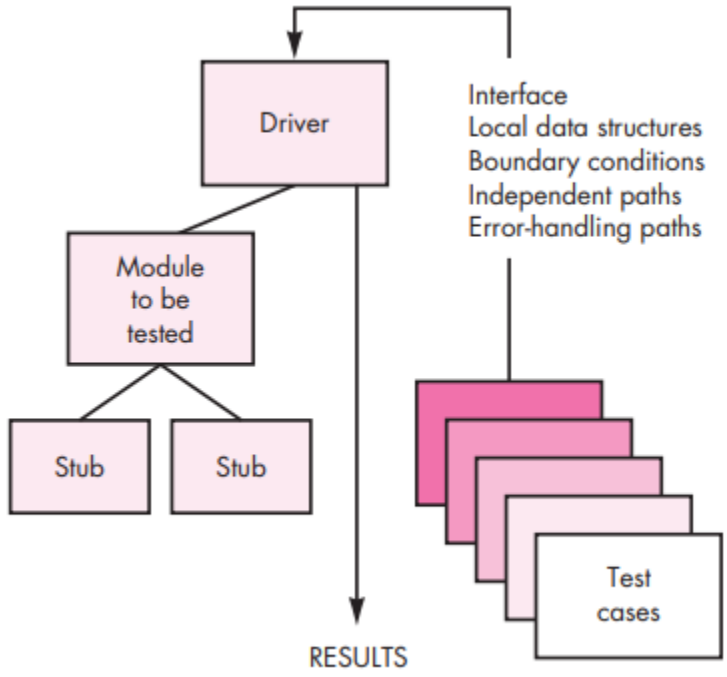

Figure 4: Unit-Test Environment

Benefits of Unit Testing

- Early detection and correction of defects
- Verification of individual units' functionality
- Facilitating code refactoring and maintainability
- Supporting the process of integration testing

5. INTEGRATION TESTING

Integration testing is a critical phase in software development where individual units or components of a system are combined and tested together as a group. The primary goal of integration testing is to identify and resolve any defects that may arise due to interactions between these integrated components. It ensures that different parts of the software work together as expected and that the system functions.

There are *two main approaches to integration testing:*

incremental integration and

non-incremental (big bang) integration.

5.1 Incremental Integration Testing

Incremental integration involves gradually integrating and testing individual components or units of the software. The process proceeds step by step, where each new component is added to the existing integrated components, and the system is tested after each addition. The testing process continues iteratively until all components are integrated and tested together. This approach allows developers to detect and fix issues early on and incrementally build up the system.

There are two main strategies within incremental integration testing:

a. ***Top-Down Integration Testing:*** In this approach, testing starts with the highest-level components (such as the main module) and then gradually integrates lower-level modules. Stub modules or simulators are used for components that are not yet available to simulate their behaviour during testing.

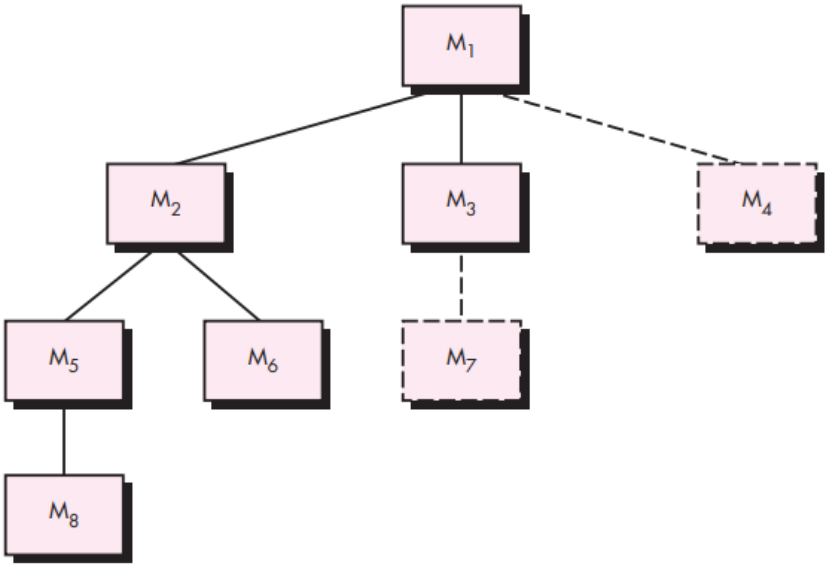

Figure 5: Top-down integration

Referring to Figure 5, Depth-first integration testing involves integrating all components along a significant control path of the program's structure. The choice of which path to integrate first is somewhat arbitrary and relies on specific characteristics of the application. For instance, if the left-hand path is selected, components M1, M2, and M5 would be integrated initially. Following that, M8 or, if necessary for the proper functioning of M2, M6 would be integrated. Subsequently, the central and right-hand control paths are constructed.

On the other hand, breadth-first integration testing includes integrating all components that are directly subordinate at each level, progressing across the program structure horizontally. Based on the given figure, components M2, M3, and M4 would be the first ones to be integrated using this approach. The next control level will be M5, M6 and so on and continues.

The integration process consists of a sequence of five steps:

- The primary control module acts as a test driver, and for all components directly subordinate to it, stubs are used as substitutes.
- Depending on the chosen integration approach (whether depth or breadth-first), the subordinate stubs are gradually replaced one by one with actual components.
- Tests are performed as each component is integrated.

- After completing each set of tests, another stub is substituted with the actual component.
- Regression testing (which will be discussed later in this section) might be carried out to verify that no new errors have been introduced.

b. ***Bottom-Up Integration Testing:*** In this approach, testing starts with the lowest-level components (e.g., individual functions) and gradually integrates higher-level modules. Driver modules are used to simulate the behaviour of higher-level components during testing.

The process of implementing a bottom-up integration strategy involves the following steps:

- Grouping low-level components into clusters, also known as builds, which carry out specific subfunctions of the software.
- Creating a driver, which is a control program used for testing, to manage the input and output of test cases.
- Conducting testing on each cluster.
- Gradually combining the clusters, moving upwards in the program structure, and removing the drivers as integration progresses.

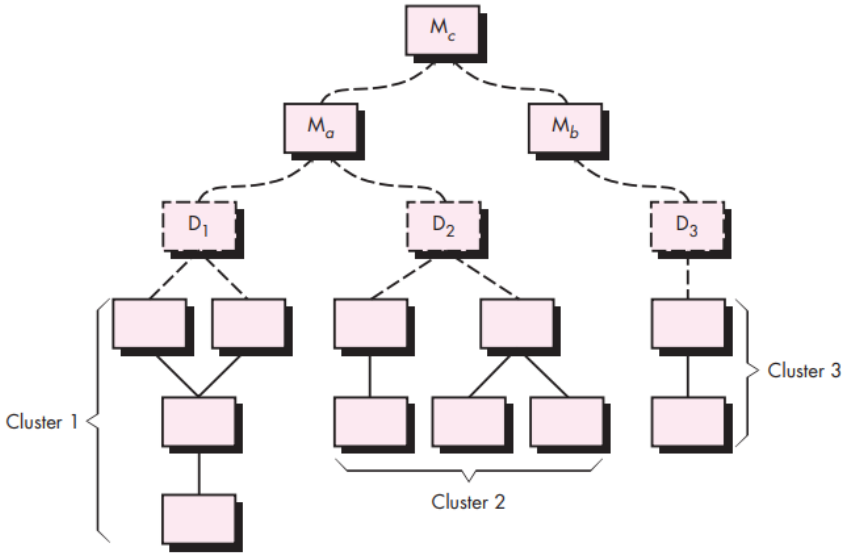

Figure 6: Bottom-up Integration

Integration follows the pattern depicted in Figure 6, where components are combined to create three clusters, namely, clusters 1, 2, and 3. Each of these clusters is subjected to testing using a driver, represented by a dashed block in the figure. Specifically, components within clusters 1 and 2 are subordinate to Ma, and drivers D1 and D2 facilitate their testing. Similarly, cluster 3 is tested with the aid of driver D3 before its integration with module Mb. As the integration process progresses upward, the requirement for individual test drivers diminishes. When the top two levels of the program structure are integrated top-down, the number of drivers can be substantially reduced, resulting in a significant simplification of cluster integration.

Ultimately, both Ma and Mb will be integrated with component Mc, and this process continues similarly as integration moves forward.

Advantages and Disadvantages of Incremental Integration Testing:

Advantages: Early detection of defects, more manageable debugging, and easier to identify the root cause of issues.

Disadvantages: More effort is required to set up the testing environment, the potential overhead of creating stubs and drivers.

5.2 Non-Incremental (Big Bang) Integration Testing:

The non-incremental or big bang integration testing approach involves integrating all the components simultaneously and then conducting testing. This method does not follow the gradual step-by-step integration process; instead, it integrates everything at once.

In big bang integration testing, all components are combined into the system and tested as a whole. While this approach may be faster, it can be challenging to pinpoint the exact source of defects if they occur, as all components are integrated simultaneously.

Advantages and Disadvantages Non-Incremental (Big Bang) Integration Testing:

Advantages: Faster to set up, no need to create stubs or drivers.

Disadvantages: Difficult to isolate the source of defects, and issues in the interaction between multiple components may not be apparent until the big bang testing phase, potentially causing more significant delays.

5.3 Regression Testing

Whenever a new module is added during integration testing, the software changes. These changes can establish new data flow paths, introduce new input/output interactions, and invoke new control logic.

As a result, functions that previously worked perfectly may encounter issues. In the context of an integration test strategy, regression testing is the process of re-running a subset of tests that have already been executed to ensure that unintended side effects have not propagated due to the changes.

Successful tests, regardless of their type, help to uncover errors, which then need to be addressed and corrected. Whenever software is modified, whether due to testing or other reasons, it alters some aspect of the software configuration, such as the program itself, its documentation, or the associated data. Regression testing plays a crucial role in ensuring that these changes do not introduce unintended behaviour or new errors.

Regression testing can be performed manually by re-executing a selected subset of test cases, or automated capture/playback tools can be used. These tools allow software engineers to record test cases and their outcomes for subsequent playback and comparison.

The regression test suite, which comprises the subset of tests to be executed, includes three different classes of test cases:

1. A representative sample of tests that exercise all software functions.
2. Additional tests that focus on software functions likely to be affected by the recent changes.
3. Tests that specifically target the software components that have undergone modifications.

As integration testing advances, the volume of regression tests may significantly increase. To handle this effectively, it is essential to thoughtfully craft the regression test suite, incorporating tests that target different classes of errors within each major program function. Attempting to re-run every test for each program function following a change becomes impractical and inefficient. Thus, a well-designed regression test suite optimizes the testing process and ensures thorough coverage without unnecessary redundancy.

5.4 Smoke Testing

Smoke testing is a widely used integration testing approach during software product development. It serves as a time-critical pacing mechanism, allowing the software team to regularly evaluate the project's progress.

The essence of the smoke-testing approach involves the following activities:

1. Integration of software components, which have been translated into code, into a build. A build comprises all necessary data files, libraries, reusable modules, and engineered components required to implement one or more product functions.
2. Designing a series of tests aimed at uncovering errors that could prevent the build from functioning correctly. The focus is on identifying "showstopper" errors that have the highest likelihood of causing delays in the software project.
3. Performing daily smoke testing of the entire product by integrating the build with other builds. The product, in its current state, is subjected to smoke testing daily. The integration approach may follow either a top-down or bottom-up strategy.

The daily frequency of testing the entire product might surprise some readers, but it provides both managers and practitioners with a realistic assessment of the progress in integration testing.

As described by ***McConnell [McC96],*** *"The smoke test should exercise the entire system from end to end. It doesn't need to be exhaustive, but it should be capable of detecting major problems. The smoke test should be comprehensive enough that if the build passes, one can assume it is stable enough for more thorough testing."*

Smoke testing offers several advantages when applied to complex and time-critical software projects:

i. *Minimized Integration Risk:* By conducting smoke tests daily, potential incompatibilities and critical errors are identified early in the integration process. This proactive approach reduces the chances of serious schedule delays when such issues are detected.

ii. *Improved End Product Quality:* Since smoke testing is integration-focused, it has the potential to uncover not only functional errors but also architectural and component-level design flaws. Early detection and correction of these errors lead to an overall improvement in the quality of the final product.

iii. *Simplified Error Diagnosis and Correction:* Like other integration testing approaches, errors uncovered during smoke testing are typically associated with the recently added software increments. This makes it easier to pinpoint the cause of the newly discovered errors and facilitates their prompt resolution.

iv. *Enhanced Progress Assessment:* As smoke testing progresses each day, more of the software is integrated, and its functionality is demonstrated. This creates a positive impact on team morale and provides managers with a clear indication of the project's progress.

6. SUMMARY

Organizing software testing involves establishing a systematic approach to testing within a software development project. It includes defining roles and responsibilities for testing team members, creating test plans and test cases, setting up testing environments, and ensuring effective communication between developers and testers. Proper organization helps streamline the testing process and ensures that software meets quality standards before deployment.

A software testing strategy is a comprehensive plan that outlines the testing approach and methodologies to be employed throughout the software development lifecycle. It involves determining the types of testing (e.g., unit testing, integration testing, system testing, etc.), the testing scope, resource allocation, automation strategies, and the criteria for test completion. A well-defined testing strategy ensures thorough testing and early detection of defects, leading to a more reliable and stable software product.

Unit testing is the first level of testing and focuses on verifying the correctness of individual units or components of the software in isolation. Each unit is tested independently to ensure that it functions as intended and produces the expected results. Developers typically conduct unit tests and automated testing frameworks are often utilized to streamline the process and maintain code quality.

Top-down integration testing is an incremental approach to integration testing where testing starts with the highest-level components (e.g., main module) and gradually incorporates lower-level modules. Stub modules are used for components that are not yet available to simulate their behaviour during testing. This method allows for early detection of integration issues and better debugging.

Bottom-up integration testing is another incremental approach to integration testing, but it starts with the lowest-level components (e.g., individual functions) and progressively integrates higher-level modules. Driver modules are used to simulate the behaviour of higher-level components during testing. Bottom-up integration also facilitates early detection of integration problems.

7. SELF-ASSESSMENT QUESTIONS

1. What is the purpose of organizing software testing?
   A) To allocate resources for software development
   B) To identify software requirements
   C) To plan and manage the testing process
   D) To design the software architecture
2. Which of the following statements best describes a software testing strategy?
   A) A plan for unit testing only
   B) A detailed project schedule
   C) A high-level approach to conducting software testing
   D) A set of coding guidelines for developers
3. Which type of testing focuses on verifying the smallest testable units of code in isolation?
   A) Integration testing
   B) System testing
   C) Unit testing
   D) Regression testing
4. In top-down integration testing, which components are tested first?
   A) The lowest-level components
   B) The highest-level components
   C) The middle-level components
   D) Components are randomly selected for testing
5. Which approach to integration testing involves gradually adding and testing individual components until all units are integrated?
   A) Incremental integration testing
   B) Big bang integration testing
   C) Top-down integration testing
   D) Bottom-up integration testing
6. What is the main advantage of bottom-up integration testing?
   A) Early detection of high-level defects

B) Faster integration process

C) Easier to identify the root cause of issues

D) Reduced overhead of creating stubs and drivers

7. During software testing, what does regression testing aim to ensure?

A) All test cases are executed in a single run

B) New functionalities have been implemented correctly

C) No new defects have been introduced by recent changes

D) All code has been reviewed and approved

8. What does smoke testing primarily focus on?

A) Comprehensive testing of all software functionalities

B) Integration of different software modules

C) Identifying show-stopper errors early in the testing process

D) Validation of end-user requirements

9. Which of the following best describes the purpose of a unit test?

A) To test the entire system end-to-end

B) To test individual units of code in isolation

C) To verify the integration of multiple modules

D) To test the software in a real-world environment

10. In which integration testing approach are stubs and drivers commonly used?

A) Incremental integration testing

B) Big bang integration testing

C) Top-down integration testing

D) Bottom-up integration testing

8. SELF-ASSESSMENT ANSWERS

1. C) To plan and manage the testing process
2. Answer: C) A high-level approach to conducting software testing
3. Answer: C) Unit testing
4. Answer: B) The highest-level components
5. Answer: A) Incremental integration testing
6. Answer: D) Reduced overhead of creating stubs and drivers
7. Answer: C) No new defects have been introduced by recent changes
8. Answer: C) Identifying show-stopper errors early in the testing process
9. Answer: B) To test individual units of code in isolation
10. Answer: C) Top-down integration testing

9. TERMINAL QUESTIONS

1. Explain the purpose of organizing software testing.
2. With a neat diagram explain the Software Testing Strategy.
3. Explicate testing strategic issues.
4. Define Unit Testing and what are its benefits. Explain the unit test environment.
5. Differentiate top-down and bottom-up integration testing.
6. Write a note on Regression testing and Smoke testing.

10. TERMINAL ANSWERS

1. Refer to section 10.2
2. Refer to Section 10.3
3. Refer to Section 10.3.1
4. Refer to Sections 10.4 and 10.4.1
5. Refer to Section 10.5
6. Refer to Sections 10.5.2 and 10.5.3