1. Basics of Software Testing

What is Testing ?

Testing is the process of evaluating a system or a component to determine its behavior, functionality, or performance in order to identify any defects, errors, or discrepancies. It involves executing a system or component under controlled conditions and observing its results to compare them with the expected outcomes. The main objective of testing is to ensure that the system or component meets the specified requirements, works as intended, and is reliable and robust.

Testing can be applied to various domains, including software development, hardware manufacturing, product quality assurance, and even scientific experiments. It encompasses a wide range of activities, such as planning and designing test cases, executing tests, analyzing results, and reporting any issues discovered. Testers use different techniques, methodologies, and tools to carry out testing effectively, depending on the nature and complexity of the system being tested.

The primary goals of testing include:

1. Finding defects: Testing aims to uncover bugs, errors, or flaws in the system or component to ensure that it functions correctly and delivers the expected results.

2. Verifying functionality: Testing verifies that the system or component behaves according to the defined specifications and requirements.

3. Enhancing quality: By identifying and addressing issues during the testing phase, the overall quality of the system or component can be improved.

4. Ensuring reliability: Testing helps ensure that the system or component functions reliably under normal and exceptional conditions, minimizing the chances of failures or malfunctions.

5. Mitigating risks: Testing reduces the risks associated with using a system or component by identifying and resolving potential problems before they impact users or critical operations.

6. Validating user expectations: Testing ensures that the system or component meets the end-users' expectations, delivering the intended features and functionality.

Principles of Software Testing

Software testing is a crucial process in the software development lifecycle that aims to identify defects, errors, or malfunctions in a software system. The principles of software testing provide a framework for performing effective and efficient testing. Here are some key principles of software testing:

1. Exhaustiveness: It is impossible to test all possible inputs and scenarios of a software system. However, testing should strive to be as comprehensive as possible within the given time and resource constraints.

2. Early Testing: Testing activities should start as early as possible in the software development lifecycle.

3. Defect Clustering: Studies have shown that defects tend to cluster around certain modules, features, or functionality.

4. Pesticide Paradox: Repetitive testing with the same set of test cases can lead to diminishing returns, as it tends to uncover fewer defects over time.

5. Testing is Context-Dependent: Testing strategies, techniques, and priorities can vary depending on the project's context, including the nature of the software, its complexity, the target audience, and the available resources.

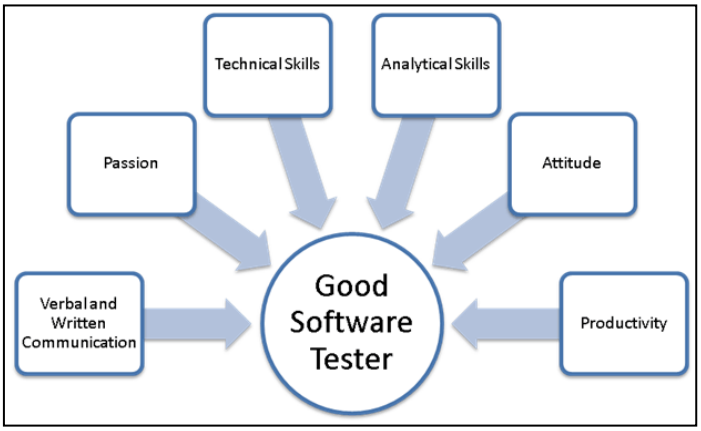
6. Testing is Risk-Driven: Testing efforts should be focused on mitigating the highest risks to the software's quality and functionality.

7. Documentation: Proper documentation of test plans, test cases, and test results is essential for effective testing.

8. Independence: To maintain objectivity and unbiased testing, the testing team should be independent of the development team.

9. Continuous Improvement: Testing is an iterative process, and continuous improvement is key to enhancing its effectiveness.

Skills of a Software Tester



- **Analytical skills**: A good software tester should have sharp analytical skills. Analytical skills will help break up a complex software system into smaller units to gain a better understanding and create test cases.
- **Communication skill**: A good software tester must have good verbal and written communication skill. Testing artifacts (like test cases/plans, test strategies, bug reports, etc.) created by the software tester should be easy to read and comprehend.
- **Time Management & Organization Skills:** Testing at times could be a demanding job especially during the release of code. A software tester must efficiently manage workload, have high productivity, exhibit optimal time management, and organization skills
- **Basic knowledge of Database/ SQL:** Software Systems have a large amount of data in the background. This data is stored in different types of databases like Oracle, MySQL, etc. in the backend.
- **Basic knowledge of Linux commands:** Most of the software applications like Web-Services, Databases, Application Servers are deployed on Linux machines.
- **Knowledge and hands-on experience of a Test Management Tool:** Test Management is an important aspect of Software testing. Without proper test management techniques, software testing process will fail.

Entry and Exit Criteria

Entry and exit criteria are important aspects of software testing that help define when testing activities should begin and end. They serve as guidelines and

checkpoints to ensure that testing is conducted in a structured and systematic manner. Here's an explanation of entry and exit criteria in software testing:

Entry Criteria : Entry criteria, also known as preconditions, are the conditions that must be met before testing can begin. These criteria ensure that the software and testing environment are ready for testing. The specific entry criteria may vary depending on the project and organization, but common examples include:

- **Availability of testable software:** The software build or version to be tested should be available and stable enough for testing to start.
- **Completion of unit testing:** Unit testing should be completed, and critical defects found during unit testing should be resolved before proceeding to higher-level testing.
- **Test environment readiness:** The required hardware, software, and test environment setup should be in place, including test servers, databases, network connectivity, and test data.
- **Test documentation readiness:** Test plans, test cases, and other testing artifacts should be prepared and reviewed to ensure they are comprehensive and accurate.
- **Test data availability:** Sufficient and representative test data should be available for testing different scenarios and functionalities.
- **Test resources allocation:** Sufficient test resources, including testers, test tools, and test infrastructure, should be allocated for conducting effective testing.

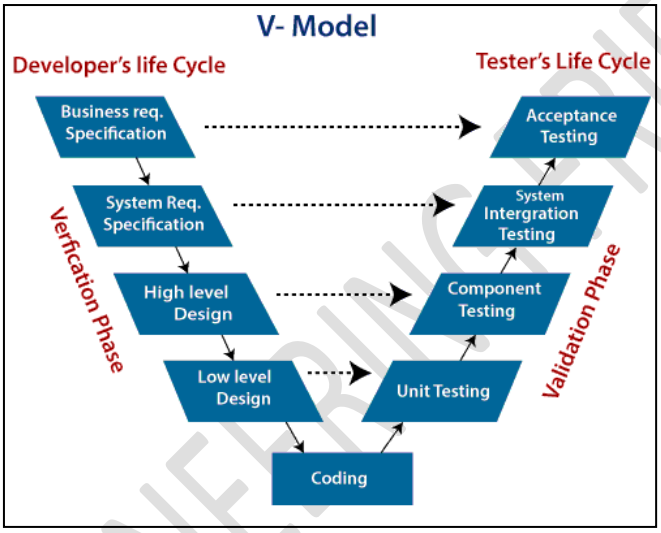
Exit Criteria : Exit criteria, also known as completion criteria or test completion criteria, are the conditions that must be satisfied to determine when testing activities can be concluded. They provide a basis for deciding whether

testing has achieved its objectives and whether the software is ready for release. The specific exit criteria may vary based on the project and organization, but common examples include:

- **Test coverage:** The defined test coverage goals, such as functional coverage, code coverage, or risk-based coverage, should be met to ensure that critical functionalities and areas have been adequately tested.
- **Defect resolution:** All critical defects should be fixed, and a predefined level of defect resolution, such as a specific defect closure rate or severity level, should be achieved.
- **Stability and reliability:** The software should demonstrate stability and reliability by meeting predefined stability metrics, such as a certain number of consecutive error-free test runs or acceptable system performance.
- **Test execution completion:** All planned test cases should be executed, and the execution status should be reviewed to ensure sufficient test coverage.
- **Test artifacts review:** Test documentation, including test plans, test cases, and test results, should be reviewed and approved to ensure accuracy and completeness.
- **Stakeholder acceptance:** The relevant stakeholders, such as product owners or project managers, should review the test results and provide their acceptance or sign-off on the testing activities.
- **Resource constraints:** The allocated time, budget, and resources for testing should be exhausted or reaching predefined limits.
- **Regulatory compliance**: If the software needs to comply with specific regulations or standards, the exit criteria may include successful completion of compliance testing and meeting the necessary requirements.

V Model

V-Model also referred to as the Verification and Validation Model. In this, each phase of SDLC must complete before the next phase starts. It follows a sequential design process same as the waterfall model. Testing of the device is planned in parallel with a corresponding stage of development.



Verification: It involves a static analysis method (review) done without executing code. It is the process of evaluation of the product development process to find whether specified requirements meet.

Validation: It involves dynamic analysis method (functional, non-functional), testing is done by executing code. Validation is the process to classify the software after the completion of the development process to determine whether the software meets the customer expectations and requirements.

There are the various phases of Verification Phase of V-model:

- **Business requirement analysis:** This is the first step where product requirements understood from the customer's side. This phase contains

detailed communication to understand customer's expectations and exact requirements.

- **System Design:** In this stage system engineers analyze and interpret the business of the proposed system by studying the user requirements document.
- **Architecture Design:** The baseline in selecting the architecture is that it should understand all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology detail, etc. The integration testing model is carried out in a particular phase.
- **Module Design:** In the module design phase, the system breaks down into small modules. The detailed design of the modules is specified, which is known as Low-Level Design
- **Coding Phase:** After designing, the coding phase is started. Based on the requirements, a suitable programming language is decided. There are some guidelines and standards for coding. Before checking in the repository, the final build is optimized for better performance, and the code goes through many code reviews to check the performance.

There are the various phases of Validation Phase of V-model:

- **Unit Testing:** In the V-Model, Unit Test Plans (UTPs) are developed during the module design phase. These UTPs are executed to eliminate errors at code level or unit level. A unit is the smallest entity which can independently exist, e.g., a program module. Unit testing verifies that the smallest entity can function correctly when isolated from the rest of the codes/ units.
- **Integration Testing:** Integration Test Plans are developed during the Architectural Design Phase. These tests verify that groups created and tested independently can coexist and communicate among themselves.

- **System Testing:** System Tests Plans are developed during System Design Phase. Unlike Unit and Integration Test Plans.
- **Acceptance Testing:** Acceptance testing is related to the business requirement analysis part. It includes testing the software product in user atmosphere.

When to use V-Model ?

- When the requirement is well defined and not ambiguous.
- The V-shaped model should be used for small to medium-sized projects where requirements are clearly defined and fixed.
- The V-shaped model should be chosen when sample technical resources are available with essential technical expertise.

Advantage (Pros) of V-Model:

1. Easy to Understand.
2. Testing Methods like planning, test designing happens well before coding.
3. This saves a lot of time. Hence a higher chance of success over the waterfall model.
4. Avoids the downward flow of the defects.
5. Works well for small plans where requirements are easily understood.

Disadvantage (Cons) of V-Model:

1. Very rigid and least flexible.
2. Not a good for a complex project.

3. Software is developed during the implementation stage, so no early prototypes of the software are produced.
4. If any changes happen in the midway, then the test documents along with the required documents, has to be updated.

Verification	Validation
Process of evaluating the product or system at various development stages to ensure that it complies with the specified requirements.	Process of evaluating the final product or system during or at the end of the development process to determine if it satisfies the user's intended use and needs.
Focuses on checking whether the software is built correctly according to the design and requirements.	Focuses on checking whether the software meets the user's expectations and requirements.
Performed through reviews, inspections, walkthroughs, and static analysis techniques.	Performed through testing, simulations, and user feedback.
Takes place before validation.	Takes place after verification and is a dynamic activity.
Ensures that the software meets the design specifications and requirements.	Ensures that the software meets the user's needs and expectations.
It is a process-oriented activity.	It is a product-oriented activity.
Typically involves internal stakeholders, such as developers, quality assurance teams, and project managers.	Typically involves external stakeholders, such as end-users, clients, and customers.
Aims to identify and fix defects and discrepancies early in the development lifecycle.	Aims to assess the software's overall suitability, usability, and fitness for the intended purpose.
Examples of verification activities include code reviews, inspections, unit testing, and static analysis.	Examples of validation activities include system testing, user acceptance testing, usability testing, and beta testing.

Methods of Testing

- **Static Testing**
- **Dynamic Testing**

Static Testing

Static Testing is a type of a Software Testing method which is performed to check the defects in software without actually executing the code of the software application. Whereas in Dynamic Testing checks, the code is executed to detect the defects. Static testing is performed in early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily. The errors that cannot be found using Dynamic Testing, can be easily found by Static Testing. **Static Testing Techniques:**
There are mainly two type techniques used in Static Testing:

1. **Review:** In static testing review is a process or technique that is performed to find the potential defects in the design of the software. It is process to detect and remove errors and defects in the different supporting documents like software requirements specifications. People examine the documents and sorted out errors, redundancies and ambiguities. Review is of four types:

- **Informal:** In informal review the creator of the documents put the contents in front of audience and everyone gives their opinion and thus defects are identified in the early stage.

- **Walkthrough:** It is basically performed by experienced person or expert to check the defects so that there might not be problem further in the development or testing phase.
- **Peer review:** Peer review means checking documents of one-another to detect and fix the defects. It is basically done in a team of colleagues.
- **Inspection:** Inspection is basically the verification of document the higher authority like the verification of software requirement specifications (SRS).

2. **Static Analysis:** Static Analysis includes the evaluation of the code quality that is written by developers. Different tools are used to do the analysis of the code and comparison of the same with the standard. It also helps in following identification of following defects:

(a) Unused variables

(b) Dead code

(c) Infinite loops

(d) Variable with undefined value

(e) Wrong syntax

Static Analysis is of three types:

- **Data Flow:** Data flow is related to the stream processing.
- **Control Flow:** Control flow is basically how the statements or instructions are executed.
- **Cyclomatic Complexity:** Cyclomatic complexity defines the number of independent paths in the control flow graph made from the code or flowchart so that minimum number of test cases can be designed for each independent path.

Dynamic Testing

Dynamic Testing is a type of Software Testing which is performed to analyze the dynamic behavior of the code. It includes the testing of the software for the input values and output values that are analyzed. Dynamic Testing is basically performed to describe the dynamic behavior of code. It refers to the observation of the physical response from the system to variables that are not constant and change with time.

To perform dynamic testing the software should be compiled and run. It includes working with the software by giving input values and checking if the output is as expected by executing particular test cases which can be done with either manually or with automation process.

In 2 V's i.e., Verification and Validation, Validation is Dynamic Testing.

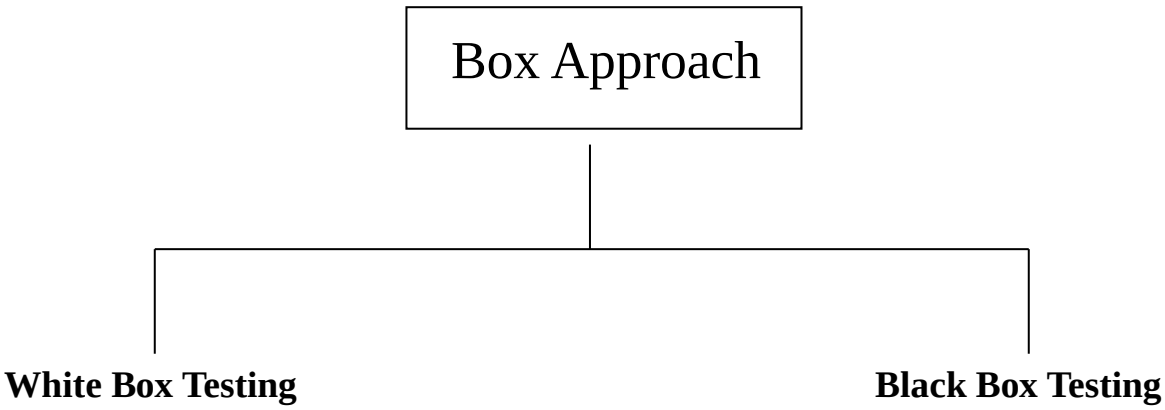
Levels of Dynamic Testing

There are various levels of Dynamic Testing. They are:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

There are several levels of dynamic testing that are commonly used in the software development process, including:

1. **Unit testing:** Unit testing is the process of testing individual software components or “units” of code to ensure that they are working as intended. Unit tests are typically small and focus on testing a specific feature or behavior of the software.
2. **Integration testing:** Integration testing is the process of testing how different components of the software work together. This level of testing typically involves testing the interactions between different units of code, and how they function when integrated into the overall system.
3. **System testing:** System testing is the process of testing the entire software system to ensure that it meets the specified requirements and is working as intended. This level of testing typically involves testing the software’s functionality, performance, and usability.
4. **Acceptance testing:** Acceptance testing is the final stage of dynamic testing, which is done to ensure that the software meets the needs of the end-users and is ready for release. This level of testing typically involves testing the software’s functionality and usability from the perspective of the end-user.
5. **Performance testing:** Performance testing is a type of dynamic testing that is focused on evaluating the performance of a software system under a specific workload. This can include testing how the system behaves under heavy loads, how it handles a large number of users, and how it responds to different inputs and conditions.
6. **Security testing:** Security testing is a type of dynamic testing that is focused on identifying and evaluating the security risks associated with a software system. This can include testing how the system responds to different types of security threats, such as hacking attempts, and evaluating the effectiveness of the system’s security features.



White Box Testing

White box testing, also known as clear box testing or structural testing, is a software testing technique that examines the internal structure and workings of the software application. In white box testing, the tester has knowledge of the internal code, architecture, and implementation details of the software being tested. The goal of white box testing is to assess the correctness of the internal logic, ensure that all code paths are executed, and validate the system against the specified requirements.

White box testing is particularly effective for detecting issues related to code-level defects, control flow errors, and logical flaws. By examining the internal structure, white box testing complements other testing techniques, such as black box testing, to provide a more comprehensive assessment of software quality.

1. Internal Structure Knowledge: In white box testing, the tester has access to the internal structure of the software application. This includes knowledge of the source code, design documents, algorithms, and system architecture.

Understanding the internal workings of the software enables the tester to design test cases that target specific code segments and ensure thorough coverage.

2. Test Coverage: White box testing aims to achieve high test coverage by exercising all possible code paths and decision points within the software. This involves designing test cases to execute different branches, loops, and conditionals to ensure that all logical scenarios are tested. Test coverage metrics such as statement coverage, branch coverage, and path coverage are used to assess the thoroughness of the testing.

3. Techniques: White box testing utilizes various techniques to design and execute test cases. Some commonly used techniques include:

- **Statement Coverage :** This technique ensures that every statement in the code is executed at least once during testing. It helps identify untested or unreachable code segments.
- **Branch Coverage :** Branch coverage aims to test all possible branches or decision points in the code. It verifies that each branch outcome has been evaluated.
- **Path Coverage :** Path coverage focuses on testing all possible paths through the code. It ensures that every possible combination of branches and loops is exercised.
- **Boundary Value Analysis :** This technique tests the boundary conditions of input variables to uncover defects that may occur at the edges of their valid ranges.
- **Equivalence Partitioning :** Equivalence partitioning divides the input domain into groups or partitions and selects representative test cases from each partition.

- **Code Reviews :** White box testing may involve code reviews and static analysis techniques to identify potential defects, coding standards violations, or performance bottlenecks.

Advantages :

White box testing offers several advantages, including:

- **Early defect detection:** By examining the internal structure of the software, white box testing helps identify defects early in the development process, reducing the cost and effort required for fixing them.
- **Thorough test coverage:** White box testing aims for high test coverage by exercising all code paths, ensuring a more comprehensive evaluation of the software's behavior.
- **Improved code quality:** White box testing helps in improving code quality by identifying and fixing logic errors, code inefficiencies, and potential vulnerabilities.
- **Design validation:** White box testing validates the internal logic and verifies that the software functions correctly according to the design and requirements.

Limitations :

White box testing also has some limitations, including:

- **Requires internal knowledge :** White box testing requires testers to have access to the internal code and implementation details, which may not always be available or practical.

- **Limited external perspective :** White box testing focuses on the internal structure, which may overlook certain external behaviors and user perspectives.
- **Time and resource-intensive :** Achieving high test coverage through white box testing can be time-consuming and resource-intensive, especially for complex systems.

Black Box Testing

Black box testing is a software testing technique that focuses on the external behavior and functionality of a software application without considering its internal structure or implementation details. In black box testing, the tester treats the software as a "black box" and tests it solely based on the input and output, without knowledge of its internal workings. The goal of black box testing is to validate the software against the specified requirements, uncover defects, and ensure that it meets the user's expectations.

Black box testing is a valuable technique for evaluating the functionality, usability, and overall quality of software applications. By treating the software as a black box, testers can validate its behavior without being influenced by internal complexities. Black box testing complements other testing techniques, such as white box testing, to provide a comprehensive evaluation of software systems.

1. External Perspective : Black box testing considers the software application as a black box, where the tester has no knowledge of its internal architecture,

design, or code. The focus is on understanding and evaluating the software's behavior from an external perspective.

2. Functional Testing : Black box testing primarily emphasizes the functional aspects of the software. It aims to verify that the software performs the expected functions, produces correct outputs, and responds correctly to various inputs and user interactions.

3. Test Design Techniques : Black box testing utilizes various techniques to design test cases based on the software's requirements and specifications. Some commonly used techniques include:

- **Equivalence Partitioning :** This technique divides the input domain into groups or partitions and selects representative test cases from each partition. It helps in identifying common behaviors and potential defects.
- **Boundary Value Analysis :** Boundary value analysis focuses on testing the boundaries or extreme values of input variables to identify defects that may occur at those boundaries.
- **Decision Table Testing :** Decision table testing involves creating a table with combinations of inputs and expected outputs to test different scenarios and combinations of conditions.
- **State Transition Testing :** This technique is used to test software applications that have different states and transitions between those states. It verifies that the software correctly handles state changes and transitions.
- **Error Guessing :** Error guessing is an informal technique where the tester relies on their experience and intuition to guess potential errors or defects that may exist in the software.
- **Testing Independence :** Black box testing can be performed by individuals who are not directly involved in the software development process. Testers

can focus on the software's functionality without being biased by its internal implementation.

Advantages

Black box testing offers several advantages, including:

- **Simulates real user scenarios:** Black box testing tests the software from the user's perspective, ensuring that it meets user expectations and behaves correctly in various scenarios.
- **Encourages thorough requirement validation:** Black box testing helps validate that the software meets the specified requirements and performs the intended functions.
- **Detects integration issues:** Black box testing can uncover defects and issues that arise due to interactions between different components or systems.
- **Enhances user experience**: Black box testing ensures that the software is user-friendly, intuitive, and provides a seamless user experience.

Limitations

Black box testing also has some limitations, including:

- **Limited coverage :** Since black box testing focuses solely on external behavior, it may not uncover certain types of defects or errors related to the internal structure or code.
- **Incomplete testing:** It is challenging to achieve complete coverage of all possible scenarios and inputs, especially in complex systems.
- **Lack of transparency :** Black box testing does not provide insights into the internal code or architecture, making it difficult to pinpoint the root cause of defects.

Assignment 1

1. Define software testing. List out principles of software testing.
2. Describe required skills of a software tester.
3. Define test case with its features.
4. Write down the test cases for facebook login.
5. Explain entry criteria and exit criteria.
6. Explain V model in software testing.
7. Difference between validation and verification.
8. Explain methods of testing.
9. Difference between static and dynamic testing.
10. Explain white box testing with its types.
11. Explain black box testing with its types.

Chapter 1 Checklist

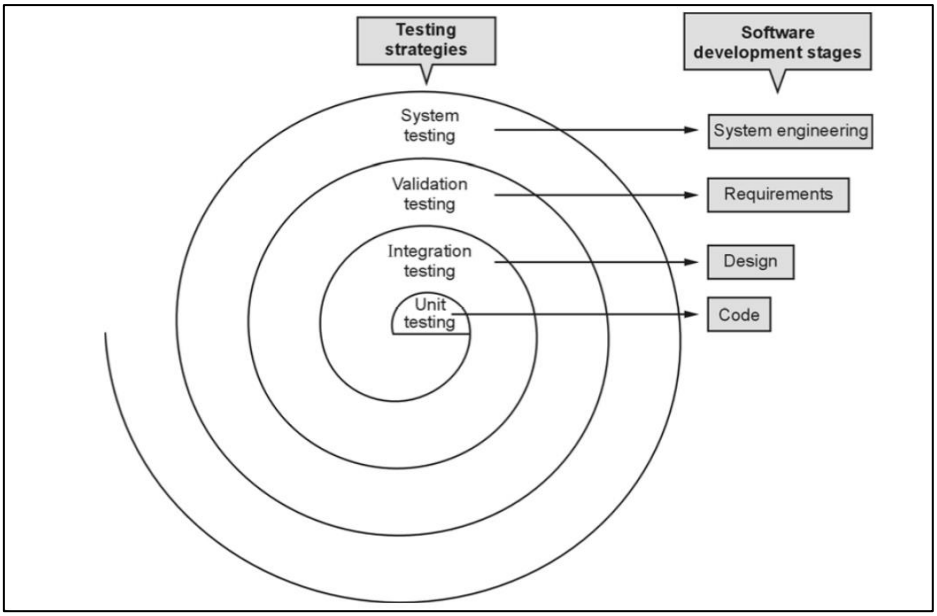
Sr. No.	Topics	Status (clear / doubt)
1.	Basic concepts of testing	
2.	Principles of testing	
3.	Static and dynamic testing	
4.	V model	
5.	Verification and validation	
6.	Quality assurance and quality control	
7.	White box testing	
8.	Black box testing	
9.	Test Case	

Doubts

2. Levels of Testing

In software development, testing is an essential process to ensure the quality, functionality, and reliability of a software system. Testing is typically performed at different levels, each with a specific focus and purpose.

Here are the common levels of testing:



1. Unit Testing: Unit testing involves testing individual components or units of code in isolation to ensure that they function correctly. It usually focuses on testing small sections of code, such as functions or methods, and is typically performed by developers using frameworks like JUnit or NUnit.

2. Integration Testing: Integration testing verifies the interaction and communication between different modules or components of a software system. It aims to uncover defects that may arise due to the integration of these components. Integration testing can be conducted at different levels, such as

module-level, where individual modules are tested together, or system-level, where the entire system is tested as a whole.

3. System Testing: System testing evaluates the behavior of the entire system as a whole, ensuring that all components work together correctly and meet the specified requirements. It focuses on testing the system against functional and non-functional requirements, including features, performance, security, and reliability. System testing is typically performed by dedicated testers or a testing team.

4. Acceptance Testing: Acceptance testing, also known as user acceptance testing (UAT), is conducted to determine whether the system meets the expectations and requirements of the end users or stakeholders. It is usually performed in a realistic environment that resembles the production environment and involves end users or business representatives. The primary goal is to validate that the software is ready for deployment and use.

Unit Testing

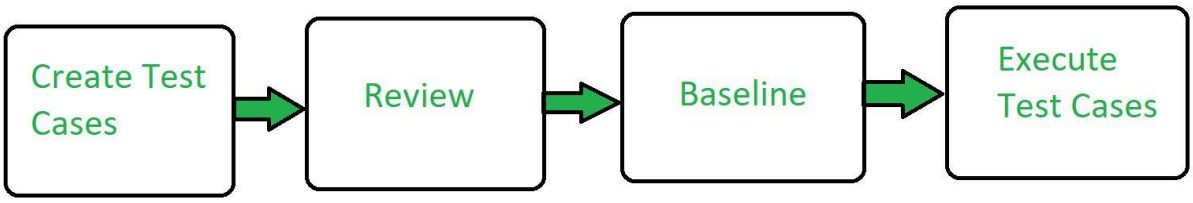
Unit testing is a type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements. Unit testing is typically performed by developers, and it is performed early in the development process before the code is integrated and tested as a whole system.

Unit tests are automated and are run each time the code is changed to ensure that new code does not break existing functionality. Unit tests are designed to validate the smallest possible unit of code, such as a function or a method, and test it in isolation from the rest of the system. This allows developers to quickly identify and fix any issues early in the development process, improving the overall quality of the software and reducing the time required for later testing.

The objective of Unit Testing is:

1. To isolate a section of code.
2. To verify the correctness of the code.
3. To test every function and procedure.
4. To fix bugs early in the development cycle and to save costs.
5. To help the developers to understand the code base and enable them to make changes quickly.
6. To help with code reuse.

Workflow of Unit

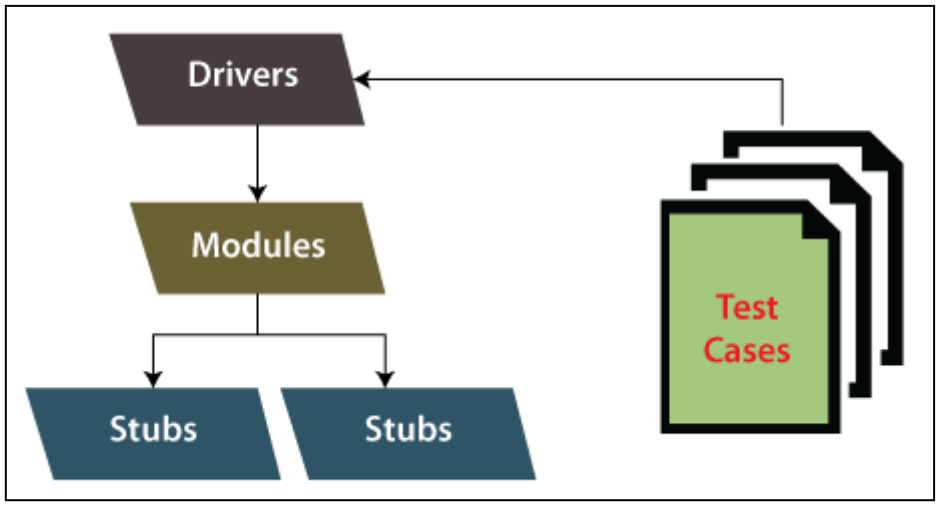


Unit Testing Techniques:

There are 3 types of Unit Testing Techniques. They are

1. **Black Box Testing:** This testing technique is used in covering the unit tests for input, user interface, and output parts.
2. **White Box Testing:** This technique is used in testing the functional behavior of the system by giving the input and checking the functionality output including the internal design structure and code of the modules.
3. **Gray Box Testing:** This technique is used in executing the relevant test cases, test methods, test functions, and analyzing the code performance for the modules.

Stub and Driver in Unit Testing



In unit testing, stubs and drivers are used to facilitate the testing of individual units of code in isolation. Let's explore the purpose and functionality of stubs and drivers in more detail:

1. Stubs:

Stubs are placeholder implementations of dependencies or collaborators that a unit of code being tested relies on. These dependencies may include other functions, methods, classes, or external services. Stubs are used to simulate the behavior and responses of these dependencies in order to isolate the unit under test.

- The main purpose of using stubs is to remove the dependencies on real or complex components during unit testing, ensuring that the unit being tested is truly isolated. By replacing these dependencies with stubs.
- you can control the inputs and outputs to mimic different scenarios and verify the behavior of the unit under various conditions.
- Stubs are typically simpler and more lightweight compared to the actual dependencies. They often provide predefined responses or behaviors that are suitable for the test cases being executed. Stubs can be manually created or generated using mocking frameworks or libraries.

2. Drivers:

Drivers, also known as test drivers or test harnesses, are used in unit testing to execute and control the flow of the unit being tested. Drivers act as a bridge between the test environment and the unit under test. They provide the necessary inputs to the unit and capture its outputs or behavior for verification.

- The purpose of a driver is to create the necessary context or environment required for the unit to execute correctly. It sets up any required data, initializes variables, or configures the system state to prepare the unit for testing.

- The driver then triggers the execution of the unit under test, passing the inputs and capturing the outputs or side effects produced by the unit.
- The use of drivers allows for the systematic execution of test cases and provides a structured approach to validate the behavior of the unit under different scenarios or inputs.

Integration Testing

Integration testing is a level of software testing that focuses on verifying the interactions and integration between different components or modules of a software system. The purpose of integration testing is to detect defects that may arise due to the combination and interaction of these components.

Integration testing ensures that the integrated system works as expected and meets the specified requirements, identifying any issues related to data flow, communication, interoperability, and interfaces between components. Here are some key aspects of integration testing:

1. Integration Points: Integration testing identifies the integration points between components, which can include function calls, data transfers, shared resources, APIs, or communication channels. These integration points represent areas where different components interact and exchange data or control.

2. Integration Strategies: There are several strategies for performing integration testing, including the top-down approach, bottom-up approach, and sandwich or hybrid approach.

- **Top-Down Approach:** In this approach, testing begins with the higher-level or main modules, and lower-level modules are gradually integrated and tested. Stubbed or simulated versions of lower-level modules may be used to simulate their behavior until they are fully developed.
- **Bottom-Up Approach:** In this approach, testing starts with the lower-level or foundational modules, and higher-level modules are incrementally integrated and tested. Drivers or harnesses may be used to simulate the behavior of higher-level modules until they are fully implemented.
- **Sandwich/Hybrid Approach:** This approach combines elements of both the top-down and bottom-up approaches. Testing starts from both ends of the system, focusing on the most critical or complex modules first, and gradually integrating and testing the remaining modules

Top-Down Integration

Top-down integration testing is an approach to integration testing where testing begins with the higher-level or main modules of a software system, and lower-level modules are gradually integrated and tested. This approach follows a step-by-step process, where testing progresses from the top of the system hierarchy towards the lower levels.

Here's how top-down integration testing typically works:

1. Test Main Module: In top-down integration testing, the testing process starts with the main module or the highest-level module in the system hierarchy. The

main module is the entry point or the central component that interacts with other modules in the system.

2. Substitute Lower-Level Modules: Since the lower-level modules are not yet available or fully implemented, stubs or simulated versions are used to substitute these modules. Stubs are simpler, lightweight versions of the lower-level modules that provide predefined responses or behaviors.

3. Execute Tests: Test cases are designed and executed to verify the behavior and interactions of the main module with the stubs or simulated versions of lower-level modules. These tests focus on the functionality, data flow, and communication between the main module and the stubs.

4. Gradual Integration: As lower-level modules become available or complete, they are integrated into the system one by one. The stubs are then replaced with the actual modules. The integration and testing process continues by progressively integrating more lower-level modules into the system.

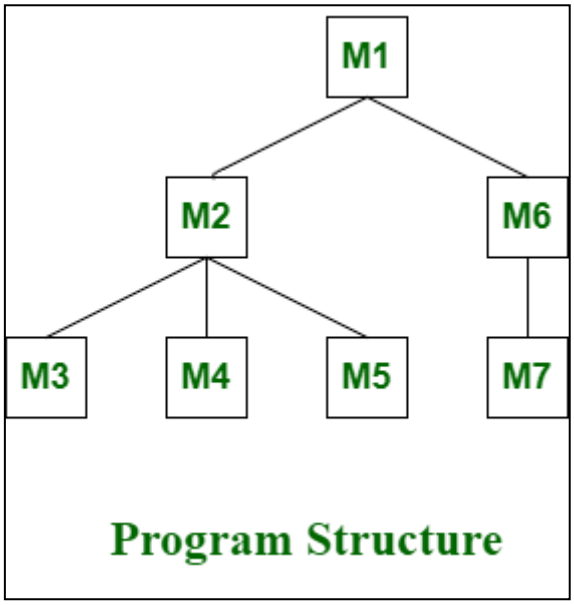
5. Iterative Testing: With each iteration of integration, new tests are created and executed to ensure the proper functioning and interactions of the newly integrated components. This iterative process continues until all modules are integrated and the entire system is tested.

6. Validation of Functionality: As the integration progresses, the focus of testing expands from the main module to the interactions between modules at various levels. The goal is to validate the overall functionality, data flow, and behavior of the integrated system.

7. Defect Identification and Resolution: As testing proceeds, any defects, inconsistencies, or issues in the interactions between modules are identified and logged. These issues are then addressed, and the necessary fixes or

modifications are made to ensure the proper integration and functioning of the system.

8. Completion of Integration: Once all modules are integrated and the entire system is tested, top-down integration testing is considered complete. The integrated system is expected to meet the functional and non-functional requirements defined for the software project.



In the top-down integration testing, if breadth-first approach is adopted, then we will integrate module M1 first, then M2, M6. Then we will integrate module M3, M4, M5, and at last M7.

Bottom-Up Integration

Bottom-up Testing is a type of incremental integration testing approach in which testing is done by integrating or joining two or more modules by moving upward from bottom to top through control flow of architecture structure. In these, low-level modules are tested first, and then high-level modules are

tested. This type of testing or approach is also known as inductive reasoning and is used as a synthesis synonym in many cases. Bottom-up testing is user-friendly testing and results in an increase in overall software development. This testing results in high success rates with long-lasting results.

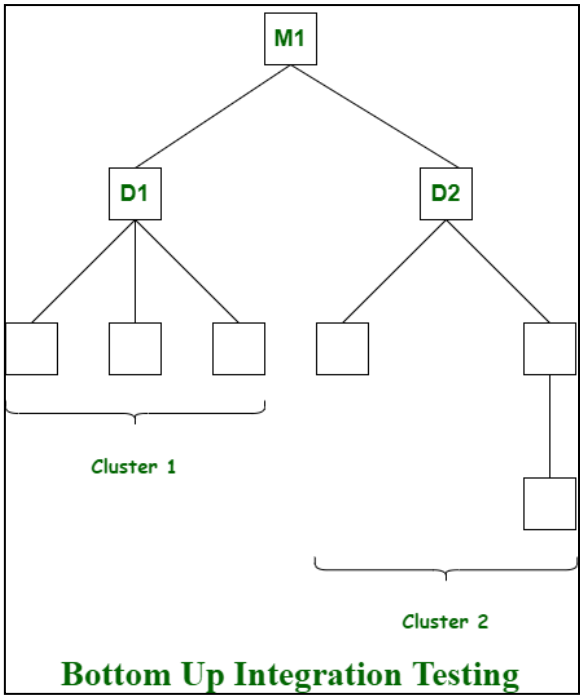
Processing :

Following are the steps that are needed to be followed during the processing :

1. Clusters are formed by merging or combining low-level modules or elements. These clusters are also known as builds that are responsible for performing the certain secondary or subsidiary function of a software.
2. It is important to write a control program for testing. These control programs are also known as drivers or high-level modules. It simply coordinates input and output of a test case.
3. Testing is done of entire build or cluster containing low-level modules.
4. At lastly, control program or drivers or high levels modules are removed and clusters are integrated by moving upward from bottom to top in program structure with help of control flow.

Example –

In the last, modules or components are combined together to form cluster 1 and cluster 2. After this, each cluster is tested with the help of a control program. The cluster is present below the high-level module or driver. After testing, driver is removed and clusters are combined and moved upwards with modules.



Bottom Up Integration Testing

Testing on Web Applications

- **Performance Testing**
- **Load Testing**
- **Stress Testing**
- **Security Testing**

Performance Testing

Performance Testing is a type of software testing that ensures software applications to perform properly under their expected workload. It is a testing technique carried out to determine system performance in terms of sensitivity, reactivity and stability under a particular workload.

Performance testing is a type of software testing that focuses on evaluating the performance and scalability of a system or application. The goal of performance testing is to identify bottlenecks, measure system performance under various loads and conditions, and ensure that the system can handle the expected number of users or transactions.

Load Testing

Load testing simulates a real-world load on the system to see how it performs under stress. It helps identify bottlenecks and determine the maximum number of users or transactions the system can handle.

Stress Testing

Stress testing is a type of load testing that tests the system’s ability to handle a high load above normal usage levels. It helps identify the breaking point of the system and any potential issues that may occur under heavy load conditions.

Security Testing

Security testing is an essential component of web application testing that focuses on identifying vulnerabilities and weaknesses in a web application's security measures. The purpose of security testing is to ensure that the application's data, functionality, and user interactions are protected from unauthorized access, malicious attacks, and potential breaches. It involves evaluating the application's security controls, configurations, and protocols to identify potential vulnerabilities and ensure compliance with security standards and best practices.

Security testing is an ongoing process, and it should be performed at various stages of the web application's development lifecycle. Regular security assessments, including periodic penetration testing, can help identify and mitigate security risks, ensuring the protection of sensitive data and the overall security posture of the web application.

Here are some key aspects and techniques involved in security testing for web applications:

1. Vulnerability Assessment:

Conduct a comprehensive assessment of the web application's vulnerabilities, including common security weaknesses such as injection attacks (e.g., SQL injection, XSS), cross-site scripting, cross-site request forgery, insecure direct object references, etc.

2. Authentication and Authorization Testing:

Verify the effectiveness and robustness of the web application's authentication mechanisms, such as username/password validation, session management, multi-factor authentication, and password recovery functionality.

3. Input Validation Testing:

Validate the handling of user inputs, ensuring that the application properly sanitizes and validates user inputs to prevent common security vulnerabilities like code injection attacks, buffer overflows, or command injection.

4. Security Configuration Testing:

Review and assess the configuration settings of the web application's underlying infrastructure, servers, frameworks, and databases for potential security weaknesses.

5. Session Management Testing:

Test the web application's session management mechanisms to ensure that session tokens or cookies are securely generated, transmitted, and stored.

6. Error Handling and Logging Testing:

Assess the web application's error handling and logging mechanisms to ensure that error messages do not reveal sensitive information and are properly logged for troubleshooting and security analysis.

7. Data Protection and Encryption Testing:

Evaluate the protection of sensitive data, such as passwords, personally identifiable information (PII), or financial data, by assessing the encryption

algorithms, key management, data transmission security, and secure storage mechanisms.

8. Security Headers and Secure Communication:

Verify the presence and effectiveness of security headers, such as Content Security Policy (CSP), Strict Transport Security (HSTS), and Cross-Origin Resource Sharing (CORS), to mitigate common security risks.

9. Business Logic Testing:

Assess the web application's business logic to identify any security vulnerabilities or weaknesses in the design and implementation of critical functionalities.

10. Compliance and Regulatory Testing:

Validate the web application's compliance with relevant security standards, regulations, and industry-specific requirements (e.g., PCI DSS, GDPR).

Acceptance Testing

Acceptance testing is a type of software testing performed to determine if a system or application meets the specified requirements and is acceptable for delivery to the end users or stakeholders. It focuses on validating the system's functionality, usability, reliability, and compliance with business requirements or user expectations. Acceptance testing is typically performed by the end users or subject matter experts (SMEs) in collaboration with the development team.

The ultimate goal of acceptance testing is to gain confidence in the system's readiness for deployment and ensure that it fulfills the intended purpose and meets the expectations of the end users or stakeholders. By conducting thorough acceptance testing, organizations can reduce the risk of delivering a system that does not align with the users' needs or the business requirements, ultimately improving customer satisfaction and the success of the software application.

Alpha & Beta Testing

Alpha and beta testing are two phases of user acceptance testing that are conducted to gather feedback and validate the software system before its final release. Let's understand each phase in detail:

1. Alpha Testing:

- Alpha testing is performed by a select group of users or internal stakeholders in a controlled environment under the supervision of the development team.
- It is usually conducted in the early stages of the software development process, often before the system is feature-complete.
- The purpose of alpha testing is to identify any defects, usability issues, or functional gaps in the system and provide feedback for improvement.
- Alpha testers may execute predefined test cases, explore the system, and perform typical user actions to simulate real-world usage scenarios.
- The development team closely collaborates with the alpha testers, gathering their feedback, addressing reported issues, and making necessary improvements.

- Alpha testing helps identify critical issues early in the development process and allows for iterative refinement of the system.

2. Beta Testing:

- Beta testing involves releasing the software to a larger group of external users or customers who represent the target user base.
- It is typically conducted after alpha testing and is closer to the final release of the software.
- Beta testers use the system in their own environments and provide feedback on its performance, usability, functionality, and any identified issues.
- Beta testing aims to gather real-world user feedback and validate the system's stability, compatibility, and overall user satisfaction.
- The development team may provide specific test scenarios or areas of focus for the beta testers, but they also have the flexibility to explore the system according to their needs.
- Beta testing helps uncover potential issues or usage patterns that were not identified during alpha testing and allows for final refinements and adjustments based on user feedback.
- The feedback collected during beta testing is valuable in making informed decisions regarding the software's readiness for the final release.

Assignment 2

1. List out levels of testing and explain any one of them.
2. Explain unit testing with the help of an example.
3. Define stub and driver.
4. Explain Integration testing with its types.
5. Explain acceptance testing.
6. Difference alpha and beta testing.
7. Explain testing on web applications.
8. Write a short note on : load testing and stress testing.

Chapter 2 Checklist

Sr. No.	Topics	Status (clear / doubt)
1.	Levels of testing	
2.	Unit testing	
3.	Integration testing	
4.	Acceptance testing	
5.	Testing on web applications	
6.	Security testing	
7.	Performance testing	
8.	Load testing	
9.	Stress testing	
10.	Stud and driver	
11.	Client server testing	
12.	GUI testing	

Doubts

3. Test Management

Test Management

Test management in software testing refers to the process of planning, organizing, controlling, and monitoring all activities and resources related to testing within a software development project. It involves the coordination and execution of various testing activities to ensure that the software meets the desired quality and functionality requirements.

Test management tools are often used to support these activities by providing features for test case management, test execution tracking, defect management, and reporting. These tools help streamline the testing process, improve efficiency, and enhance the overall quality of the software being tested.

Test management encompasses a range of tasks and responsibilities, including:

1. Test Planning: Defining the overall test strategy, objectives, and scope for the project. This involves identifying the testing goals, selecting appropriate test techniques and tools, and creating a detailed test plan.

2. Test Case Design: Creating test cases that cover different scenarios and test conditions to verify the functionality and behavior of the software. Test cases may include inputs, expected outputs, and steps to reproduce specific scenarios.

3. Test Environment Setup: Establishing the required test environment, including hardware, software, and network configurations, to mimic the production environment as closely as possible.

4. Test Execution: Carrying out the planned testing activities, executing test cases, and recording the test results. This involves both manual and automated testing techniques, as well as tracking defects and issues discovered during testing.

5. Defect Management: Managing and tracking the defects or issues identified during testing. This includes documenting the details of each defect, assigning severity and priority levels, and monitoring the progress of defect resolution.

6. Test Reporting: Generating and sharing reports that summarize the testing progress, including the number of test cases executed, passed, and failed, as well as any outstanding defects. These reports provide stakeholders with visibility into the quality of the software under test.

7. Test Coverage Analysis: Assessing the completeness of testing by analyzing the extent to which different aspects of the software have been tested. This helps identify any areas that require additional testing or improvement.

8. Test Team Collaboration: Facilitating effective communication and collaboration among the members of the testing team, as well as coordinating with other project stakeholders, such as developers, project managers, and business analysts.

Test Plan

A **test plan** is a document that outlines the objectives, scope, approach, and activities for testing a software application or system. It serves as a roadmap for the testing effort and provides guidance to the testing team, ensuring that testing is conducted systematically and efficiently.

The primary purpose of a test plan is to define the overall strategy and provide a clear direction for the testing process. It specifies what needs to be tested, how it will be tested, the resources required, and the timeline for testing activities. A well-defined test plan helps ensure that all necessary aspects of the software are thoroughly tested, and that potential risks and challenges are addressed.

A test plan typically includes the following key elements:

1. Introduction: Provides an overview of the software being tested and the purpose of the test plan.

2. Objectives: Clearly states the goals and objectives of the testing effort, such as ensuring the software meets quality standards, identifying defects, or validating specific requirements.

3. Scope: Defines the boundaries of the testing effort by specifying what is included and what is excluded from testing. It clarifies the features, modules, or components that will be tested.

4. Test Strategy: Outlines the overall approach and methodology for testing, including the test levels, test types, and test techniques to be employed.

5. Test Environment: Describes the hardware, software, and network configurations required for testing, ensuring that the testing environment closely resembles the production environment.

6. Test Deliverables: Identifies the specific documents, reports, or artifacts that will be produced during the testing process, such as test cases, test scripts, test data, and test logs.

7. Test Schedule: Provides a timeline for the testing activities, including milestones, start and end dates, and any dependencies or constraints that may impact the testing effort.

8. Resource Allocation: Specifies the resources needed for testing, including personnel, equipment, tools, and any external dependencies.

9. Test Execution: Describes how the test cases will be executed, including the test data, test environments, and any specific configurations or prerequisites.

10. Defect Management: Outlines the process for reporting, tracking, and managing defects discovered during testing, including the roles and responsibilities of the testing team.

11. Risk Management: Identifies potential risks and challenges that may impact the testing effort and provides strategies for mitigating and addressing those risks.

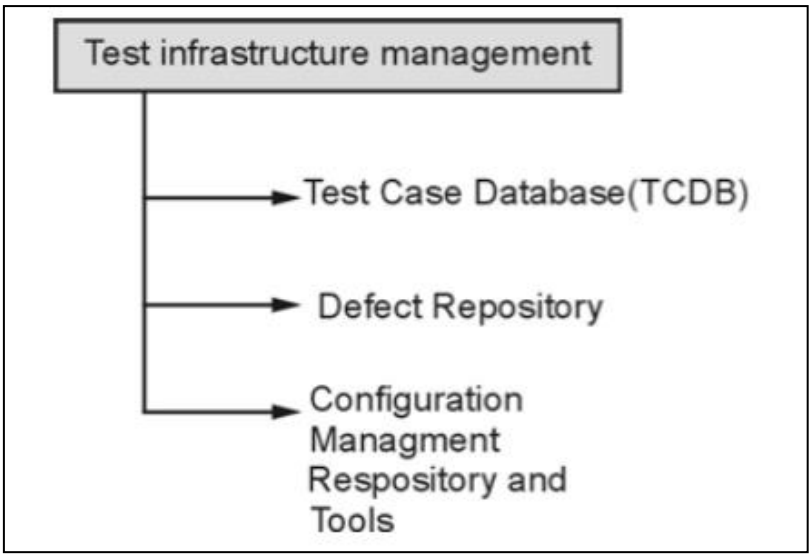
12. Test Reporting: Describes the reporting mechanisms, including the frequency and format of test progress reports, metrics, and communication channels.

13. Approval: Specifies the stakeholders or responsible parties who need to review and approve the test plan.

Test Infrastructure Management

Test infrastructure management refers to the process of planning, setting up, maintaining, and optimizing the hardware, software, and network resources necessary for testing activities within a software development project. It involves ensuring that the testing environment is properly configured, available, and ready to support the testing effort.

Effective test infrastructure management is crucial for ensuring a stable, reliable, and efficient testing environment. It enables testing teams to execute tests smoothly, identify defects accurately, and deliver high-quality software products. By providing a well-managed test infrastructure, organizations can improve the overall testing process and enhance the effectiveness and efficiency of their software development efforts.



Test Case Database -:

1) Test Case Database(TCDB) :

Test case database is a collection of the information about test cases in an organization. The contents of test case database are as given below -

Entity	Purpose	Attributes
Test case	Records all the information about the tests.	(1) Test case ID (2) Name of test case (3) Owner of test case (4) Status
Test case-product cross reference.	Provides mapping between test case and corresponding feature of the product.	(1) Test case ID (2) Module ID
Test case run history.	Provides the history of when test was running and what was the result.	(1) Test case ID (2) Run date (3) Time (4) Status of Run (Pass or Fail).
Test case - Defect cross reference.	Details of test cases introduced to test certain defects detected in product.	(1) Test case ID (2) Defect reference.

Defect Repository -:

A defect repository, also known as a bug tracking system or issue tracking system, is a centralized database or tool used to capture, track, and manage defects or issues identified during software testing and development. It serves as a repository for recording, monitoring, and resolving software defects throughout the entire software development lifecycle.

By using a defect repository, organizations can effectively manage and track defects, facilitate communication and collaboration among teams, prioritize and resolve issues efficiently, and ensure the overall quality of the software being developed.

Entity	Purpose	Attributes
Details of defect	Record all static information about test.	(1) Defect ID (2) Defect priority (3) Defect description (4) Affected product (5) Environment information. (6) Customer who encountered with the defect. (7) Date and time of defect occurrence.
Details of defect test	Details of test cases for a given defect. Cross reference with the TCDB.	(1) Test case ID (2) Defect ID
Fixing details	Details of the fixes for given defect.	(1) Defect ID (2) Fix details
Communication	Details of the communication that occurs among various stakeholder during defect management.	(1) Test case ID (2) Defect Reference. (3) Details of communication.

Software Configuration Management & Tools -:

Software Configuration Management (SCM) in software testing refers to the discipline and set of practices aimed at managing and controlling the changes made to software artifacts throughout the software development lifecycle. SCM encompasses the management of software configuration items (SCIs) and the associated processes, tools, and techniques for version control, change control, and release management.

Effective software configuration management in software testing helps ensure the integrity, traceability, and repeatability of software testing activities. It promotes collaboration, reduces risks associated with changes, and supports the delivery of high-quality software products.

Key aspects of software configuration management in software testing include:

1. Version Control: SCM involves the systematic management of versions and revisions of software artifacts, such as source code, test scripts, test data, and documentation. Version control ensures that changes are tracked, enables collaboration among team members, and allows for easy retrieval of specific versions when needed.

2. Change Control: SCM establishes processes and controls for managing changes to software artifacts. It involves defining change request procedures, assessing the impact of proposed changes, obtaining necessary approvals, and implementing changes in a controlled and traceable manner. Change control helps maintain stability, minimize risks, and ensure that changes do not adversely affect the quality of the software.

3. Baseline Management: SCM defines and manages baselines, which are stable and controlled snapshots of the software artifacts at specific points in time. Baselines provide a reference point for future changes, facilitate accurate configuration identification, and support reproducibility of software testing and releases.

4. Configuration Identification: SCM involves the identification and documentation of the configuration items (CIs) that comprise the software system.

5. Release Management: SCM encompasses the planning, coordination, and control of software releases. It involves defining release criteria, managing release schedules, coordinating activities between development and testing teams, and ensuring the proper packaging and deployment of software releases.

6. Build Management: SCM involves managing the process of building and assembling software components into executable releases or deliverables. This includes configuring build environments, defining build procedures, automating build processes, and tracking build versions.

7. Configuration Auditing and Verification: SCM includes conducting audits and verification activities to ensure that software configurations are consistent with the defined baselines, standards, and requirements.

8. SCM Tools and Infrastructure: SCM is supported by various tools and infrastructure that facilitate version control, change tracking, and release management.

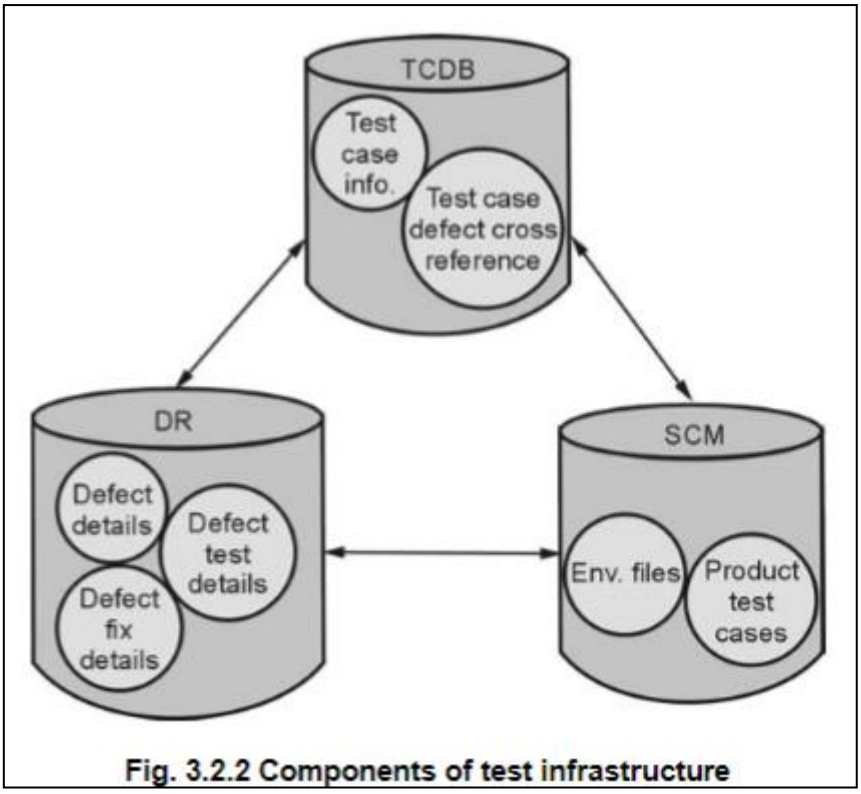


Fig. 3.2.2 Components of test infrastructure

Test Case