Q1. Attempt any EIGHT of the following (out of TEN):

a) What is software testing?

Ans: Software testing is the process of evaluating a software application or system to ensure that it meets specified requirements and functions correctly. It involves executing the software under controlled conditions and observing its behavior to detect errors, bugs, or defects. The primary goal of software testing is to identify and rectify any issues before the software is released to end-users, thereby enhancing its quality and reliability.

b) What is static testing?

Ans: Static testing is a type of software testing that does not involve the execution of the code. Instead, it involves reviewing and examining the software artifacts such as requirements, specifications, design documents, and code to identify defects, inconsistencies, and potential issues early in the development lifecycle. The main goal of static testing is to improve the quality of the software by detecting and fixing defects at an early stage, thereby reducing the cost and effort required for later stages of testing and development.

c) State the advantages of manual testing

Ans:

1. Flexibility: Manual testing allows testers to adapt and respond quickly to changes in requirements or project scope. Test cases can be easily modified or extended based on evolving needs.
2. Exploratory Testing: Manual testing enables testers to explore the software application in an ad-hoc manner, uncovering unexpected issues or user experience problems that might not be detected through automated tests.
3. User Perspective: Manual testers can simulate real-world user scenarios and interactions, providing valuable insights into the usability and user-friendliness of the software.
4. Early Detection of Usability Issues: Manual testing allows testers to assess the user interface (UI) and user experience (UX) early in the development process, identifying usability issues that automated tests may overlook.
5. Cost-Effectiveness for Small Projects: Manual testing can be more cost-effective for small-scale projects or projects with limited resources, as it does not require significant investment in test automation tools and infrastructure.
6. Human Judgment and Creativity: Manual testers can apply human judgment, intuition, and creativity to uncover complex or subtle defects that automated tests may miss. They can also devise innovative test scenarios to thoroughly validate the software's behavior.

d) What are formulae for calculating cyclomatic complexity?

Ans: The formulae for calculating cyclomatic complexity are:

McCabe's Cyclomatic Complexity (V(G)):

¥(V(G) = E - N + 2P ¥)

where:

- ¥(V(G) ¥) = Cyclomatic Complexity
- ¥(E ¥) = Number of edges in the control flow graph
- ¥(N ¥) = Number of nodes in the control flow graph
- ¥(P ¥) = Number of connected components (usually 1 unless dealing with multiple entry/exit points)

This formula is commonly used to calculate cyclomatic complexity based on the control flow graph of a software module.

e) What is grey box testing?

Ans: Grey box testing is a software testing technique that combines elements of both black box and white box testing. In grey box testing, testers have partial knowledge of the internal workings of the software system, such as access to the code structure, database schema, or architecture, but they do not have full visibility or control over all aspects. This approach allows testers to design test cases based on a combination of understanding the internal logic of the system and assessing its functionality from an end-user perspective. Grey box testing aims to uncover defects, vulnerabilities, and performance issues while considering both the system's internal behavior and its external functionality.

f) Define Validation Testing

Ans: Validation testing is a software testing process that evaluates whether a software system or application meets the requirements and expectations of the end-users and stakeholders. It focuses on verifying that the software product satisfies the intended use and purpose in its specific environment. Validation testing ensures that the software delivers the right functionality, features, and performance as per the user's needs and expectations. This testing phase typically occurs after the completion of development and prior to the software's release to the end-users.

g) What is debugging?

Ans: Debugging is the process of identifying, analyzing, and fixing errors, bugs, or defects in a software application or system. It involves tracing and diagnosing the root cause of unexpected behavior or failures within the codebase. Debugging techniques may include using software tools, such as debuggers, to inspect variables, step through code execution, and pinpoint the location of errors. The primary goal of debugging is to ensure the proper functioning and reliability of the software by eliminating issues that could impact its performance or usability.

h) Explain terms – Error, Fault and Failure

Ans:

1. Error: In the context of software testing, an error refers to a human action that produces an incorrect result. It can occur at any stage of the software development process, from requirements gathering to coding and implementation. Errors are unintentional mistakes made by developers, designers, or users that lead to deviations from the expected behavior of the software.
2. Fault: A fault, also known as a defect or bug, is a flaw or imperfection in the software's code or design that causes it to behave incorrectly or produce incorrect results when executed. Faults are typically the result of errors introduced during the development process. They can manifest as syntax errors, logical errors, or design flaws in the software.
3. Failure: A failure occurs when the software behaves incorrectly or does not meet the specified requirements or expectations during execution. It is the observable manifestation of a fault when the software deviates from its intended behavior. Failures can result from faults in the software's code, design, or environment, and they are typically detected during testing or when the software is in use.

i) Define regression testing

Ans: Regression testing is a software testing technique used to ensure that recent code changes or modifications to a software application do not adversely affect the existing functionality of the system. It involves retesting the previously tested functionalities along with the newly added or modified features to verify that the changes have not introduced any unintended side effects or regressions. The primary goal of regression testing is to maintain the stability and integrity of the software by identifying and fixing defects that may have been introduced during the development or maintenance process.

j) What is software metrics?

Ans: Software metrics are quantitative measures used to assess various attributes of a software product or process. These measures provide objective data that can be used to evaluate and improve the quality, performance, and efficiency of software development and maintenance activities. Software metrics encompass a wide range of parameters, including code complexity, defect density, code coverage, development effort, and software reliability, among others. By collecting and analyzing software metrics, organizations can gain insights into the effectiveness of their software development processes, identify areas for improvement, and make data-driven decisions to enhance the overall quality and performance of their software products.

Q2. Attempt any FOUR of the following(out of FIVE):

a) What is difference between Verification and Validation?

Ans: Certainly! Below is a comprehensive explanation of the difference between Verification and Validation, fulfilling the requirement of two A4 size pages.

Verification and Validation (V&V) are two essential processes in software engineering aimed at ensuring the quality and correctness of software products. While both are critical aspects of the software development lifecycle, they serve distinct purposes and involve different approaches. This essay will delve into the differences between verification and validation, highlighting their definitions, objectives, processes, and key distinctions.

1. Definitions:

Verification refers to the process of evaluating software artifacts to ensure that they adhere to specified requirements and standards. It focuses on determining whether the software is being developed correctly. In other words, verification answers the question: "Are we building the product right?"

Validation, on the other hand, involves evaluating the software to ensure that it meets the intended needs and expectations of the users. It focuses on determining whether the right product is being built. Validation addresses the question: "Are we building the right product?"

2. Objectives:

The primary objective of verification is to confirm that the software product adheres to its predefined specifications, design, and development standards. It involves activities such as code reviews, inspections, walkthroughs, and static analysis to detect defects early in the development process. Verification ensures that the software is developed according to the documented requirements and standards, minimizing the likelihood of errors in subsequent stages.

In contrast, the main objective of validation is to ensure that the software meets the user's needs and expectations in its specific operational environment. Validation involves dynamic testing techniques such as functional testing, system testing, acceptance testing, and usability testing to assess whether the software fulfills its intended purpose and delivers the desired functionality and performance to end-users. Validation verifies that the software satisfies the customer's requirements and provides value in real-world scenarios.

3. Processes:

Verification primarily focuses on examining the software artifacts, including requirements specifications, design documents, code, and test cases, to identify discrepancies, inconsistencies, or deviations from the defined standards. It involves a systematic review and analysis of the software documentation and artifacts to ensure completeness, correctness, and consistency. Verification activities are typically performed throughout the development lifecycle and are integrated into the various phases of software development, such as requirements analysis, design, implementation, and testing.

In contrast, validation entails executing the software and evaluating its behavior and performance against the user's expectations and acceptance criteria. It involves dynamic testing activities that simulate real-world usage scenarios to validate the software's functionality, usability, reliability, and performance. Validation activities are typically conducted towards the end of the development lifecycle, once the software has been developed or implemented, to ensure that it meets the customer's requirements and is ready for deployment.

4. Key Distinctions:

The key distinction between verification and validation lies in their focus and objectives. Verification focuses on verifying that the software is being developed correctly according to predefined specifications and standards, while validation focuses on validating whether the right product is being developed to meet the user's needs and expectations.

Another distinction is in the timing of the activities. Verification activities are typically performed throughout the software development lifecycle, starting from the requirements phase, whereas validation activities are often conducted towards the end of the development lifecycle, closer to the deployment phase, to ensure that the software meets the customer's requirements before release.

Furthermore, verification involves static testing techniques that examine software artifacts without executing the code, whereas validation involves dynamic testing techniques that involve executing the software and evaluating its behavior in real-world scenarios.

While verification and validation are both essential processes in software engineering, they serve distinct purposes and involve different approaches. Verification focuses on ensuring that the software is developed correctly, adhering to predefined specifications and standards, while validation focuses on validating whether the software meets the user's needs and expectations in its intended operational environment.

b) Explain software testing lifecycle with diagram

Ans:

$$\begin{tikzcd}
\text{Requirement} \arrow[d] \\
\text{Analysis} \arrow[d] \\
\text{Test Planning} \arrow[d] \\
\text{Test Case Development} \arrow[d] \\
\text{Test Environment Setup} \arrow[d] \\
\text{Test Execution} \arrow[d] \\
\text{Defect Tracking \& Mgmt} \arrow[d] \\
\text{Test Reporting} \arrow[d] \\
\text{Test Closure}
\end{tikzcd}$$

The Software Testing Lifecycle (STLC) is a systematic approach to software testing that encompasses various phases to ensure the quality and reliability of a software product. Below is an explanation of the STLC along with a diagram illustrating its phases.

1. Requirement Analysis:

- In this phase, testers collaborate with stakeholders to understand the project requirements, including functional and non-functional aspects.
- Testers analyze requirements documents, user stories, and other relevant documentation to identify testable components and define test objectives.

2. Test Planning:

- Test planning involves creating a comprehensive test plan that outlines the testing approach, scope, objectives, resources, schedule, and deliverables.
- Testers define test strategies, select appropriate testing techniques, and allocate resources based on project requirements and constraints.

3. Test Case Development:

- Test case development entails designing and documenting test cases based on the defined requirements and test objectives.
- Testers create test scenarios, test scripts, and test data to ensure thorough coverage of the application's functionalities and use cases.

4. Test Environment Setup:

- Test environment setup involves configuring the necessary hardware, software, and infrastructure required to execute the tests effectively.

- Testers set up test beds, install the application under test (AUT), and deploy supporting tools and frameworks to create a stable testing environment.

5. Test Execution:

   - Test execution is the phase where testers execute the prepared test cases against the application under test (AUT).

   - Testers run test scripts, input test data, and observe the application's behavior to identify defects, anomalies, or deviations from expected results.

6. Defect Tracking and Management:

   - Defect tracking and management involve documenting, prioritizing, and tracking defects found during the testing process.

   - Testers log defects in a defect tracking system, provide detailed descriptions, assign severity and priority levels, and collaborate with development teams to resolve issues.

7. Test Reporting:

   - Test reporting encompasses generating and communicating test reports to stakeholders, project managers, and other relevant parties.

   - Testers compile test results, metrics, and insights into comprehensive reports that provide visibility into the testing progress, quality of the software, and any associated risks.

8. Test Closure:

- Test closure involves formalizing the conclusion of the testing activities and obtaining stakeholders' acceptance of the testing outcomes.

- Testers evaluate the testing process, gather lessons learned, archive test artifacts, and prepare for future testing efforts or releases.

This diagram illustrates the sequential flow of activities in the Software Testing Lifecycle (STLC), starting from Requirement Analysis and concluding with Test Closure. Each phase in the STLC contributes to ensuring the quality and reliability of the software product through systematic testing processes and activities.

c) Explain boundary value analysis in detail.

Ans: Boundary Value Analysis (BVA) is a software testing technique used to identify errors at boundaries rather than finding them in the center of input domains. It is based on the principle that errors often occur at the edges or boundaries of input ranges rather than within the range itself. Here's a detailed explanation of Boundary Value Analysis:

1. Definition: Boundary Value Analysis is a black-box testing technique that focuses on testing boundary conditions of input ranges rather than testing at random or arbitrary points within the range. It involves selecting input values that lie at the boundaries, just above and just below the specified limits.

2. Objective: The primary objective of Boundary Value Analysis is to identify defects or errors that may arise due to incorrect handling of boundary conditions in the software under test. By testing at the

boundaries, testers aim to uncover boundary-related issues such as off-by-one errors, boundary violations, and data truncation.

3. Process:

Identify Input Domain: First, testers identify the input domain or range for each input parameter or variable in the software.

Select Boundary Values: Next, testers select input values that lie at the boundaries of the input domain. For numeric ranges, these values typically include the minimum value, maximum value, and values just above and just below the boundaries.

Design Test Cases: Testers design test cases using the selected boundary values for each input parameter. For example, if a parameter accepts values from 1 to 10, the boundary values for testing would be 0, 1, 2, 9, 10, and 11.

Execute Test Cases: Test cases are executed using the chosen boundary values, and the software's behavior is observed. Testers verify whether the software handles boundary conditions correctly and produces the expected outcomes.

Analyze Results: Test results are analyzed to identify any defects or anomalies observed at the boundaries. Defects found during boundary value analysis are logged, prioritized, and reported for resolution.

4. Benefits:

Efficiency: Boundary Value Analysis helps in efficient test case design by focusing on critical input values that are likely to reveal defects.

Thoroughness: By testing at the boundaries, this technique provides thorough coverage of edge cases and boundary conditions, increasing the likelihood of detecting errors.

Early Defect Detection: Boundary value analysis enables early detection of defects related to boundary conditions, allowing for timely resolution and reducing the risk of defects slipping into later stages of development or production.

Risk Mitigation: By targeting potential areas of risk, such as boundary conditions, BVA helps mitigate the risk of software failures and enhances the overall reliability and quality of the software product.

In summary, Boundary Value Analysis is a systematic testing technique that focuses on testing input boundaries to identify defects related to boundary conditions. By selecting input values at the edges of input ranges, testers can efficiently uncover errors and improve the robustness of the software under test.

d) Explain acceptance testing in detail.

Ans: Acceptance Testing is a critical phase in the software testing lifecycle that evaluates whether a software application or system meets the business requirements and is ready for deployment. This testing phase involves executing test scenarios and validating the software's functionality against user expectations and acceptance criteria. Below is a comprehensive explanation of acceptance testing:

1. Definition: Acceptance Testing is a formal testing process conducted to determine whether a software system satisfies the acceptance criteria

and meets the stakeholders' requirements. It verifies whether the software fulfills its intended purpose and is acceptable for delivery to the end-users. Acceptance testing is typically performed by end-users, business stakeholders, or quality assurance teams representing the users' interests.

2. Types of Acceptance Testing:

a. User Acceptance Testing (UAT): User Acceptance Testing involves testing the software from an end-user's perspective to ensure that it meets the user's needs, expectations, and business objectives. End-users or subject matter experts execute test cases in a real-world environment to validate the software's usability, functionality, and performance. UAT verifies that the software aligns with the user's workflow and delivers the desired outcomes.

b. Business Acceptance Testing (BAT): Business Acceptance Testing focuses on validating the software's alignment with the organization's business processes, policies, and objectives. It ensures that the software supports business operations, complies with regulatory requirements, and meets the strategic goals of the organization. BAT evaluates the software's effectiveness in enabling business activities and driving business value.

c. Operational Acceptance Testing (OAT): Operational Acceptance Testing verifies whether the software meets the operational requirements and is ready for deployment in the production environment. It assesses aspects such as installation, configuration, performance, scalability, reliability, and maintainability of the software in the target production environment. OAT ensures that the software can be effectively deployed, operated, and maintained by the operations team.

3. Process:

a. Define Acceptance Criteria: The first step in acceptance testing is to define clear and measurable acceptance criteria based on the requirements and expectations of the stakeholders. Acceptance criteria outline the specific conditions that must be met for the software to be accepted.

b. Develop Test Scenarios: Test scenarios are developed based on the acceptance criteria to simulate real-world usage scenarios and business workflows. Test cases are designed to validate various aspects of the software, including functionality, usability, performance, and compliance.

c. Execute Test Cases: Testers or end-users execute the test scenarios according to the predefined acceptance criteria. They interact with the software, input data, and perform operations to validate its behavior and functionality. Test results are recorded, and any deviations from the expected outcomes are identified and documented.

d. Verify Acceptance Criteria: Test results are compared against the acceptance criteria to determine whether the software meets the specified requirements. Any discrepancies or issues identified during testing are analyzed, reported, and addressed by the development team.

e. Obtain Stakeholder Approval: Once the software successfully passes acceptance testing and meets the acceptance criteria, stakeholders review the test results and provide formal approval for the software's release or deployment. Stakeholder approval signifies that the software is ready for production use.

4. Benefits:

a. Customer Satisfaction: Acceptance testing ensures that the software meets the user's needs and expectations, leading to increased customer satisfaction and confidence in the software product.

b. Risk Mitigation: By validating the software against predefined acceptance criteria, acceptance testing helps mitigate the risk of defects, errors, or failures in the production environment.

c. Enhanced Quality: Acceptance testing contributes to the overall quality and reliability of the software by identifying and resolving issues related to functionality, usability, and performance.

d. Alignment with Business Goals: Acceptance testing ensures that the software aligns with the organization's business goals, processes, and objectives, driving value and effectiveness in meeting business needs.

Acceptance Testing is a crucial phase in the software testing lifecycle that verifies whether the software meets the user's requirements and is ready for deployment. By validating the software against acceptance criteria, acceptance testing ensures customer satisfaction, mitigates risks, enhances quality, and aligns with business goals.

e) Explain test case design along with example.

Ans: Test Case Design:

Test case design is the process of creating detailed test cases that specify inputs, actions, and expected outcomes to validate the functionality of a software application or system. It involves identifying test scenarios, defining test objectives, selecting test conditions, and designing test steps to ensure thorough coverage of the software under test. Below is an explanation of test case design along with an example:

1. Identification of Test Scenarios:

- The first step in test case design is to identify test scenarios based on the software requirements, use cases, and functional specifications. Test scenarios represent distinct functionalities, features, or user interactions that need to be tested.

2. Definition of Test Objectives:

- Once the test scenarios are identified, test objectives are defined to outline the goals and purposes of each test case. Test objectives specify what needs to be verified or validated through the test case.

3. Selection of Test Conditions:

- Test conditions are selected based on the test objectives and requirements. Test conditions represent specific situations, inputs, or states that need to be tested to ensure the correctness and robustness of the software.

4. Design of Test Steps:

- Test steps are designed to describe the sequence of actions that testers need to perform to execute the test case. Test steps include instructions for setting up preconditions, executing actions, inputting data, and verifying outcomes.

Example of Test Case Design:

Test Case Title: Login Functionality Verification

Test Objective: To verify that users can successfully log in to the application using valid credentials.

Test Conditions:

- Valid username and password provided.
- Invalid username with a valid password provided.
- Valid username with an invalid password provided.
- Username field left blank.
- Password field left blank.

Test Steps:

1. Launch the application and navigate to the login page.
2. Enter a valid username and password.
3. Click on the 'Login' button.
4. Verify that the user is successfully logged in and redirected to the home page.
5. Log out of the application.

6. Repeat steps 1-3 with an invalid username and a valid password.

7. Verify that the appropriate error message is displayed, indicating invalid credentials.

8. Repeat steps 1-3 with a valid username and an invalid password.

9. Verify that the appropriate error message is displayed, indicating invalid credentials.

10. Repeat steps 1-3 with the username field left blank.

11. Verify that the appropriate error message is displayed, indicating the required field.

12. Repeat steps 1-3 with the password field left blank.

13. Verify that the appropriate error message is displayed, indicating the required field.

Expected Outcomes:

- Test steps 4, 7, and 9: Successful login with valid credentials.

- Test steps 11 and 13: Display of error message indicating the required field.

- Test steps 6 and 8: Display of error message indicating invalid credentials.

Conclusion:

Test case design involves systematically creating detailed test cases to validate the functionality of a software application or system. By following a structured approach and designing comprehensive test cases, testers can ensure thorough coverage and effective validation of the software under test.

This example of test case design demonstrates how test objectives, conditions, steps, and expected outcomes are defined for verifying the login functionality of an application. Each test step describes the actions to be performed, inputs to be provided, and expected outcomes to be verified, ensuring comprehensive testing coverage.

Q3) Attempt any FOUR of the following(out of FIVE):

a) Explain any four testing principle In detail.

Ans:

1. Early Testing: Early testing is the principle that testing activities should begin as early as possible in the software development lifecycle. This principle emphasizes the importance of identifying defects and issues early in the process when they are less costly and easier to address. By starting testing activities in the early stages, such as during requirements analysis or design, defects can be detected and resolved before they propagate further downstream. Early testing helps in improving the overall quality of the software, reducing rework, and minimizing the risk of defects slipping into later stages of development.

2. Testing Shows Presence of Defects: This principle states that testing can never prove the absence of defects but can only demonstrate their presence. No matter how rigorously a software application is tested, it is impossible to guarantee that it is defect-free. Testing provides information about the quality and reliability of the software by

identifying defects and anomalies in its behavior. However, the absence of detected defects does not imply that the software is completely error-free. Therefore, testing efforts should focus on identifying and mitigating risks associated with potential defects rather than striving for absolute perfection.

3. Exhaustive Testing is Impossible: This principle acknowledges that it is not feasible to test every possible input, condition, or scenario within a software application. The number of possible combinations of inputs, states, and interactions in a complex software system is virtually infinite, making exhaustive testing impractical and economically unfeasible. Instead, testing efforts should be focused on prioritizing test cases based on risk, probability of occurrence, and criticality. Testers aim to achieve sufficient test coverage to mitigate the most significant risks and ensure the software's reliability within the available time and resources.

4. Defect Clustering: Defect clustering, also known as the Pareto Principle or the 80/20 rule, suggests that a relatively small number of modules or components in a software system typically contain the majority of defects. This principle implies that defects tend to cluster around certain areas of the software, such as frequently used features, complex functionalities, or modules with high complexity. Therefore, testers can prioritize testing efforts by focusing on these critical areas to identify and address a significant portion of defects. Defect clustering highlights the importance of risk-based testing and targeted test case design to maximize the effectiveness of testing efforts.

b) Explain white box testing and its techniques.

Ans: White box testing, also known as structural or glass box testing, is a software testing technique that examines the internal structure, design, and implementation of a software application. Unlike black box testing, which focuses on testing the software's functionality without considering its internal workings, white box testing is based on an understanding of the underlying code and logic. It involves testing individual components, modules, or units of the software to ensure that they function correctly and fulfill their intended purpose. White box testing techniques are used to validate the correctness of the software's internal logic, paths, and algorithms.

Techniques of White Box Testing:

1. Statement Coverage (or Line Coverage):

- Statement coverage is a white box testing technique that aims to ensure that every executable statement in the code is executed at least once during testing. Test cases are designed to exercise each line of code to verify its functionality and identify any unreachable or dead code. Statement coverage helps in identifying areas of the code that have not been executed, indicating potential gaps in the testing coverage.

2. Branch Coverage (or Decision Coverage):

- Branch coverage is a white box testing technique that focuses on testing all possible branches or decision points within the code. It aims to ensure that every possible decision outcome, such as true or false branches in conditional statements, is tested at least once. Test cases are designed to exercise each branch of the code to verify its correctness and identify any logic errors or inconsistencies. Branch coverage helps in detecting missing or incorrect branching conditions that could lead to unexpected behavior in the software.

3. Path Coverage:

- Path coverage is a white box testing technique that aims to test every possible path or route through the code. It involves identifying all possible combinations of statements and branches within the code and designing test cases to traverse each path. Path coverage provides a more thorough assessment of the software's logic and control flow, helping in identifying complex interactions and dependencies between different parts of the code. However, achieving complete path coverage may be impractical for large or complex software systems due to the exponential number of possible paths.

4. Condition Coverage:

- Condition coverage is a white box testing technique that focuses on testing all possible conditions within the code, such as Boolean expressions and loop conditions. It ensures that every condition evaluates to both true and false during testing. Test cases are designed to exercise each condition independently to verify its behavior under different circumstances. Condition coverage helps in identifying errors related to conditional logic, such as missing or incorrect conditions, boundary conditions, and edge cases.

Conclusion

White box testing techniques are essential for verifying the internal logic, paths, and control flow of a software application. By examining the code's structure and behavior, white box testing helps in identifying defects, errors, and vulnerabilities that may not be detected through black box testing alone. Statement coverage, branch coverage, path coverage, and condition coverage are some of the key techniques used in white box testing to ensure thorough testing and validation of the software's internal components. These techniques complement black box testing approaches, providing a comprehensive assessment of the software's quality and reliability.

c) Explain sandwich and Big-Bang approach of integration testing.

Ans: Sandwich Approach and Big-Bang Approach in Integration Testing:

Integration testing is a crucial phase in the software testing process where individual software modules are combined and tested as a group to ensure that they function together correctly. Two common approaches to integration testing are the sandwich approach and the Big-Bang approach. Below, I will explain each approach in detail:

1. Sandwich Approach:

In the sandwich approach to integration testing, also known as incremental or bottom-up integration testing, software modules are integrated and tested incrementally in a sequential manner, starting from the lowest-level modules and gradually moving towards higher-level modules or subsystems. The approach derives its name from the incremental layering of modules, resembling layers of a sandwich being stacked on top of each other.

Explanation:

- Incremental Integration: Modules are integrated and tested one by one, starting with the lowest-level or most fundamental modules. Once a module is integrated, it is tested in conjunction with previously integrated modules to verify its interactions and dependencies.

- Incremental Builds: As more modules are integrated, the software system grows incrementally, and the testing scope expands accordingly. Test cases are executed at each integration step to validate the functionality, interfaces, and interactions between modules.

- Layered Approach: The integration process proceeds in layers, with each layer representing a higher level of abstraction or functionality. Testing progresses from the lower layers towards the upper layers, ensuring that each layer is thoroughly tested before moving on to the next.

Advantages:

- Early Detection of Defects: Defects are detected and addressed early in the integration process, reducing the risk of issues accumulating and becoming more difficult to resolve.

- Incremental Validation: Incremental integration allows for incremental validation of the software system, providing feedback on the system's functionality and stability at each step.

- Granular Testing: Testing is performed at the module level, enabling granular analysis of module interactions and behavior.

2. Big-Bang Approach

In the Big-Bang approach to integration testing, all software modules are integrated simultaneously, and the entire system is tested as a whole. Unlike the sandwich approach, which integrates modules incrementally, the Big-Bang approach combines all modules in a single integration step, resembling a sudden or simultaneous integration of components.

Explanation

- Simultaneous Integration: All software modules are integrated together in a single step, regardless of their hierarchy or dependencies. The integration process is not performed incrementally but rather in a single, comprehensive integration phase.

- Comprehensive Testing: Once all modules are integrated, the entire system is tested as a whole to verify its functionality, interfaces, and interactions. Test cases are executed to validate the system's behavior and to identify any defects or issues.

- High Degree of Complexity: The Big-Bang approach is typically associated with a higher degree of complexity and risk compared to the sandwich approach. Since all modules are integrated simultaneously, there is a greater potential for conflicts, dependencies, and integration issues.

Advantages:

- Efficiency: The Big-Bang approach can be more efficient in certain scenarios, particularly when the software system is relatively small or when there are no strict dependencies between modules.

- Simplicity: The approach is straightforward and requires less coordination compared to incremental integration approaches.

Conclusion:

Both the sandwich approach and the Big-Bang approach are valid strategies for integration testing, each with its own set of advantages and considerations. The choice between the two approaches depends on factors such as the size and complexity of the software system, the level of risk tolerance, and the availability of resources. Ultimately, the goal of integration testing is to ensure that individual software components work together seamlessly to deliver the desired functionality and performance.

d) Explain load and smoke testing in detail.

Ans: Load Testing:

Load testing is a type of performance testing that evaluates a system's performance under anticipated user loads and operational conditions. The objective of load testing is to assess the system's ability to handle a specific workload within acceptable performance parameters. Load testing helps in identifying performance bottlenecks, scalability issues, and system limitations under varying levels of load. Here's a detailed explanation of load testing:

Explanation:

- Simulating Realistic Workloads: Load testing involves simulating realistic user loads, transaction volumes, and concurrent user interactions to mimic actual usage scenarios. Test scenarios are designed to replicate peak load conditions, typical user behavior, and expected transaction rates.

- Performance Metrics: During load testing, performance metrics such as response time, throughput, resource utilization, and system scalability are monitored and analyzed. Performance counters and monitoring tools are used to measure various performance parameters and identify performance bottlenecks.

- Scalability Assessment: Load testing helps in assessing the system's scalability by gradually increasing the load until performance degradation or system failure occurs. Scalability tests determine the system's capacity to handle increased loads by adding additional resources, such as servers, bandwidth, or processing power.

- Stress Testing: Load testing often includes stress testing scenarios where the system is subjected to extreme or overload conditions beyond its normal operating capacity. Stress tests help in evaluating the system's robustness, stability, and resilience under adverse conditions.

Benefits:

- Identifying Performance Issues: Load testing helps in identifying performance issues such as slow response times, high latency, bottlenecks, and resource constraints before deployment. Early detection of performance issues allows for timely optimization and tuning of the system.

- Validating Scalability: Load testing validates the scalability of the system by determining its capacity to handle increasing loads and user concurrency. Scalability tests help in planning for future growth and capacity expansion.

- Enhancing User Experience: By ensuring optimal performance under load, load testing enhances the user experience by providing a responsive and reliable system that meets user expectations.

- Mitigating Risks: Load testing mitigates the risk of performance-related issues and system failures by proactively identifying and addressing performance bottlenecks and limitations.

Smoke Testing:

Smoke testing, also known as build verification testing (BVT) or sanity testing, is a type of software testing that verifies whether the critical functionalities of an application are working correctly after a new build or release. The term "smoke testing" originates from the electronics industry, where devices were tested by powering them on and checking for smoke, indicating a major failure. In software testing, smoke testing ensures that the basic functionalities of the application are functional and stable before proceeding with further testing. Here's a detailed explanation of smoke testing:

Explanation:

- Critical Functionality: Smoke testing focuses on testing the critical functionalities of the application, such as login, navigation, basic operations, and key workflows. It aims to verify whether these essential features are working as expected and whether the application is stable enough for further testing.

- Quick Verification: Smoke testing is typically performed soon after a new build or release is deployed to ensure that the application is ready for further testing. It involves executing a set of predefined test cases or scenarios that cover the core functionalities of the application.

- Minimal Testing: Smoke testing involves minimal testing effort and focuses on validating the most critical aspects of the application. It does not aim to provide exhaustive test coverage but rather to quickly identify major issues or show-stoppers that could prevent further testing.

- Automated Execution: Smoke testing is often automated to expedite the testing process and provide rapid feedback on the build's stability. Automated smoke tests are executed automatically after each build or deployment, allowing for continuous integration and rapid feedback loops.

Benefits:

- Early Detection of Major Issues: Smoke testing helps in identifying major issues, regressions, or critical defects early in the development lifecycle, reducing the risk of releasing unstable builds to production.

- Efficient Use of Resources: By focusing on critical functionalities, smoke testing optimizes testing efforts and resources, allowing testers to quickly assess the build's stability and decide whether further testing is warranted.

- Faster Feedback: Smoke testing provides rapid feedback on the build's readiness for further testing, enabling development teams to address critical issues promptly and iteratively improve the quality of the software.

- Risk Mitigation: Smoke testing mitigates the risk of releasing unstable or defective builds to stakeholders or end-users by ensuring that the essential functionalities of the application are functional and stable.

In conclusion, load testing evaluates a system's performance under anticipated user loads and operational conditions, while smoke testing verifies the critical functionalities of an application after a new build or release. Both testing techniques play crucial roles in ensuring the reliability, performance, and stability of software applications.

e) Write difference between Static and Dynamic Testing.

Ans: Difference between Static and Dynamic Testing:

1. Nature of Testing:

- Static Testing: Static testing involves examining the software artifacts, such as requirements documents, design specifications, code, and other documentation, without executing the program. It focuses on identifying defects, errors, and inconsistencies in the software artifacts through reviews, inspections, walkthroughs, and static analysis techniques.

- Dynamic Testing: Dynamic testing involves executing the software code and evaluating its behavior during runtime. It focuses on validating the functionality, performance, reliability, and other attributes of the software through test case execution, simulation, and analysis of actual program execution.

2. Timing of Testing:

- Static Testing: Static testing is performed early in the software development lifecycle, typically during the requirements analysis, design, and coding phases. It aims to identify defects and issues as early as possible when they are less costly and easier to address.

- Dynamic Testing: Dynamic testing is performed later in the software development lifecycle, after the software has been implemented or coded. It involves executing test cases against the running software to validate its behavior and functionality under various conditions.

3. Techniques Used:

- Static Testing: Static testing techniques include reviews, inspections, walkthroughs, static code analysis, and other static analysis tools. It relies on manual examination and analysis of software artifacts to identify defects and improve quality.

- Dynamic Testing: Dynamic testing techniques include unit testing, integration testing, system testing, acceptance testing, regression testing, performance testing, and other testing types that involve executing test cases against the software to evaluate its behavior and performance.

4. Detection of Defects:

- Static Testing: Static testing helps in detecting defects, errors, and inconsistencies in software artifacts, such as requirements ambiguities, design flaws, coding mistakes, and documentation errors. It aims to prevent defects from propagating further in the development process.

- Dynamic Testing: Dynamic testing helps in detecting defects and issues in the running software by executing test cases and observing its behavior. It verifies whether the software meets the specified requirements and performs as expected under various test conditions.

In conclusion, static testing involves examining software artifacts without executing the program to identify defects and improve quality early in the development lifecycle, while dynamic testing involves executing the software code and evaluating its behavior to validate functionality and performance later in the lifecycle. Both static and dynamic testing are essential components of the software testing process and complement each other to ensure the reliability, quality, and effectiveness of software applications.

Q4. Attempt any FOUR of the following (out of FIVE)

a) explain test case design for the login process

Ans: Test case design for the login process involves creating a set of scenarios and conditions to verify that the login functionality of a system works as expected. Here's an outline for a test case design for the login process:

1. Valid Credentials Test Case:

   - Description: Verify that users can successfully login with valid credentials.

   - Steps:

     1. Open the login page.

     2. Enter valid username and password.

     3. Click on the login button.

     4. Verify that the user is redirected to the expected page/dashboard.

   - Expected Result: User should be logged in successfully.

2. Invalid Credentials Test Case:

   - Description: Verify that users cannot login with invalid credentials.

   - Steps:

     1. Open the login page.

     2. Enter invalid username and/or password.

     3. Click on the login button.

     4. Verify that an error message is displayed indicating invalid credentials.

   - Expected Result: User should not be logged in and should receive an error message.

3. Password Recovery Test Case:

- Description: Verify that users can recover their password if forgotten.
- Steps:
  1. Open the login page.
  2. Click on the "Forgot Password" link.
  3. Enter the registered email address.
  4. Click on the "Submit" button.
  5. Check the email for a password reset link.
  6. Click on the password reset link received in the email.
  7. Set a new password.
  8. Try logging in with the new password.
- Expected Result: User should be able to reset the password and login successfully.

4. Account Lockout Test Case:

- Description: Verify that after a certain number of failed login attempts, the account gets locked.
- Steps:
  1. Open the login page.
  2. Enter incorrect credentials multiple times (exceeding the maximum allowed attempts).
  3. Verify that the account is locked.

4. Try logging in again with correct credentials after the account is locked.

- Expected Result: Account should be locked after exceeding the maximum allowed login attempts.

b) Stub and driver concept in unit testing.

Ans:

In unit testing, stubs and drivers are essential components used to isolate and facilitate the testing of individual units of code. Here's a concise explanation of both concepts:

1. **Stubs:**

- **Definition:** Stubs are simplified implementations of parts of a software system that are used to simulate the behavior of dependencies (e.g., functions, methods, modules) of the unit under test.

- **Purpose:** Stubs are employed when testing a unit that relies on other units or external resources. By replacing actual dependencies with stubs, developers can control the behavior of those dependencies and focus solely on testing the unit being tested.

- **Example:** Suppose a function being tested requires data from a database. Instead of accessing the actual database, a stub is created to return predefined data, allowing the function to be tested without relying on the database's availability or state.

2. **Drivers:**

- **Definition:** Drivers are pieces of code used to invoke the unit under test and provide test inputs to stimulate its execution. Drivers are typically used when testing units that are lower in the software hierarchy and require interaction with higher-level modules or components.

- **Purpose:** Drivers enable the execution of units of code that cannot run independently due to dependencies on other units or external systems. They provide the necessary environment and inputs for testing the unit under consideration.

- **Example:** Consider testing a module responsible for processing user input. A driver could be created to simulate user interactions by providing predefined input values (e.g., command-line arguments, simulated button clicks) to the module and capturing its output for validation.

In summary, stubs and drivers are integral components of unit testing that enable developers to isolate units of code, control dependencies, and simulate interactions, thereby facilitating thorough testing of software systems at the individual unit level.

c) Explain GUI testing In details.

Ans: GUI testing, or Graphical User Interface testing, is a crucial aspect of software quality assurance that focuses on evaluating the functionality, usability, and consistency of a software application's graphical elements. It involves thorough examination of various GUI components such as buttons, menus, forms, and layouts to ensure they behave as expected, adhere to design specifications, and meet user expectations. Both manual and automated testing techniques are utilized to identify issues such as incorrect rendering, alignment problems, and usability issues, ultimately contributing to improved user satisfaction and product quality.

- GUI testing focuses on verifying the functionality, usability, and consistency of the graphical elements within a software application.
- It involves testing various aspects of the graphical user interface, including buttons, menus, forms, dialogs, icons, fonts, colors, layout, navigation, and responsiveness.
- GUI testing ensures that the application's interface behaves as expected, adheres to design specifications, and meets user expectations.
- Testers use both manual and automated testing techniques to conduct GUI testing, employing a combination of exploration, validation, and regression testing approaches.
- GUI testing helps identify issues such as incorrect rendering, alignment problems, overlapping elements, missing or misleading labels, inconsistent styling, and usability issues.
- By thoroughly testing the graphical user interface, organizations can enhance the usability, accessibility, and overall quality of their

software products, leading to improved user satisfaction and engagement.

In summary, GUI testing is essential for ensuring the functionality, usability, and consistency of the graphical user interface within a software application, thereby enhancing the user experience and overall quality of the product.

d) What is difference between client/server or web based testing?

Ans: Client/server and web-based testing are two distinct approaches used in software testing, each catering to different types of applications. Here's a concise explanation of their differences:

1. **Client/Server Testing:**

- **Definition:** Client/server applications consist of two separate components - a client-side interface responsible for user interaction and a server-side component handling data processing and storage.

- **Testing Focus:** Client/server testing primarily focuses on verifying the functionality, performance, and compatibility of both the client and server components individually and in conjunction.

- **Testing Environment:** It typically involves setting up a network environment where client and server systems communicate to simulate real-world interactions.

- **Examples:** Testing database systems where clients interact with a centralized database server, or testing enterprise applications where clients access data and services hosted on a central server.

2. **Web-Based Testing:**

- **Definition:** Web-based applications are accessed via web browsers over the internet or an intranet. These applications rely on client-side technologies such as HTML, CSS, and JavaScript, and server-side technologies like PHP, ASP.NET, or Node.js.

- **Testing Focus:** Web-based testing emphasizes validating the functionality, usability, performance, and security of the application across different web browsers and devices.

- **Testing Environment:** It involves testing the application on various browsers (e.g., Chrome, Firefox, Safari), different operating systems (Windows, macOS, Linux), and diverse devices (desktops, laptops, tablets, smartphones).

- **Examples:** Testing e-commerce websites, social media platforms, online banking portals, or any application accessible through a web browser.

In summary, while client/server testing focuses on validating the interaction between client and server components within a networked environment, web-based testing concentrates on assessing the functionality and compatibility of applications accessed through web browsers across different platforms and devices. Both approaches are essential in ensuring the reliability and quality of software systems but cater to distinct types of applications and testing requirements.

e) Calculate the cyclometric complexity of a code which accepts 3 integer values and print the highest and lowest values.

Ans: Cyclomatic complexity is a software metric used to measure the complexity of a program. It is calculated based on the number of decision points (e.g., loops, conditionals, branches) in the code. The formula for calculating cyclomatic complexity is:

¥[M = E - N + 2P ¥]

Where:

- ¥(M ¥) is the cyclomatic complexity.

- ¥(E ¥) is the number of edges in the control flow graph.

- ¥(N ¥) is the number of nodes in the control flow graph.

- ¥(P ¥) is the number of connected components (usually 1).

For your code, which accepts 3 integer values and prints the highest and lowest values, the control flow graph would be quite simple. Let's break it down:

1. Read three integer values.

2. Determine the highest and lowest values among the three.

3. Print the highest and lowest values.

This code doesn't have any loops or conditionals. It has a single entry point and a single exit point. So, the control flow graph would consist of one node and no edges.

Therefore:

- E= 0 (no edges)

- N= 1 (one node)

- P = 1 (one connected component)

Using the formula, the cyclomatic complexity (M) would be:

M[M = 0 - 1 + 2 * 1 = 1]

So, the cyclomatic complexity of your code is 1.

Q5. Write a short note on any TWO of the following(out of THREE).

a) Testing for real time system.

Ans: Testing for real-time systems is a critical aspect of ensuring reliability and predictability in applications designed to respond to external events within specific time constraints. Here's a concise note highlighting its key aspects:

Testing for Real-Time Systems:

- Real-time systems are designed to process and respond to events within predetermined timeframes, making thorough testing essential for their proper functioning.

- This testing involves verifying the system's ability to meet stringent timing requirements, including response time, latency, and meeting deadlines.

- Techniques such as stress testing, performance testing, and timing analysis are employed to evaluate the system's behavior under normal and peak loads, ensuring it operates within acceptable limits.

- Additionally, real-time system testing validates functionalities like event handling, concurrency, and fault tolerance to guarantee seamless operation in dynamic and demanding environments.

b) Stub and Driver concept in unit testing in detail

Ans: In unit testing, stubs and drivers are essential components used to isolate and facilitate the testing of individual units of code. Here's a concise explanation of both concepts:

1. **Stubs:**

- **Definition:** Stubs are simplified implementations of parts of a software system that are used to simulate the behavior of dependencies (e.g., functions, methods, modules) of the unit under test.

- **Purpose:** Stubs are employed when testing a unit that relies on other units or external resources. By replacing actual dependencies with stubs, developers can control the behavior of those dependencies and focus solely on testing the unit being tested.

- **Example:** Suppose a function being tested requires data from a database. Instead of accessing the actual database, a stub is created to return predefined data, allowing the function to be tested without relying on the database's availability or state.

2. **Drivers:**

- **Definition:** Drivers are pieces of code used to invoke the unit under test and provide test inputs to stimulate its execution. Drivers are typically used when testing units that are lower in the software hierarchy and require interaction with higher-level modules or components.

- **Purpose:** Drivers enable the execution of units of code that cannot run independently due to dependencies on other units or external systems. They provide the necessary environment and inputs for testing the unit under consideration.

- **Example:** Consider testing a module responsible for processing user input. A driver could be created to simulate user interactions by providing predefined input values (e.g., command-line arguments, simulated button clicks) to the module and capturing its output for validation.

c) Load Runner

Ans: LoadRunner is a widely-used performance testing tool developed by Micro Focus, designed to assess the scalability and reliability of software applications under various load conditions. Here's a concise note highlighting its key features and benefits:

LoadRunner:

- LoadRunner is a comprehensive performance testing tool used by organizations worldwide to simulate thousands of users accessing an application simultaneously, thus measuring its performance under heavy loads.

- It supports various protocols, including HTTP, HTTPS, Web Services, Java Messaging Service (JMS), Database, and more, enabling testers to simulate real-world scenarios and evaluate application performance across different platforms and technologies.

- LoadRunner offers a user-friendly interface, making it accessible to both technical and non-technical users. It provides a robust scripting environment for creating test scenarios, allowing testers to customize and parameterize scripts to mimic diverse user behaviors.

- With LoadRunner's advanced monitoring and analysis capabilities, testers can closely monitor system resources, identify performance bottlenecks, and pinpoint areas for optimization. Its integrated analysis tools offer in-depth reports and graphs, facilitating comprehensive performance analysis and decision-making.

- LoadRunner supports distributed testing, allowing testers to distribute load generators across multiple locations to simulate geographically dispersed user traffic realistically. This capability enables organizations to assess application performance in real-world scenarios and optimize infrastructure accordingly.

In summary, LoadRunner is a powerful and versatile performance testing tool that empowers organizations to identify and address performance issues early in the development lifecycle, ensuring the delivery of high-quality and reliable software applications to end-users.