**Static Testing:**

**Definition:**Static testing is a software testing technique in which the software or related documentation is examined without executing the code. It involves reviewing and analyzing the software artifacts, such as requirements, design documents, and code, to identify defects, improve quality, and ensure adherence to coding standards.

**Advantages of Static Testing:**

1. **Early Defect Detection:**Static testing allows for the identification of defects early in the software development life cycle, reducing the cost and effort associated with fixing issues later in the process.
2. **Cost-Effective:**Since static testing is performed without executing the code, it is a cost-effective way to find and fix defects. It doesn't require running the entire application, making it less resource-intensive.
3. **Improved Quality:**By identifying and addressing issues in the early stages, static testing contributes to overall software quality. It helps prevent defects from propagating through subsequent phases of development.
4. **Knowledge Transfer:**Static testing provides an opportunity for team members to share knowledge. It promotes collaboration and helps team members gain a better understanding of the software artifacts.
5. **Standard Adherence:**Static testing helps ensure that the software artifacts adhere to coding standards, design guidelines, and best practices, leading to more consistent and maintainable code.

**Disadvantages of Static Testing:**

1. **Limited Scope:**Static testing may not catch all types of defects, especially those related to runtime behavior and interactions that can only be identified through dynamic testing.
2. **Subjectivity:**The effectiveness of static testing can be influenced by the subjectivity of reviewers. Different team members may interpret requirements or code differently, leading to varying results.
3. **Time-Consuming:**Depending on the size and complexity of the software artifacts, static testing can be time-consuming. It may require significant effort for thorough reviews, especially in large projects.
4. **Not Applicable to All Artifacts:**Some artifacts, such as executable code cannot be fully tested using static techniques. Dynamic testing is necessary to validate the runtime behavior of the software.

In practice, a combination of static and dynamic testing is often used to achieve comprehensive software testing coverage. While static testing is effective in finding certain types of defects early, dynamic testing is essential for validating the actual runtime behavior of the software. Both approaches contribute to building reliable and high-quality software.

**Structured Group Examinations**

**Definition:**Structured group examinations typically refer to formal and organized processes where a group of individuals collaboratively examines and reviews a document, code, or other artifacts in a structured manner. This structured approach involves predefined roles, guidelines, and a systematic process for evaluating the quality, correctness, and adherence to standards of the examined material.

**Advantages:**

1. **Comprehensive Review:**Structured group examinations ensure a more thorough and comprehensive review of software artifacts as multiple perspectives are considered.
2. **Knowledge Sharing:**The process facilitates knowledge sharing among team members. It provides an opportunity for less experienced members to learn from more experienced colleagues.
3. **Consistency:**Having predefined roles and guidelines promotes consistency in the review process, ensuring that key aspects are consistently examined across different artifacts.
4. **Identification of Defects:**The collaborative nature of structured group examinations enhances the likelihood of identifying defects, errors, or areas for improvement in the reviewed material.
5. **Enhanced Communication:**The structured approach encourages effective communication among team members, fostering a better understanding of the project and its requirements.

**Disadvantages:**

1. **Time-Consuming:**Structured group examinations can be time-consuming, particularly in large projects or when reviewing extensive documentation. This may impact development timelines.
2. **Subjectivity:**Despite having a structured process, there may still be some subjectivity in the interpretation of standards or guidelines by different team members, potentially leading to variations in reviews.
3. **Resource Intensive:**Organizing and conducting structured group examinations may require a significant allocation of resources, including time and personnel.

It's worth noting that the specific advantages and disadvantages may vary based on the nature of the project, the artifacts being examined, and the characteristics of the development team. Adjustments to the process can be made to mitigate potential challenges and enhance the effectiveness of structured group examinations.

**Static Analysis**

**Definition:**Static analysis is a software testing technique that involves examining the source code, design documents, or other software artifacts without executing the program. The primary goal is to identify potential issues, defects, and vulnerabilities by analyzing the code or documentation statically, before the code is run.

**Advantages of Static Analysis:**

1. **Early Defect Detection:**Static analysis enables the early detection of defects, potential errors, and code quality issues before the code is executed. This can significantly reduce the cost and effort required for later-stage defect fixing.
2. **Consistency and Standards Adherence:**Static analysis tools can enforce coding standards, best practices, and design guidelines, ensuring that the codebase adheres to consistent coding styles and quality standards.
3. **Identification of Security Vulnerabilities:** Static analysis is effective in identifying security vulnerabilities and potential weaknesses in the code that might lead to security breaches. It contributes to building more secure software.
4. **Automation:**Static analysis tools can be automated to scan code and provide quick feedback to developers. This automation helps in integrating static analysis into the development workflow, promoting continuous code quality improvement.
5. **Increased Productivity:**By catching potential issues early in the development process, developers can address them more efficiently, leading to increased productivity and faster development cycles.
6. **Documentation Improvement:**Static analysis can contribute to improving documentation by highlighting inconsistencies or discrepancies between the code and associated documentation.

**Disadvantages of Static Analysis:**

1. **False Positives and Negatives:**Static analysis tools may generate false positives (indicating issues that aren't actual problems) or false negatives (missing actual issues). This requires careful manual review to filter out valid issues.
2. **Limited Understanding of Runtime Behaviour:**Static analysis examines the code in isolation and may not fully capture the runtime behaviour or interactions with other components. Certain issues can only be identified through dynamic analysis.
3. **Resource Intensive:**Performing thorough static analysis on large codebases can be resource-intensive, requiring significant computational resources and potentially leading to longer analysis times.
4. **Lack of Context:**Static analysis tools may lack the contextual understanding of the business logic or project-specific requirements, leading to limitations in identifying certain issues that require domain knowledge.
5. **Complexity Handling:**Some static analysis tools may struggle with handling the complexity of certain code structures, especially in cases involving advanced programming paradigms or language features.
6. **Learning Curve:**Introducing static analysis tools may have a learning curve for development teams. It may take time for team members to understand and configure the tools effectively.

In practice, a balanced approach involves combining static analysis with other testing techniques, such as dynamic analysis and manual reviews, to achieve comprehensive software quality assurance.

**Using a tool like SonarQube or Checkmarx to analyze a codebase and identify issues such as unused variables, memory leaks, or potential security risks like SQL injection vulnerabilities.**

**Control flow & Data flow**

**Control Flow:**

**Definition: Control flow refers to the order in which instructions or statements are executed in a program. It determines the path that the program follows as it progresses from one statement to the next. Control flow is governed by control structures like conditionals (if statements, switch statements) and loops (for, while loops).**

**Key Concepts:**

* **Conditionals: Decision-making structures that execute different code blocks based on specified conditions.**
* **Loops: Repetitive structures that execute a set of statements multiple times until a certain condition is met.**
* **Function Calls: The transfer of control to a function, which then returns control to the calling code upon completion.**

**Data Flow:**

**Definition: Data flow represents the movement and transformation of data within a program. It describes how data is input, processed, and output in a system. Understanding data flow is crucial for tracking how variables are created, modified, and utilized throughout the execution of a program.**

**Key Concepts:**

* **Variables: Storage locations for data that can be manipulated during program execution.**
* **Assignment: The process of giving a value to a variable, altering its data flow.**
* **Expressions: Combinations of variables, operators, and constants that yield a value.**

**Relationship Between Control Flow and Data Flow:**

**Control flow and data flow are interconnected in a program:**

* **Control flow decisions (e.g., if statements) can determine the execution path and influence data flow.**
* **Data flow may affect control flow when the value of variables influences the branching or looping behaviour of the program.**

**A holistic understanding of both control flow and data flow is essential for comprehensive program comprehension, testing, and debugging. Analysing control and data flow helps developers and testers identify potential issues, optimize code, and ensure the correct functioning of a software system.**

**2.1.5    Determining Metrics**

**Determining metrics is a critical aspect of software development and project management. Metrics provide quantitative measures that help assess various aspects of the software development process, product quality, and team performance. The selection of metrics should align with project goals and objectives. Here are some common areas where metrics are often determined:**

1. **Productivity Metrics:**
   * **Lines of Code (LOC): Measures the size of the codebase. However, this metric should be used cautiously as it doesn't necessarily reflect productivity accurately.**
   * **Function Points: Measures the functionality provided by a software system. It's often used to estimate project effort and compare productivity across projects.**
2. **Quality Metrics:**
   * **Defect Density: The number of defects per unit of size (e.g., defects per KLOC). Helps identify the quality of the codebase.**
   * **Code Coverage: Measures the percentage of code executed during testing. Indicates how well the code has been exercised by tests.**
   * **Static Code Analysis Violations: The number of rule violations identified by static analysis tools. Indicates potential code quality issues.**
3. **Testing Metrics:**
   * **Test Coverage: Measures the extent to which the application has been tested. It can include code coverage, requirements coverage, and more.**
   * **Defect Rejection Rate: The percentage of defects initially found by testing that were rejected by development. A lower rejection rate is generally better.**
   * **Test Case Effectiveness: Measures how effective test cases are in finding defects. It includes metrics like defect detection rate.**
4. **Efficiency Metrics:**
   * **Time to Fix Defects: Measures the time taken to address and fix defects reported during testing.**
   * **Time to Market: Measures the time it takes to deliver a product or feature from the beginning of development to release.**
5. **Customer Satisfaction Metrics:**
   * **User Satisfaction Surveys: Collect feedback from end-users to assess their satisfaction with the software.**
   * **Net Promoter Score (NPS): Measures the likelihood of users recommending the product to others.**
6. **Process Metrics:**
   * **Lead Time: Measures the time taken from the initiation of work to its completion.**
   * **Cycle Time: Measures the time taken to complete one iteration of a process.**
7. **Agile Metrics:**
   * **Velocity: Measures the amount of work completed in a sprint. Helps in planning future sprints.**
   * **Burndown Chart: Visualizes the progress of completing the user stories or tasks over time.**
8. **Security Metrics:**
   * **Number of Security Vulnerabilities: Measures the count of identified security vulnerabilities in the software.**
   * **Time to Patch: Measures the time taken to fix security vulnerabilities.**

**When determining metrics, it's important to consider the context, team goals, and the specific needs of the project. Additionally, metrics should be used judiciously to avoid unintended consequences, and continuous evaluation of their relevance and effectiveness is essential.**

**2.2.1    Dynamic Testing**

**Definition: Dynamic testing is a software testing technique where the software is executed and tested against various inputs to observe its behavior, performance, and other dynamic characteristics. This type of testing involves running the software and assessing its responses to different test cases. Dynamic testing helps identify defects, errors, and unexpected behaviors that may arise during the execution of the software.**

**Key Aspects of Dynamic Testing:**

1. **Test Case Design:**
   * **Creation of Test Cases: Testers design test cases based on specifications, requirements, and other relevant documentation.**
   * **Input Data Selection: Test cases include specific input data that is designed to exercise different aspects of the software, such as boundary values, error conditions, and valid inputs.**
2. **Test Execution:**
   * **Running the Software: Dynamic testing involves executing the actual software with the specified test cases.**
   * **Observing Behaviour: Testers observe the behaviour of the software, including any errors, crashes, unexpected outputs, or performance issues.**
3. **Types of Dynamic Testing:**
   * **Functional Testing: Focuses on verifying that the software functions as expected according to the specified requirements. It includes unit testing, integration testing, system testing, and acceptance testing.**
   * **Non-functional Testing: Targets non-functional aspects of the software, such as performance, security, usability, and reliability. Examples include performance testing, security testing, and usability testing.**
4. **Dynamic Analysis:**
   * **Code Instrumentation: In dynamic testing, code may be instrumented to collect information about the software's runtime behaviour. This is common in tools used for profiling, code coverage analysis, and performance monitoring.**
   * **Memory Analysis: Dynamic testing can involve monitoring memory usage to identify memory leaks or excessive memory consumption during runtime.**
5. **Debugging and Troubleshooting:**
   * **Error Identification: Dynamic testing helps in identifying and locating errors, defects, and unexpected behaviors in the software.**
   * **Logging and Tracing: Developers often use logging and tracing mechanisms to capture runtime information for later analysis and debugging.**
6. **Regression Testing:**
   * **Ensuring Stability: Dynamic testing is essential for regression testing, where previously identified defects are retested to ensure they have been fixed and to check for any new issues introduced during development.**
7. **Automation:**
   * **Automated Testing: Dynamic testing can be performed manually or through automated testing tools. Automated testing is particularly beneficial for repetitive tests, regression tests, and large-scale testing efforts.**
8. **Test Reporting:**
   * **Results Analysis: Testers analyze the results of dynamic testing to assess the overall quality of the software.**
   * **Defect Reporting: Any defects or issues identified during dynamic testing are documented and reported to the development team for resolution.**

**Dynamic testing is a crucial part of the overall testing strategy, complementing static testing techniques such as code reviews and inspections. Both dynamic and static testing contributes to the identification and resolution of defects in the software development process.**

**2.2.2    Black Box Testing**

**Definition:**Black box testing is a software testing method in which the internal structure, design, or implementation details of the software being tested are not known to the tester. The focus is on validating the functionality and behaviour of the software from an external perspective. Testers treat the software as a "black box," testing it based on its inputs and observing its outputs without having knowledge of the internal code.

**Key Aspects of Black Box Testing:**

1. **Test Design:**
   * Testers design test cases based on the specifications, requirements, or functional documentation provided for the software.
   * Test cases are created to cover different scenarios, including valid inputs, invalid inputs, boundary conditions, and expected system behaviour.
2. **No Knowledge of Internal Code:**
   * Testers do not have access to the source code, algorithms, or the internal logic of the software being tested.
   * Testing is performed solely based on the externally visible behaviour of the software.
3. **Functional and Non-functional Testing:**
   * Focuses on validating the functionality of the software to ensure it meets the specified requirements.
   * Also includes testing non-functional aspects such as performance, usability, security, and compatibility.
4. **Types of Black Box Testing:**
   * Functional Testing: Verifies that the software functions as expected according to the specified requirements. Includes unit testing, integration testing, system testing, and acceptance testing.
   * Non-functional Testing: Ensures that non-functional requirements, such as performance, reliability, and security, are met without knowledge of the internal implementation.
5. **Testing Levels:**
   * Black box testing can be applied at various testing levels, including unit testing, integration testing, system testing, and acceptance testing.
   * Each level of testing focuses on different aspects of the software's functionality and interactions.
6. **Equivalence Partitioning:**
   * Testers divide input data into partitions or classes and select representative test cases from each partition to ensure comprehensive coverage.
7. **Boundary Value Analysis:**
   * Test cases are designed to evaluate the behaviour of the software at the boundaries of input ranges.
8. **Regression Testing:**
   * Black box testing is often used for regression testing, where previously identified defects are retested to ensure they have been fixed and to check for any new issues introduced during development.

**Advantages of Black Box Testing:**

1. **Tester Independence:**Testers don't need knowledge of the internal code, allowing for independence from the development team.
2. **Focus on End-User Perspective:** Emphasizes testing from the end-user's perspective, ensuring that the software meets user requirements.
3. **Encourages Systematic Testing:**Allows for systematic and comprehensive testing of the software's functionality and behaviour.

**Disadvantages of Black Box Testing:**

1. **Limited Coverage:**May not provide complete coverage of all possible scenarios, especially if requirements are incomplete or unclear.
2. **Redundancy in Testing:**Some functionalities may be tested redundantly at multiple testing levels, leading to inefficiencies.

Black box testing is an essential part of a balanced testing strategy, working in conjunction with white box testing and other testing techniques to ensure thorough software validation.

**2.2.3    Equivalence Class Partitioning**

Equivalence Class Partitioning (ECP), also known as Equivalence Partitioning or E-partitioning, is a software testing technique that divides the input domain of a system into classes of equivalent and representative values. The goal of equivalence class partitioning is to reduce the number of test cases needed to effectively test a software system while ensuring that each class is tested at least once.

**Key Concepts of Equivalence Class Partitioning:**

1. **Equivalence Classes:**
   * An equivalence class is a set of input values that are expected to behave in a similar way from the perspective of the system being tested.
   * Each class represents a distinct behaviour or outcome.
2. **Partitioning:**
   * The input domain is divided into different partitions, with each partition representing an equivalence class.
   * Partitions are created based on the assumption that if one value in a class behaves in a certain way, other values in the same class are likely to behave similarly.
3. **Selection of Representative Values:**
   * Test cases are then designed to include at least one value from each equivalence class to ensure comprehensive coverage.
   * The selection of representative values aims to test the system's behavior under different conditions.

**Example of Equivalence Class Partitioning:**

Consider a simple scenario of validating user age input for a website that requires users to be between 18 and 65 years old:

* **Equivalence Classes:**
  + Class 1: Age less than 18 (Invalid)
  + Class 2: Age between 18 and 65 (Valid)
  + Class 3: Age greater than 65 (Invalid)
* **Test Cases:**
  + Test Case 1: Age = 15 (Class 1 - Invalid)
  + Test Case 2: Age = 25 (Class 2 - Valid)
  + Test Case 3: Age = 70 (Class 3 - Invalid)

In this example, the input domain (user age) is partitioned into three equivalence classes based on the expected behaviour of the system. Test cases are then designed to cover each class.

**Advantages of Equivalence Class Partitioning:**

1. **Efficiency:**Reduces the number of test cases needed to cover the input domain, making testing more efficient.
2. **Comprehensive Coverage:**Ensures that various scenarios are tested, including valid and invalid inputs.
3. **Error Detection:**Increases the likelihood of detecting defects associated with different input conditions.

**Considerations:**

1. **Boundary Values:**Equivalence class partitioning often considers boundary values as they are critical for testing.
2. **Combining Classes:**Sometimes, it may be appropriate to combine certain equivalence classes if their behaviour is expected to be similar.

While equivalence class partitioning is a valuable technique, it is often used in conjunction with other testing techniques to ensure thorough coverage of the system under test.

**Top of Form**

**Bottom of Form**

**2.2.4    Boundary Value Analysis**

Boundary Value Analysis (BVA) is a software testing technique that focuses on testing values at the boundaries of the input domain. The idea behind this technique is that errors and defects are more likely to occur near the edges of the input range rather than in the middle. By testing values at the boundaries, testers aim to identify potential issues related to boundary conditions, which are often the cause of many defects.

**Key Concepts of Boundary Value Analysis:**

1. **Input Domain:**
   * BVA is primarily applied to the input domain of a system, where input values are expected to fall within a specific range or set of constraints.
2. **Boundary Conditions:**
   * The boundaries of the input domain are the values at which the behaviour of the system might change. These are typically the minimum and maximum values allowed.
3. **Test Values:**
   * Test values are selected to be on the boundaries or immediately outside the boundaries of the input domain.
   * Values are chosen to cover the lower and upper limits, as well as values just below and just above these limits.
4. **Categories of Boundary Value Analysis:**
   * **Minimum Values:**Test values at the lower boundaries.
   * **Just Above Minimum:**Test values immediately above the lower boundaries.
   * **Maximum Values:**Test values at the upper boundaries.
   * **Just Below Maximum:**Test values immediately below the upper boundaries.

**Example of Boundary Value Analysis:**

Consider a system that accepts positive integers from 1 to 100 inclusive. The boundary values for this input domain would be 1 (minimum), 100 (maximum), 2 (just above the minimum), and 99 (just below the maximum).

* **Test Cases:**
  + Test Case 1: Input = 1 (Minimum)
  + Test Case 2: Input = 2 (Just above the minimum)
  + Test Case 3: Input = 99 (Just below the maximum)
  + Test Case 4: Input = 100 (Maximum)

In this example, the test cases are selected to cover the boundaries and values immediately outside the boundaries to ensure thorough testing.

**Advantages of Boundary Value Analysis:**

1. **Efficiency:**Focuses testing efforts on critical areas where defects are more likely to occur.
2. **Comprehensive Coverage:**Helps ensure that different boundary conditions are tested, increasing the likelihood of defect detection.
3. **Early Detection:** Often identifies defects early in the testing process, reducing the cost of fixing issues later.

**Considerations:**

1. **Combination with Equivalence Class Partitioning:**BVA is often used in conjunction with Equivalence Class Partitioning for comprehensive testing.
2. **Adaptability:**Testers need to adapt BVA to the specific requirements and constraints of the system under test.

While BVA is a powerful technique, it is important to note that it should not be the sole testing strategy. Combining BVA with other testing techniques, such as Equivalence Class Partitioning and functional testing, provides more comprehensive test coverage.

**Top of Form**

**2.2.5    Cause Effect Graphing and Decision Table Technique**

**Cause-Effect Graphing:**

Cause-Effect Graphing, also known as Cause-and-Effect Graph, is a software testing technique that helps design test cases based on the relationships between different factors or conditions that can affect the behaviour of a system. It is particularly useful for systems with complex inputs and dependencies. The technique involves creating a graphical representation of the possible combinations of input conditions and their corresponding outcomes.

**Key Concepts of Cause-Effect Graphing:**

1. **Factors and Conditions:**
   * Identify the various factors or conditions that can influence the behavior of the system.
   * Factors may be input variables, states, or events that can affect the system.
2. **Cause-Effect Graph:**
   * Create a cause-effect graph that visually represents the relationships between different factors and the corresponding outcomes or effects.
   * Factors are represented as nodes, and the relationships between them are depicted as edges.
3. **Test Case Generation:**
   * Generate test cases based on the cause-effect graph to cover different combinations of input conditions.
   * Each test case is designed to exercise a unique combination of factors to ensure comprehensive testing.

**Advantages of Cause-Effect Graphing:**

1. **Comprehensive Coverage:**Helps ensure that a broad set of combinations is covered in the testing process.
2. **Visualization:**Provides a clear visual representation of the relationships between factors and outcomes.

**Example:**

A **login system** that has the following conditions:

1. **Condition A**: User enters a correct username.
2. **Condition B**: User enters a correct password.
3. **Condition C**: User has an active account.
4. **Effect X**: Login successful.
5. **Effect Y**: Login failed.

The possible outcomes depend on combinations of the conditions.

**Graph Representation:**

|  |  |
| --- | --- |
| **Causes** | **Effects** |
| A = Correct Username | X = Login Success |
| B = Correct Password | Y = Login Failure |
| C = Active Account |  |

**Logical Relations:**

* **Effect X (Login successful)** occurs if **Condition A, B, and C are all true** (AND relation).
* **Effect Y (Login failed)** occurs if **any of the conditions A, B, or C is false** (OR relation).

**Cause-Effect Graph for Login System:**

A B C

/ \ / \ / \

(True) (False) (True) -> (X = Success)

/ \ / \

[OR] [AND] [AND]

| | | |

(Effect Y - Failure) [AND] -> Effect X

**Possible Test Cases:**

From this graph, we can generate the following test cases:

1. **Test Case 1**: A = true, B = true, C = true → **Login successful (X)**
2. **Test Case 2**: A = false, B = true, C = true → **Login failed (Y)** (Username is incorrect)
3. **Test Case 3**: A = true, B = false, C = true → **Login failed (Y)** (Password is incorrect)
4. **Test Case 4**: A = true, B = true, C = false → **Login failed (Y)** (Inactive account)

**Decision Table Technique:**

**Definition:**The Decision Table Technique is a testing technique that systematically handles combinations of different input conditions and their corresponding actions or outcomes. It is particularly useful for systems with multiple inputs, where the behavior depends on the various combinations of conditions. The decision table represents all possible combinations and helps in designing test cases to cover different scenarios.

**Key Concepts of Decision Table Technique:**

1. **Conditions and Actions:**
   * Identify the input conditions and corresponding actions or outcomes that affect the behaviour of the system.
   * Conditions are factors that influence the decision-making process.
2. **Decision Table:**
   * Create a decision table that systematically lists all possible combinations of input conditions and the associated actions or outcomes.
   * Rows represent different combinations, and columns represent conditions and actions.
3. **Test Case Generation:**
   * Generate test cases based on the decision table to cover different scenarios and combinations of input conditions.

**Advantages of Decision Table Technique:**

1. **Systematic Approach:**Provides a systematic and structured method for handling combinations of input conditions.
2. **Comprehensive Coverage:**Helps ensure that various scenarios are considered, improving test coverage.

Both Cause-Effect Graphing and Decision Table Technique are effective for handling complex combinations of input conditions in a systematic manner, and the choice between them may depend on the specific characteristics of the system under test.

Example:

**Example of Decision Table:**

Let’s consider a simple **discount system** where a customer can get a discount based on the following conditions:

* **Condition 1**: Customer is a **regular customer**.
* **Condition 2**: Customer's **purchase amount** exceeds $100.
* **Action 1**: Apply a **10% discount**.
* **Action 2**: Apply a **5% discount**.
* **Action 3**: **No discount**.

**Decision Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Rule No.** | **Condition 1 (Regular Customer)** | **Condition 2 (Purchase Amount > $100)** | **Action 1 (10% Discount)** | **Action 2 (5% Discount)** | **Action 3 (No Discount)** |
| 1 | Yes | Yes | Yes | No | No |
| 2 | Yes | No | No | Yes | No |
| 3 | No | Yes | No | Yes | No |
| 4 | No | No | No | No | Yes |

**2.3.1   White Box Testing (Statement, Branch Coverage, Test of Conditions, Path Coverage)**

White box testing, also known as clear box testing, glass box testing, or structural testing, is a software testing method where the internal structure, design, and implementation details of the software being tested are known to the tester. In white box testing, the tester has access to the source code of the application, and the testing is done at the code level.

**Key characteristics and aspects of white box testing include:**

1. **Code Coverage:**White box testing aims to ensure that every line of code is executed at least once during the testing process. This helps in identifying any dead code or code that is not exercised by the test cases.
2. **Test Case Design:**Test cases in white box testing are designed based on the internal logic of the code. Testers use techniques such as control flow testing, data flow testing, and path testing to design test cases that cover different paths and conditions in the code.
3. **Focus on Internal Logic:**The testing process involves evaluating the internal logic, data structures, and algorithms used in the software. This helps in uncovering errors or vulnerabilities that may not be apparent from a user's perspective.
4. **Unit Testing:**White box testing is often associated with unit testing, where individual units or components of the software are tested in isolation. This ensures that each part of the software functions as intended.
5. **Automation:**White box testing is well-suited for automation, as it involves the execution of a large number of test cases at the code level. Automated testing tools can be used to facilitate this process.
6. **Performance Testing:**White box testing can also be used to assess the performance of the software by analysing aspects such as code execution times, memory usage, and resource utilization.
7. **Security Testing:**Security vulnerabilities can be identified through white box testing by examining the code for potential weaknesses and vulnerabilities that may be exploited by attackers.

White box testing is complementary to black box testing, where the internal structure of the software is not known to the tester. Together, these testing methods provide comprehensive test coverage and help ensure the reliability and quality of the software.

1. **Statement Coverage:**
   * **Definition:**Statement coverage, also known as line coverage, measures the percentage of executable statements that have been exercised by a set of test cases.
   * **Objective:**The goal is to ensure that every line of code has been executed at least once during testing.
   * **Calculation: Statement Coverage=**Number of Executed StatementsTotal Number of Statements×100Statement Coverage=Total Number of StatementsNumber of Executed Statements​×100

### Example:

def check\_number(num):

if num > 0:

print("Positive number")

else:

print("Non-positive number")

print("End of function")

**Test Cases for Statement Coverage:**

1. **Test Case 1: num = 5**
   * The condition num > 0 evaluates to True, so "Positive number" is printed.
   * "End of function" is also printed after the if-else block.
2. **Test Case 2: num = -5**
   * The condition num > 0 evaluates to False, so "Non-positive number" is printed.
   * "End of function" is printed as well.

**Statement Coverage Calculation:**

* In this example, there are three statements to cover:
  1. if num > 0:
  2. print("Positive number")
  3. print("Non-positive number")
  4. print("End of function")

By running both test cases (num = 5 and num = -5), we cover all the statements:

* num > 0 (both conditions are tested)
* print("Positive number") (covered in the first test case)
* print("Non-positive number") (covered in the second test case)
* print("End of function") (covered in both test cases)

Thus, the statement coverage is 100%, as all statements are executed.

### Statement Coverage Metric:

Statement coverage is calculated as:

Statement Coverage=Number of statements executed/Total number of statements×100

1. **Branch Coverage:**
   * **Definition:**Branch coverage measures the percentage of decision branches that have been exercised by a set of test cases.
   * **Objective:**The goal is to ensure that both true and false branches of decision points have been taken during testing.
   * **Calculation:**Branch Coverage=Number of Executed BranchesTotal Number of Branches×100Branch Coverage=Total Number of BranchesNumber of Executed Branches​×100
2. **Test of Conditions:**
   * **Definition:**Test of conditions involves designing test cases to evaluate the different logical conditions within a program.
   * **Objective:**Ensure that each condition within a decision point is tested under true and false conditions.
3. **Path Coverage:**
   * **Definition:** Path coverage aims to test all possible paths through a program's control flow graph**.**
   * **Objective:**The goal is to ensure that every possible route from the program's entry point to exit point is traversed at least once.
   * **Challenges:**Achieving complete path coverage can be challenging due to the exponential growth in the number of paths as the program complexity increases.

**2.3.2   Gray Box Testing**

Gray Box Testing is a software testing technique that combines elements of both black box testing and white box testing. In Gray box testing, testers have partial knowledge of the internal workings of the system under test. They possess some information about the internal code, architecture, or algorithms, but not the complete details.

**Key Characteristics of Gray Box Testing:**

1. **Partial Knowledge:**
   * Testers have partial access to the internal structure or code of the application.
   * They may have access to design documents, architecture diagrams, or limited portions of the source code.
2. **Focus on Functionalities:**
   * Gray box testing primarily focuses on the functional aspects of the system.
   * Testers aim to validate whether the system behaves correctly based on different inputs and conditions.
3. **Combines Black Box and White Box Techniques:**
   * Testers use a combination of black box and white box testing techniques.
   * They design test cases based on both the expected functionality and some understanding of the internal logic.
4. **Scenario-Based Testing:**
   * Testers often design test scenarios that simulate real-world usage and potential user interactions.
   * Test cases are created to cover various scenarios and conditions that users might encounter.
5. **Advantages of Gray Box Testing:**
   * **Balanced Approach:**Offers a balanced approach by leveraging both external and internal perspectives.
   * **Effective Test Design:**Testers can design more effective test cases by considering both functional requirements and internal logic.
6. **Challenges of Gray Box Testing:**
   * **Limited Internal Knowledge:**Testers may not have access to the complete internal details, limiting their understanding.
   * **Dependency on Documentation:**The effectiveness of Gray box testing depends on the availability and accuracy of documentation.

**Example of Gray Box Testing:**

Consider a scenario where a tester is testing an e-commerce website with partial knowledge of the source code. The tester has information about the data flow between the user interface and the database but does not have access to the detailed algorithms used for payment processing.

* **Gray Box Testing Scenarios:**
  1. **User Registration:**
     + Test the user registration process by providing valid and invalid inputs.
     + Verify that user data is correctly stored in the database.
  2. **Shopping Cart:**
     + Test the functionality of adding items to the shopping cart and updating quantities.
     + Verify that the shopping cart total is calculated accurately.
  3. **Payment Processing:**
     + Simulate payment transactions using various payment methods.
     + Validate that payment information is securely transmitted to the payment gateway.

In this example, the tester uses a combination of functional testing (black box) and an understanding of the data flow (partial white box) to ensure that critical functionalities work as expected.

Gray box testing is valuable in situations where having some knowledge of the internal workings can enhance the effectiveness of test case design and overall test coverage. It allows testers to focus on high-priority areas while considering the external behaviour and internal logic of the system.

**2.3.3   Intuitive and Experience**

**Intuition:**

**Definition:**Intuition is the ability to understand or know something immediately without the need for conscious reasoning. It involves a gut feeling, a deep-seated perception, or a sense of knowing that is not always based on explicit evidence or logical analysis. Intuition is often associated with quick decision-making, where individuals rely on their instincts or subconscious understanding.

**Key Characteristics of Intuition:**

1. **Rapid Decision-Making:**Intuition enables quick decision-making without the need for lengthy analysis or deliberation.
2. **Implicit Knowledge:**It often draws on implicit knowledge and past experiences stored in the subconscious mind.
3. **Subjectivity:**Intuition is subjective and varies from person to person. What feels intuitive for one person may not be the same for another.
4. **Creativity:**Intuition is linked to creativity, allowing individuals to make novel connections or come up with innovative ideas.
5. **Uncertainty:**While intuition can be valuable, it may also be prone to errors or biases, especially in situations of uncertainty.

**Experience:**

**Definition:** Experience refers to the knowledge, skills, and understanding acquired through practical involvement, observation, or active participation in events, activities, or situations over time. It involves learning from real-world encounters and is a key factor in personal and professional development.

**Key Characteristics of Experience:**

1. **Learning:**Experience is a primary means of learning. Individuals accumulate knowledge and skills through firsthand encounters and exposure.
2. **Adaptability:**Experience contributes to adaptability and the ability to navigate various situations based on lessons learned from past events.
3. **Contextual Understanding:**Experienced individuals often have a deeper understanding of the context and nuances related to their field or area of expertise.
4. **Mistakes and Reflection:**Learning from mistakes is a crucial aspect of experience. Reflection on past experiences helps individuals refine their approaches.
5. **Expertise:**Over time, accumulated experience can lead to expertise in a particular domain, allowing individuals to provide valuable insights and guidance.

**Relationship Between Intuition and Experience:**

1. **Informed Intuition:**Experience can inform intuition. Individuals with extensive experience in a specific domain may develop intuitive insights based on their deep understanding of patterns and trends.
2. **Decision-Making:**While intuition can facilitate quick decision-making, experience provides the foundation for making informed and effective decisions.
3. **Learning from Experience:**Intuition can guide individuals in situations where explicit knowledge is lacking, but experience remains a key source of learning.

In various fields, individuals often rely on a combination of intuition and experience to navigate complex situations. While intuition can be valuable for quick judgments, experience provides the depth of understanding and knowledge necessary for sustained success and growth. Both intuition and experience play complementary roles in decision-making, problem-solving, and personal development.

**2.3.4   Risk-Based Testing**

**Risk-Based Testing:**

Risk-Based Testing is a software testing approach that focuses on prioritizing and allocating testing efforts based on the identified risks associated with the software system. The goal is to ensure that testing efforts are directed towards areas of the application that are most likely to be problematic or have the highest impact if defects are present.

**Key Concepts of Risk-Based Testing:**

1. **Risk Assessment:**
   * Identify and assess potential risks associated with the software project. Risks can include uncertainties, complexities, dependencies, and critical functionalities.
2. **Risk Prioritization:**
   * Prioritize identified risks based on their likelihood of occurrence and their potential impact on the project's success.
   * High-priority risks are those with a higher probability of occurrence and significant potential impact.
3. **Test Planning:**
   * Develop a test plan that outlines testing strategies, objectives, and test levels based on the prioritized risks.
   * Allocate more testing resources and effort to areas with higher risks.
4. **Test Case Design:**
   * Design test cases that specifically target the higher-risk areas of the application.
   * Ensure that critical functionalities and scenarios associated with identified risks are thoroughly tested.
5. **Continuous Monitoring:**
   * Continuously monitor and reassess risks throughout the software development life cycle.
   * Adjust testing strategies and priorities based on changes in the risk landscape.

**Benefits of Risk-Based Testing:**

1. **Efficient Resource Allocation:**Ensures that testing efforts are focused on areas with the highest potential for defects, optimizing resource utilization.
2. **Early Defect Detection:**Increases the likelihood of detecting critical defects early in the development process.
3. **Effective Test Coverage:**Provides a structured approach to test critical functionalities and scenarios associated with high-priority risks.
4. **Improved Decision-Making:**Facilitates better decision-making by aligning testing efforts with project objectives and risk mitigation strategies.

**Example of Risk-Based Testing:**

Consider a software project for an e-commerce platform. Identified risks might include potential issues with payment processing, security vulnerabilities, and compatibility problems with various browsers. In a risk-based testing approach:

1. **Payment Processing Risk:**
   * Allocate additional testing efforts to thoroughly test the payment processing module, including different payment methods and scenarios.
   * Implement additional security testing measures for secure payment transactions.
2. **Security Vulnerability Risk:**
   * Focus on security testing, including penetration testing and code analysis, to identify and address potential vulnerabilities.
   * Implement encryption measures and secure coding practices.
3. **Compatibility Risk:**
   * Prioritize testing on various browsers and devices to ensure compatibility with a wide range of user environments.
   * Implement responsive design and conduct cross-browser testing.

By addressing these risks proactively through risk-based testing, the project team can enhance the overall quality of the software and reduce the likelihood of critical issues impacting users.

**2.3.5   Alpha, Beta Testing**

**Alpha Testing:**

**Definition:**Alpha Testing is a type of software testing performed by the internal testing team within the organization that developed the software. It is conducted in a controlled environment before the software is released to external users or customers. The main objective of alpha testing is to identify and fix bugs, glitches, and usability issues before the software undergoes beta testing and is released to a wider audience.

**Key Characteristics of Alpha Testing:**

1. **Performed In-House:**
   * Alpha testing is conducted within the organization that developed the software.
   * The testing team is typically composed of internal testers who have a good understanding of the software's design and requirements.
2. **Closed Environment:**
   * The testing environment is controlled, and access is restricted to the internal testing team.
   * Testers simulate real-world usage scenarios to uncover defects and issues.
3. **Objective:**
   * The primary goal is to identify and fix defects before the software is released to a larger audience.
   * It helps in ensuring the software's stability and reliability.
4. **Usability Testing:**
   * Besides functional testing, alpha testing often includes usability testing to assess how easily users can interact with the software.
5. **Collaboration with Developers:**
   * Testers work closely with developers, allowing for rapid communication and quick resolution of identified issues.
6. **Limited User Feedback:**
   * Since alpha testing is conducted internally, the feedback is limited to the organization's employees or dedicated testers.

**Beta Testing:**

**Definition:**Beta Testing is a type of software testing performed by a selected group of external users or customers in a real-world environment. It occurs after alpha testing, and the software is considered feature-complete but may still have undiscovered defects. Beta testing allows organizations to gather user feedback, identify potential issues in different usage scenarios, and make necessary improvements before a wider release.

**Key Characteristics of Beta Testing:**

1. **External Users:**
   * Beta testing involves external users who are not part of the development organization.
   * These users may represent the target audience for the software.
2. **Real-World Environment:**
   * Beta testing is conducted in a real-world environment, allowing testers to use the software in diverse situations.
   * It helps uncover issues that may not have been identified in controlled testing environments.
3. **Limited Duration:**
   * Beta testing has a specific duration during which users can explore the software and provide feedback.
   * Testers are encouraged to use the software extensively and report any problems or suggestions.
4. **Wide User Feedback:**
   * Organizations collect feedback from a diverse group of beta testers, providing valuable insights into different user perspectives.
   * Feedback is used to improve the software's quality and address issues before a general release.
5. **Parallel Development and Testing:**
   * During beta testing, development may continue on the next version of the software, allowing for parallel development and testing efforts.
6. **Public or Closed Beta:**
   * Beta testing can be public, where anyone can participate, or closed, where access is limited to a specific group of users.

**Key Differences:**

* **Timing:**Alpha testing occurs before beta testing.
* **Testers:**Alpha testing involves internal testers, while beta testing involves external users.
* **Environment:**Alpha testing is conducted in a controlled, in-house environment, while beta testing occurs in a real-world, external environment.
* **Objectives:**The primary goal of alpha testing is to identify and fix defects, while beta testing aims to gather user feedback and uncover issues in diverse usage scenarios.

**Both alpha and beta testing are essential stages in the software testing life cycle, contributing to the overall quality and user satisfaction of the software product.**

**2.3.6   Load and Stress Testing**

**Load Testing:**

**Definition:**Load Testing is a type of performance testing that assesses how a system or application behaves under expected and peak load conditions. The goal is to evaluate the system's ability to handle a specific level of concurrent users, transactions, or data volume. Load testing helps identify performance bottlenecks, measure response times, and ensure that the system can meet performance expectations under various loads.

**Key Characteristics of Load Testing:**

1. **Simulates Concurrent Users:**
   * Load testing involves simulating a realistic number of concurrent users or virtual users accessing the system simultaneously.
2. **Stress on Resources:**
   * It stresses various system resources, such as CPU, memory, network bandwidth, and database capacity, to assess performance under load.
3. **Objective Measurement:**
   * Load testing provides objective measurements of system performance, including response times, throughput, and resource utilization.
4. **Identifies Bottlenecks:**
   * The primary goal is to identify performance bottlenecks and limitations in the system architecture that could affect its scalability.
5. **Scalability Assessment:**
   * Load testing helps assess the system's scalability by determining how well it can handle increased loads without significant degradation in performance.
6. **Gradual Load Increase:**
   * Load is gradually increased over time to observe how the system responds at different load levels.
7. **Examples:**
   * Simulating a specific number of users accessing a website simultaneously.
   * Testing an e-commerce platform under peak shopping loads.

**Stress Testing:**

**Definition:**Stress Testing is a type of performance testing that evaluates how a system or application behaves under extreme conditions or beyond its normal operating capacity. The goal is to determine the system's robustness, stability, and ability to recover gracefully from adverse situations. Stress testing helps identify the breaking point or failure conditions of a system.

**Key Characteristics of Stress Testing:**

1. **Extreme Conditions:**
   * Stress testing simulates extreme conditions, such as high user loads, rapid data input, or resource exhaustion, to evaluate the system's behaviour.
2. **Beyond Normal Capacity:**
   * It assesses the system's response when subjected to loads that exceed its normal operating capacity.
3. **Failure Point Identification:**
   * Stress testing aims to identify the point at which the system fails or experiences a degradation in performance.
4. **Recovery Testing:**
   * Stress testing includes scenarios to test the system's ability to recover gracefully after experiencing stress conditions.
5. **Security Implications:**
   * Stress testing may involve testing the system's response to security attacks, such as denial-of-service (DoS) attacks.
6. **Examples:**
   * Subjecting a server to a sudden and significant increase in traffic to assess its response.
   * Testing a database system under conditions of high data input or frequent up**dates.**

**Key Differences:**

* **Focus:**Load testing focuses on assessing performance under expected and peak loads, while stress testing focuses on evaluating system behaviour under extreme conditions.
* **Purpose:**Load testing aims to optimize and tune the system for expected usage, while stress testing aims to identify failure points and assess system robustness under adverse conditions.
* **Load Levels:**Load testing involves testing at expected or defined load levels, while stress testing involves pushing the system beyond its normal operating limits.
* **Failure Handling:**Load testing primarily identifies performance bottlenecks, while stress testing specifically assesses how well the system handles stress and recovers from adverse conditions.

Both load testing and stress testing are crucial for ensuring the performance, reliability, and stability of software systems, especially in scenarios where user demands and environmental conditions can vary significantly.

**2.3.7   Key Performance Indicator (KPI’s) of software testing**

Key Performance Indicators (KPIs) in software testing are measurable metrics that provide insights into the efficiency, effectiveness, and quality of the testing processes. These indicators help assess the progress, identify areas for improvement, and ensure that testing efforts align with project goals. Below are some key performance indicators commonly used in software testing:

1. **Test Coverage:**
   * **Definition:**Test coverage measures the extent to which the software's requirements are exercised by the test cases.
   * **KPI:** Percentage of requirements covered by test cases.
   * **Importance:**High test coverage indicates a thorough testing effort, reducing the likelihood of undetected defects.
2. **Defect Density:**
   * **Definition:**Defect density measures the number of defects identified per unit of code or per test case.
   * **KPI:**Defects per KLOC (thousand lines of code) or defects per test case.
   * **Importance:**Lower defect density indicates a higher code quality and effectiveness of the testing process.
3. **Test Execution Efficiency:**
   * **Definition:**Test execution efficiency measures the speed and efficiency of test case execution.
   * **KPI:**Test cases executed per hour or per day.
   * **Importance:**Higher test execution efficiency suggests effective use of testing resources.
4. **Defect Rejection Rate:**
   * **Definition:**Defect rejection rate measures the percentage of defects initially identified by testers but rejected by developers.
   * **KPI:**Percentage of rejected defects.
   * **Importance:**A low rejection rate indicates good communication and collaboration between testers and developers.
5. **Test Automation Coverage:**
   * **Definition:**Test automation coverage measures the percentage of test cases automated compared to the total.
   * **KPI:** Percentage of test cases automated.
   * **Importance:**Higher automation coverage can improve testing efficiency, especially for repetitive and regression testing.
6. **Test Case Effectiveness:**
   * **Definition:**Test case effectiveness assesses how well test cases identify defects.
   * **KPI:**Percentage of test cases that find defects.
   * **Importance:**Higher test case effectiveness indicates that the test cases are detecting defects efficiently.
7. **Testing Cycle Time:**
   * **Definition:**Testing cycle time measures the time taken to complete a testing cycle, from test planning to execution and closure.
   * **KPI:**Total time taken for testing cycle completion.
   * **Importance:**Shorter testing cycle times can contribute to faster release cycles.
8. **Escaped Defects:**
   * **Definition:**Escaped defects measure the number of defects found by users or in production after the software is released.
   * **KPI:**Number of defects reported post-release.
   * **Importance:**A low number of escaped defects indicates effective testing and reduces the impact on users.
9. **Resource Utilization:**
   * **Definition:**Resource utilization measures the efficient use of testing resources, including people, tools, and environments.
   * **KPI:**Percentage of resource utilization.
   * **Importance:**Efficient resource utilization ensures cost-effective testing processes.
10. **Customer Satisfaction:**
    * **Definition:** Customer satisfaction measures the end-users' satisfaction with the software's quality and performance.
    * **KPI:**Customer feedback and satisfaction surveys.
    * **Importance:** Positive customer satisfaction indicates successful testing efforts meeting user expectations.

These KPIs provide a holistic view of the testing process, from the thoroughness of testing efforts to the impact on end-users. Organizations often tailor their set of KPIs based on project goals, testing objectives, and the specific context of the software being developed and tested.

|  |  |  |
| --- | --- | --- |
| Feature | Postman | SOAPUI |
| API Type | REST, GraphQL, WebSocket | SOAP, REST |
| Ease of Use | Very user-friendly and intuitive | More complex, especially for SOAP |
| Testing Type | Functional, automated tests | Functional, security, load testing |
| Scripting Language | JavaScript | Groovy |
| Collaboration | Workspaces (Team Collaboration) | Limited, unless using SoapUI Pro |
|  |  |  |
| Data-driven Testing | CSV/JSON Support | Advanced support for complex data |
| Performance Testing | Limited | Integrated LoadUI for performance |
| Cost | Free with paid options | Free (Pro requires a paid license) |

**Example RESTful API for Books**

**Base URL**: https://api.example.com/books

**1. GET - Retrieve a list of all books**

This endpoint retrieves all books in the library.

* **Request**:  
  GET https://api.example.com/books
* **Response** (Status 200 OK):

json

Copy

[

{

"id": 1,

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"year": 1960

},

{

"id": 2,

"title": "1984",

"author": "George Orwell",

"year": 1949

}

]

**2. GET - Retrieve a specific book by ID**

This endpoint retrieves a single book's details based on its unique id.

* **Request**:  
  GET https://api.example.com/books/1
* **Response** (Status 200 OK):

json

Copy

{

"id": 1,

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"year": 1960

}

**3. POST - Create a new book**

This endpoint allows you to add a new book to the library.

* **Request**:  
  POST https://api.example.com/books

**Request Body**:

json

Copy

{

"title": "Brave New World",

"author": "Aldous Huxley",

"year": 1932

}

* **Response** (Status 201 Created):

json

Copy

{

"id": 3,

"title": "Brave New World",

"author": "Aldous Huxley",

"year": 1932

}

**4. PUT - Update an existing book**

This endpoint updates the details of an existing book. It requires the book's id.

* **Request**:  
  PUT https://api.example.com/books/1

**Request Body**:

json

Copy

{

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"year": 1961

}

* **Response** (Status 200 OK):

json

Copy

{

"id": 1,

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"year": 1961

}

**5. DELETE - Delete a book**

This endpoint deletes a book based on its id.

* **Request**:  
  DELETE https://api.example.com/books/2
* **Response** (Status 204 No Content):
  + The response body is empty, indicating the book was successfully deleted.

**Summary of API Endpoints:**

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| **GET** | /books | Retrieve all books |
| **GET** | /books/{id} | Retrieve a single book by ID |
| **POST** | /books | Create a new book |
| **PUT** | /books/{id} | Update a book by ID |
| **DELETE** | /books/{id} | Delete a book by ID |