Software Testing & Quality Assurance Practice Set

Target Group : 3rd Year B.Tech

1. **Explain Alpha Testing, Beta Testing, Unit Testing, Integration Testing, System Testing, Regression Testing.**
2. **Alpha Testing**:
   * **Definition**: Alpha testing is conducted by the internal development team or a select group of users. It aims to identify defects and assess the software’s functionality before releasing it to a wider audience.
   * **Purpose**: To validate the software against the requirements and uncover any issues.
   * **Scope**: Typically done in a controlled environment.
   * **Participants**: Developers, testers, and early adopters.
   * **Feedback**: Used to improve the software before beta testing.
3. **Beta Testing**:
   * **Definition**: Beta testing involves external users who test the software in a real-world environment. It helps identify usability issues, compatibility problems, and gathers feedback.
   * **Purpose**: To evaluate the software’s performance, reliability, and user satisfaction.
   * **Scope**: Wider audience, beyond the development team.
   * **Participants**: End-users, customers, or volunteers.
   * **Feedback**: Used to make final adjustments before the official release.
4. **Unit Testing**:
   * **Definition**: Unit testing verifies individual components (units) of the software in isolation.
   * **Purpose**: To ensure that each unit functions correctly.
   * **Advantages**: Early bug detection, improved code quality, and easier maintenance.
   * **Tools**: JUnit, NUnit, xUnit.
5. **Integration Testing**:
   * **Definition**: Integration testing checks interactions between different modules or components.
   * **Purpose**: To identify issues related to data flow, communication, and interfaces.
   * **Advantages**: Detects integration problems early, ensures smooth system behavior.
6. **System Testing**:
   * **Definition**: System testing evaluates the entire software system as a whole.
   * **Purpose**: To validate that the system meets specified requirements.
   * **Scope**: End-to-end testing, including functional and non-functional aspects.
7. **Regression Testing**:
   * **Definition**: Regression testing ensures that new changes do not break existing functionality.
   * **Purpose**: To prevent unintended side effects due to code modifications.
   * **Advantages**: Maintains software stability over time.
8. **Explain White Box Testing with its applications, challenges, merits and demerits.**

What is White Box Testing?

White Box Testing, also known as transparent box, glass box, or structural testing, is a software testing methodology that examines the internal workings of an application. Testers with deep knowledge of the source code, design documents, and algorithms design test cases that target specific code paths, algorithms, and data structures. It's akin to looking "under the hood" of the software to analyze its logic and implementation.

Applications:

* Unit Testing: Ensures individual units of code (functions, modules) work as intended.
* Integration Testing: Verifies how integrated units interact and function together.
* Path Testing: Exercutes all possible code paths to uncover logic errors and edge cases.
* Boundary Value Analysis: Tests at the bounds of input ranges to detect vulnerabilities.
* Control Flow Testing: Checks if decision points execute correctly based on conditions.
* Data Flow Testing: Ensures data manipulation and flow adhere to design specifications.

Challenges:

* Expertise Requirement: Testers need in-depth knowledge of the codebase and testing techniques.
* Maintenance: Test cases require updates as the codebase evolves.
* Time and Cost: Designing comprehensive white-box test suites can be resource-intensive.
* Over-Complexity: For large, intricate systems, testing every code path might be impractical.

Merits:

* Thoroughness: Can uncover logic errors, implementation flaws, and hidden defects.
* Efficiency: Testers can directly target areas most likely to contain errors.
* Early Detection: Catches issues early in the development process, saving time and effort.
* Code Coverage: Quantifies how much of the code has been exercised by test cases.
* Security Enhancement: Identifies potential vulnerabilities based on code structure.

Demerits:

* Bias: Testers might focus on areas they understand well, neglecting others.
* Limited Scope: Only tests what's explicitly coded, not overall user experience.
* Overlooking Requirements: Test cases might not always align with functional requirements.
* Complexity for Large Systems: Testing extensive codebases can be cumbersome.

1. **Explain Black Box Testing with its applications, challenges, merits and demerits.**

*Same as 2*

1. **Explain Debugging with its tools.**

Debugging is the detective work of software development. It's the art of identifying, understanding, and fixing errors (bugs) that prevent your code from working as intended. This crucial process ensures smooth functioning, high quality, and user satisfaction for your software. Let's delve into the world of debugging and its handy tools:

Debugging Tools:

While manual code inspection is a core skill, various tools make debugging more efficient and targeted:

* Debuggers: These are software programs that let you step through your code line by line, inspect variable values, and set breakpoints to pause execution at specific points. Popular options include GDB (C/C++), LLDB (C/C++, Objective-C, Swift), WinDbg (Windows), and Visual Studio Debugger (C#, .NET).
* Logging and Tracing: Print detailed messages at key points in your code to track execution flow and pinpoint errors. Loggers often offer different severity levels (e.g., info, warning, error) to categorize messages.
* Profilers: These tools analyze your code's performance, identifying bottlenecks and areas that consume excessive resources. They help optimize code for speed and efficiency.
* Static Analysis Tools: These tools scan your code without running it, examining potential issues like syntax errors, unused variables, or security vulnerabilities. Popular examples include ESLint (JavaScript), Pylint (Python), and SonarQube (various languages).
* Version Control Systems (VCS): Tools like Git and Mercurial enable tracking code changes over time, allowing you to revert to previous versions if a bug is introduced recently.

*Choosing the Right Tools:*

The choice of tools depends on:

* Programming language: Some tools are specific to languages (e.g., GDB for C/C++, LLDB for Swift).
* Environment: Consider desktop, web, mobile, or embedded systems development.
* Personal preference: Experiment and find tools that suit your work style and familiarity.

1. **Explain Cyclomatic Complexity or Macabe’s Path Method with examples.**

Understanding the complexity of your code is crucial for its maintainability, testability, and overall quality. Here's a breakdown of Cyclomatic Complexity and McCabe's Path Method, key tools in your complexity assessment toolbox:

Cyclomatic Complexity:

* What it is: A software metric that estimates the potential number of independent paths through a program's source code. It reflects the intricacy of control flow, with higher values indicating more complex logic.
* Benefits:
  + Provides a quantitative measure of complexity, aiding in code review and improvement efforts.
  + Helps estimate testing effort, as more paths usually require more test cases.
  + Can serve as an early indicator of potential maintainability issues.
* Calculation:
  + McCabe's formula: V(G) = E - N + 2P
    - V(G): Cyclomatic complexity of the graph (code)
    - E: Number of edges (decisions, loops)
    - N: Number of nodes (executable statements)
    - P: Number of connected components (subgraphs)
* **Example:**

**Python**

def calculate\_area(shape, length, width):

if shape == "rectangle":

return length \* width

elif shape == "circle":

return 3.14 \* radius \* radius

else:

return 0

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

* + This function has E = 2 (if statements), N = 5 (statements), P = 1 (connected component).
  + V(G) = 2 - 5 + 2 \* 1 = 0. This means there's only one path through the code, indicating low complexity.

**McCabe's Path Method:**

* What it is: A specific way to calculate Cyclomatic Complexity by identifying all possible independent paths in a program's control flow graph.
* Steps:
  1. Create a control flow graph of the code.
  2. Identify decision points (e.g., if statements, loops).
  3. Calculate V(G) using McCabe's formula.
* Relationship to Cyclomatic Complexity:
  1. McCabe's Path Method is one way to calculate Cyclomatic Complexity.
  2. It emphasizes visualizing the code's logic through the control flow graph and understanding the potential execution paths.

Examples:

* A simple if statement has a complexity of 1.
* A nested if-else structure has a complexity of 2.
* A switch statement with n cases has a complexity of n + 1.
* A function with two if statements and a loop has a complexity of 4**.**

1. **Explain Manual and Automation Testing.**

Software testing ensures the quality and functionality of software applications. Two key approaches are:

1. Manual Testing:

* Humans execute test cases designed to cover various functionalities and scenarios.
* Involves:
  + Testing user interface (UI) elements for usability and aesthetics.
  + Performing exploratory testing to discover unanticipated issues.
  + Verifying data integrity and system behavior.
  + Documenting test results and defects.
* Advantages:
  + Flexibility: Adapts to changes easily and handles complex, non-repetitive tests.
  + Creativity: Testers can think outside the box and identify edge cases.
  + Cost-effective: For small projects or testing specific functionalities.
* Disadvantages:
  + Time-consuming: Repetitive testing can be slow and prone to human error.
  + Unscalability: Difficult to test large applications or regression testing thoroughly.
  + Subjectivity: Results can vary depending on individual testers' interpretations.

2. Automation Testing:

* Software tools execute pre-defined test scripts to automate repetitive tasks.
* Involves:
  + Writing test scripts to automate specific sequences of actions.
  + Using frameworks and tools to manage test cases and data.
  + Scheduling and running automated tests regularly.
  + Analyzing test results and reports.
* Advantages:
  + Speed and scalability: Efficiently tests large applications and performs repetitive tasks quickly.
  + Accuracy and consistency: Executes tests exactly the same way each time, reducing human error.
  + Regression testing: Easily re-runs tests to ensure new changes haven't caused regressions.
* Disadvantages:
  + Initial investment: Setting up automation infrastructure and scripts can be time-consuming and resource-intensive.
  + Maintenance: Scripts need to be updated as the application evolves.
  + Limited scope: Not suitable for exploratory testing or testing complex user interactions.

Choosing the Right Approach:

The ideal approach often involves a combination of manual and automation testing:

* Use manual testing for:
  + UI/UX testing.
  + Exploratory testing.
  + Testing complex interactions.
* Use automation testing for:
  + Regression testing.
  + API testing.
  + Performance testing.
  + Repetitive tasks.

1. **Explain Software Quality Assurance with examples.**

Software Quality Assurance (SQA) is the umbrella term encompassing all activities that ensure a software product meets the desired quality standards. It's more than just testing; it's a proactive approach to preventing defects and ensuring quality throughout the software development lifecycle (SDLC).

Think of SQA as building a house:

* Foundation: Strong SQA starts with clear requirements, defined processes, and effective communication.
* Construction: Throughout development, SQA involves activities like code reviews, static analysis, and various testing types (unit, integration, functional, etc.).
* Inspection: SQA professionals identify and document defects, ensuring they're resolved before moving forward.
* Final Touches: Before release, SQA activities like performance testing and security assessments ensure the software is ready for users.

Examples of SQA activities:

* Static code analysis: Tools automatically scan code for potential errors and vulnerabilities.
* Unit testing: Individual units of code (functions, modules) are tested in isolation.
* Integration testing: Multiple units are tested together to ensure they work as expected.
* Functional testing: The software's functionality is tested against requirements and specifications.
* Performance testing: The software's responsiveness and ability to handle load are assessed.
* Security testing: The software is evaluated for potential security vulnerabilities.

Benefits of SQA:

* Reduced costs: Catching defects early saves time and money compared to fixing them later.
* Improved quality: SQA helps deliver software that meets user expectations and performs as intended.
* Enhanced security: Proactive security testing helps protect against vulnerabilities.
* Increased customer satisfaction: High-quality software leads to happier customers.

1. **Discuss Statement Coverage, Conditional Coverage with its formula and applications.**

Insoftware testing, understanding how thoroughly your tests cover different aspects of your code is crucial. Two key metrics in this regard are statement coverage and conditional coverage, each offering insights into different levels of code execution.

1. Statement Coverage:

* Measures: What percentage of executable statements in your code were executed at least once during testing.
* Formula: (Number of executed statements) / (Total number of executable statements) \* 100%
* Applications:
  + Ensures basic functionality is exercised.
  + Identifies "dead code" (untested parts).
  + Supports initial test completeness assessment.
  + Useful for simple, linear code but may not be enough for complex logic.

2. Conditional Coverage:

* Measures: How well different conditions (e.g., if, else, loops) are exercised with different input values.
* Types:
  + Decision Coverage: Each condition takes both true and false paths at least once.
  + Condition Coverage: Every possible combination of truth values for all conditions within a decision is met.
  + Modified Condition/Decision Coverage (MCDC): Each condition's outcome independently affects the decision's result (more comprehensive than basic decision coverage).
* Formula:
  + Depends on the specific type of conditional coverage being measured.
  + Refer to specific testing tools or frameworks for exact calculations.
* Applications:
  + Validates code with diverse input combinations.
  + Uncovers logic errors related to specific conditions.
  + Provides deeper insights into code's behavior than statement coverage alone.
  + Particularly valuable for code with complex decision logic.

1. **Explain CMM models with applications.**

The Capability Maturity Model (CMM) family encompasses several models that assess and guide the improvement of software development processes. These models define five maturity levels that organizations can progress through, each signifying increased capability and control over their development practices. Let's delve into the key CMM models and their applications:

1. CMM (Software Capability Maturity Model):

* Focus: Process improvement for better software quality and productivity.
* Levels:
  + Initial: Ad hoc, chaotic processes.
  + Repeatable: Basic project management practices established.
  + Defined: Standardized processes documented and followed.
  + Managed: Quantitative process measurement and control implemented.
  + Optimizing: Continuous improvement through innovation and process optimization.
* Applications: Widely used in government and commercial software development sectors.
* Limitations: Viewed as rigid and less adaptable to agile methodologies.

2. CMMI (Capability Maturity Model Integration):

* Evolution of CMM: Combines CMM with other models like People CMM and SAMM.
* Focus: More flexible and comprehensive process improvement across various organizational aspects.
* Levels: Similar to CMM, but with more specific process areas and maturity levels within each.
* Applications: Widely used across industries, including software development, IT services, and engineering.

3. SCAMM (Security Capability Maturity Model):

* Focus: Secure software development practices and risk management.
* Levels:
  + Ad hoc: Limited security practices.
  + Repeatable: Basic security requirements and policies defined.
  + Defined: Standardized security processes in place.
  + Managed: Quantitative security metrics and controls implemented.
  + Optimizing: Continuous security improvement through innovation.
* Applications: Particularly relevant for organizations developing security-critical software.

4. P-CMM (People Capability Maturity Model):

* Focus: Improving workforce capabilities and competencies.
* Levels:
  + Initial: Reactive approach to workforce management.
  + Repeatable: Defined HR processes and training programs.
  + Defined: Performance-based employee development and competency management.
  + Managed: Quantitative measurement and improvement of workforce capabilities.
  + Optimizing: Continuous improvement of HR practices and talent management.
* Applications: Valuable for organizations seeking to enhance employee skills and engagement.

Benefits of CMM models:

* Process improvement: Guide organizations towards more stable, predictable, and high-quality software development.
* Risk reduction: Proactive identification and mitigation of development risks.
* Cost savings: Improved efficiency and reduced defects can lead to lower development costs.
* Customer satisfaction: Delivering higher quality software leads to happier customers.

Challenges of CMM models:

* Implementation effort: Adopting and implementing a model can be time-consuming and resource-intensive.
* Rigidity: Some models might be perceived as inflexible and difficult to adapt to specific needs.
* Certification cost: Obtaining official CMMI certification can be expensive.

1. **Explain Control flow testing and data flow testing.**

In software testing, understanding diverse approaches is crucial for achieving thorough coverage. Let's dive into control flow testing and data flow testing, exploring their distinct focuses and how they complement each other:

***1. Control Flow Testing:***

* Focus: Examining the execution pathways through your code, specifically how decisions (e.g., if, else, loops) impact code execution.
* Objectives:
  + Ensure all possible execution paths are exercised.
  + Identify logic errors related to decision handling.
  + Uncover code that might never be executed (dead code).
* Techniques:
  + Path testing: Executing all possible execution paths.
  + Statement coverage: Ensuring each statement gets executed at least once.
  + Decision coverage: Taking both true and false branches of each decision point.
  + Condition coverage: Testing all possible combinations of truth values for multiple conditions within a decision.
* Applications: Testing code with complex decision logic, error handling, or intricate branches.

2***. Data Flow Testing:***

* Focus: Examining the flow of data through your code, how variables are defined, used, and manipulated.
* Objectives:
  + Ensure valid data usage and manipulation.
  + Identify issues like uninitialized variables, redundant assignments, or missing definitions.
  + Validate data flow adheres to design specifications.
* Techniques:
  + Data definition-use analysis: Matching variable definitions to their uses.
  + Reach analysis: Determining which statements can affect specific variables.
  + Value set analysis: Identifying possible values a variable can have at different points.
* Applications: Testing code that heavily relies on data manipulation, calculations, or data integrity.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Control Flow Testing** | **Data Flow Testing** |
| Focus | Execution paths based on decisions | Data flow and manipulation |
| Objectives | Logic correctness, path coverage | Data usage, manipulation, integrity |
| Techniques | Path testing, statement coverage | Definition-use analysis, reach analysis |
| Applications | Complex decision logic, error handling | Data-intensive calculations, validations |

1. **Explain review, inspections and walkthrough.**

Review:

* What it is: An informal analysis of code, documentation, or designs by one or more individuals.
* Objectives:
  + Identify obvious errors, inconsistencies, or potential improvements.
  + Promote code collaboration and knowledge sharing.
  + Provide feedback and suggestions for the author.
* Process:
  + Author prepares the material and shares it with reviewers.
  + Reviewers independently examine the material and provide written or verbal feedback.
  + Author addresses the feedback and revises the material accordingly.
* Strengths:
  + Flexible and adaptable to different contexts.
  + Can be quick and cost-effective for small reviews.
  + Encourages communication and knowledge sharing.
* Weaknesses:
  + Subjectivity: Reliant on reviewers' individual expertise and style.
  + Depth of review varies depending on reviewers' time and effort.
  + No formal structure or defect tracking.

2. Inspections:

* What it is: A formal and structured review process involving a defined team with specific roles (moderator, reader, recorder, inspectors).
* Objectives:
  + Identify and document defects with high reliability.
  + Improve code quality through rigorous analysis.
  + Provide measurable improvement over informal reviews.
* Process:
  + Follows a predefined checklist based on coding standards and best practices.
  + Each role has specific responsibilities during the meeting.
  + Defects are documented and tracked for action.
  + Follow-up ensures defects are addressed.
* Strengths:
  + More thorough and systematic than reviews.
  + Improved defect detection and documentation.
  + Quantifiable results and process improvement.
* Weaknesses:
  + More time-consuming and resource-intensive than reviews.
  + Requires training and practice for effective implementation.
  + Might feel less flexible or adaptable to certain situations.

3. Walkthroughs:

* What it is: A collaborative discussion where the author guides reviewers through their work, explaining design decisions and rationale.
* Objectives:
  + Improve understanding of the code or design.
  + Identify potential issues early through discussion and feedback.
  + Promote collaborative problem-solving and learning.
* Process:
  + Author presents their work (code, design, etc.) to a group of reviewers.
  + Reviewers ask questions, provide feedback, and offer suggestions.
  + Author clarifies or defends their design choices.
* Strengths:
  + Encourages active participation and discussion.
  + Promotes knowledge sharing and learning within the team.
  + Can be quick and informal, suitable for early stages of development.
* Weaknesses:
  + Less structured than reviews or inspections, potentially leading to missed issues.
  + Effectiveness depends on active participation and clear communication.
  + No formal defect tracking or follow-up.

1. **Discuss check points with examples.**

The term "checkpoint" can have different meanings depending on the context. Could you please clarify which of the following you'd like me to discuss:

1. Software Development Checkpoints: These are milestones throughout the development process where progress is assessed and potential issues are identified. Examples include requirements review, design review, code review, testing milestones, and integration points.
2. Version Control Checkpoints: These are points in time where a snapshot of a project's files is saved in a version control system (e.g., Git). This allows reverting to previous versions if needed and tracking changes over time.
3. Game Checkpoints: These are save points in video games where progress is saved, allowing players to restart from that point if they die or want to replay a section.
4. Other Checkpoints: Depending on your specific field or context, "checkpoint" might have other meanings. Please provide more information if you have another interpretation in mind.
5. **Discuss equivalence partitions with examples.**

Equivalence partitioning is a powerful technique in software testing that helps you group similar input values into classes and test only one representative from each class, reducing the number of test cases while maintaining good coverage.

Essentially, you divide the possible input values into sets where any value within a set will have the same effect on the program. This way, you avoid testing every single possible value, focusing on representatives of each group instead.

Here's how it works:

1. Identify the input: What data does your program accept?
2. Define equivalence classes: Based on the input's characteristics, create groups of similar values that you expect to behave the same way.
3. Choose test cases: Select one representative value from each class to test your program.

Examples:

1. Validating age:
   * Equivalence classes: 0-120 (valid), negative values (invalid), non-numeric input (invalid).
   * Test cases: 25 (valid), -5 (invalid), "abc" (invalid).
2. Calculating area of a shape:
   * Equivalence classes: positive area (valid), zero area (valid), negative area (invalid).
   * Test cases: 10 (valid), 0 (valid), -2 (invalid).
3. **Explain ISO 9000 models with various versions.**

The ISO 9000 family of standards has gone through several revisions over the years. While various models existed in earlier versions, the current iteration focuses on a single standard: ISO 9001:2015. Understanding its history and previous models helps explain the evolution of quality management best practices.

ISO 9000:1987:

* First published version: Introduced three models for quality management systems (QMS):
  + ISO 9001: Model for design, development, production, installation, and servicing.
  + ISO 9002: Similar to 9001 but excluded design (suitable for production and service organizations).
  + ISO 9003: Focused on internal quality management processes (suitable for software development).
* Structure: Based on the UK BS 5750 standard.
* Emphasis: Conformance to a set of prescribed requirements.

ISO 9000:2000:

* Major revision: Merged the three models into a single standard, ISO 9001:2000.
* Shift: Focused on the process approach to quality management, emphasizing continual improvement and effectiveness.
* Key changes:
  + Introduced the Plan-Do-Check-Act (PDCA) cycle for continuous improvement.
  + Emphasis on customer focus and risk management.

ISO 9001:2008:

* Minor revision: Clarified and tightened existing requirements.
* Focus: Further emphasized risk-based thinking and continual improvement.

ISO 9001:2015:

* Current version: Published in September 2015.
* Significant changes:
  + Adopted a high-level structure (HLS) consistent with other management system standards.
  + Increased focus on context of the organization and context of the organization's QMS.
  + Enhanced risk-based thinking and risk management.
  + Stronger emphasis on leadership commitment and engagement.

The key takeaway:

While the specific models have changed, the core principles of quality management remain consistent:

* Customer focus: Meeting and exceeding customer expectations.
* Leadership commitment: Active involvement of leadership in promoting quality.
* Process approach: Managing and improving processes for better outcomes.
* Continual improvement: Striving for ongoing enhancements.

1. **Explain Static Testing.**

1. Static Testing:

Static testing, also known as code analysis, is a software testing methodology that examines code without actually executing it. It involves analyzing the source code itself to identify potential issues, bugs, and vulnerabilities before the code is run. Common static testing techniques include:

* Lint checking: Enforces coding standards and conventions to improve code readability and maintainability.
* Data flow analysis: Examines how data flows through the code to identify potential problems like uninitialized variables or unused calculations.
* Control flow analysis: Checks the logic and decision paths within the code to detect possible errors or unreachable code.
* Static code security analysis: Scans code for known security vulnerabilities and weaknesses.

Benefits:

* Early detection of errors, saving time and money in the long run.
* Improved code quality and maintainability.
* Reduced security risks.

Limitations:

* Cannot detect all errors, especially those requiring runtime execution.
* May generate false positives that need manual verification

1. **Explain Junit with examples.**

JUnit is a popular open-source unit testing framework for Java that simplifies writing and running unit tests. It provides annotations, assertions, and test runners to streamline the testing process.

*Example:*

*Java*

@Test

public void testAdd() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result);

}

1. **Discuss Junit Annotations with its applications.**

JUnit uses annotations to mark methods as test cases and specify their behavior. Here are some common annotations:

* @Test: Identifies a method as a test case.
* @BeforeClass: Executes a method before all test cases in a class.
* @Before: Executes a method before each test case.
* @After: Executes a method after each test case.
* @AfterClass: Executes a method after all test cases in a class.
* @Ignore: Disables a test case.
* @DisplayName: Provides a custom name for a test case.
* @Parameterized: Runs a test with multiple sets of inputs.

Applications:

* Clearly indicating test methods for automated test execution.
* Setting up and tearing down test environments efficiently.
* Organizing and grouping test cases for better readability.
* Customizing test execution behavior, such as skipping or naming tests.

1. **Discuss Junit with simple programs to verify unit and integration testing.**

***Unit Testing:***

***Java***

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result);

}

@Test

public void testSubtract() {

Calculator calculator = new Calculator();

int result = calculator.subtract(5, 2);

assertEquals(3, result);

}

}

***b) Integration Testing:***

***Java***

public class OrderServiceTest {

@Test

public void testPlaceOrder() {

OrderService orderService = new OrderService();

Product product = new Product("123", "T-Shirt", 10.0);

Order order = orderService.placeOrder(product, 2);

assertNotNull(order);

assertEquals(20.0, order.getTotalPrice());

}

}

1. **Discuss Selenium.**

Selenium is a suite of open-source tools for automating web browser interactions. It allows you to write tests that simulate user actions like clicking links, filling forms, and submitting data. This helps ensure that web applications function as expected across different browsers and devices.

Components:

WebDriver: Controls the web browser and interacts with web elements.

IDE: (Optional) Provides a visual interface for recording and editing test scripts.

Grid: (Optional) Enables parallel test execution on multiple machines.

Benefits:

Saves time and effort compared to manual testing.

Improves test coverage and reliability.

Can be used for functional, regression, and other types of web application testing**.**

1. **How software bugs affects the organizations like Meta or Google or banking systems.**

Software bugs can have wide-ranging and significant consequences for organizations like Meta, Google, and banking systems. Here are some potential impacts:

Financial losses:

* Bug fixes can be expensive, especially for critical systems.
* Service disruptions caused by bugs can lead to lost revenue and productivity.
* Reputational damage can impact customer trust and loyalty, potentially leading to financial losses.

Security vulnerabilities:

* Bugs can create security vulnerabilities that attackers can exploit, leading to data breaches, financial losses, and legal repercussions.
* In banking systems, security bugs can have devastating consequences, compromising sensitive financial information and customer trust.

Operational disruptions:

* Bugs can cause crashes, system outages, and performance issues, disrupting normal operations and impacting user experience.
* In social media platforms like Meta, outages can negatively impact user engagement and advertising revenue.

Public trust and brand image:

* Frequent or severe bugs can damage an organization's reputation and erode public trust.
* For companies like Google, maintaining user trust is crucial for its search and advertising business.

Examples:

* A 2021 bug in Facebook's (now Meta) code caused a global outage, affecting billions of users and costing the company an estimated $600 million in lost revenue.
* A 2017 bug in Google's Chrome browser exposed user data to potential theft, damaging user trust and requiring a security patch.
* A 2016 bug in the Equifax credit reporting system exposed the personal information of millions of Americans, leading to financial losses and regulatory penalties.

1. **Explain Du, Dc, DD graph, path graph and cyclomatic complexity with applications.**

These metrics aim to quantify the complexity of software modules, potentially indicating areas prone to bugs or maintenance difficulties.

Du (Unique decision statements): The number of unique decision statements (e.g., if, else if, switch) in a module.

Dc (Decision statements): The total number of decision statements in a module, including duplicates.

DD (Destinations): The number of possible exit points from a module (e.g., return statements, break).

Du, Dc, and DD can be used to create a decision-dominator (DD) graph:

Nodes represent decision statements and exit points.

Edges connect decision statements to their possible next statements based on logic.

Analyze the graph for complexity visually.

Cyclomatic complexity (McCabe's complexity): A formula using Du to estimate the number of independent paths through a module's logic.

**Applications:**

Identify modules needing refactoring to improve maintainability and reduce bug risks.

Prioritize testing efforts towards more complex areas.

Estimate development and maintenance effort.

**Limitations**:

These metrics do not consider all aspects of code complexity like data structures or algorithms.

High complexity doesn't always guarantee bugs, and vice versa.

1. **Explain Boundary value analysis and robust case analysis of the triangle program/Next date program.**

Boundary value analysis (BVA): Tests input values at the edges of equivalence classes and around boundary conditions.

Triangle program example:

* Test valid triangle sides: 3, 4, 5 (minimum valid) and 99, 100, 101 (maximum valid).
* Test invalid triangle sides: 0, 1, 1 (all equal, not a triangle), 1, 2, 3 (degenerate - sum of two sides <= third side).

Robust case analysis (RCA): Tests input values that are slightly outside valid ranges or trigger unusual error conditions.

Next date program example:

* Test valid dates like 31/12/2023 and 28/02/2024.
* Test invalid dates like 32/12/2023 (out-of-range day) and 2024/02/30 (non-existent day).
* Test RCA cases like 29/02/2025 (leap year) and 01/01/1900 (century year).

1. **Explain equivalence class analysis of triangle program or next date program.**

Equivalence class analysis is a software testing technique that involves dividing valid and invalid input values into classes where any value within a class behaves similarly. This helps reduce the number of test cases by testing representative values from each class, ensuring comprehensive coverage while avoiding redundancy.

**Triangle Program**:

1. Identify Equivalence Classes:

* Class 1: Valid Triangles:
  + Subclass 1a: All sides positive and satisfy the triangle inequality (sum of any two sides > third side).
  + Subclass 1b: Degenerate triangle (sum of two sides exactly equals the third side; valid in some interpretations).
* Class 2: Invalid Triangles:
  + Subclass 2a: All sides equal (not a triangle).
  + Subclass 2b: Any side non-positive (invalid length).
  + Subclass 2c: Triangle inequality violated (sum of two sides <= third side).

**Next Date Program**:

1. Identify Equivalence Classes:

* Class 1: Valid Dates:
  + Subclass 1a: Within current month.
  + Subclass 1b: Last day of current month.
  + Subclass 1c: First day of next month.
  + Subclass 1d: Edge case - last day of a year.
* Class 2: Invalid Dates:
  + Subclass 2a: Out-of-range day.
  + Subclass 2b: Non-existent day (February 30th, etc.).
  + Subclass 2c: Negative date.
  + Subclass 2d: Future date exceeding limit.

1. **Find the Cyclomatic Complexity for the C program segment:-**

**while (first <= last)**

**{**

**if (array [middle] <search) first=middle+1;**

**else if (array [middle] == search) found = True;**

**else last = middle – 1; middle = (first + last)/2;**

**}**

**if (first < last) not Present = True;**

Identify decision statements: The program segment contains three decision statements:

if (array[middle] < search)

else if (array[middle] == search)

else (implicit else associated with the else if)

Apply McCabe's Complexity formula:

Cyclomatic complexity (M) = E - N + 2P

E = Number of edges (decision statements + 1): 4 edges (3 decision statements + 1 exit point)

N = Number of nodes: 5 nodes (4 decision points + 1 exit point)

P = Number of connected components: 1 connected component (the entire segment)

Calculate:

M = 4 - 5 + 2 \* 1

M = 5

Therefore, the cyclomatic complexity of the program segment is 5.

Important points:

Cyclomatic complexity measures the potential number of independent execution paths through a program segment.

A higher complexity indicates more paths and potentially more testing effort required.

While 5 is not a particularly high complexity, it still suggests multiple execution paths to consider in testing.

1. **Find the Cyclomatic Complexity for the following Code: while (m<n)**

**if (x>y ) and (a<b) then a=a+1**

**y=y-1 end if**

**m=m+1 end while**

**1. Code with Missing if Statement:**

Assuming the end if statement should have a matching if statement preceding it, the complete code would be:

*while (m < n)*

*if (x > y) and (a < b) then*

*a = a + 1*

*y = y - 1*

*end if*

*m = m + 1*

*end while*

In this case, the cyclomatic complexity would still be 0.

Explanation:

There's only one decision statement: the combined condition (x > y) and (a < b) in the if statement.

Even though the condition has two parts, they're both evaluated together, so there's only one true/false outcome.

There are no alternative execution paths (no else or else if), just the if block or the code outside it.

The while loop only affects the number of repetitions, not the decision paths within each iteration.

**2. Code with Unintentional end if:**

If the end if statement is unintended and not part of the actual code, then the code would be:

*while (m < n)*

*if (x > y) and (a < b) then*

*a = a + 1*

*y = y - 1*

*m = m + 1*

*end while*

In this case, the cyclomatic complexity would be 1.

**Explanation:**

Now the while loop condition and the combined condition in the if statement act as two independent decision points.

Each decision can be true or false, leading to four possible combinations of execution paths:

Both while and if conditions are true (executes a = a + 1, y = y - 1, and m = m + 1)

while true, if false (executes only m = m + 1)

while false, if true (not possible due to loop condition)

while false, if false (falls through to end of while loop)

1. **Consider the following method :**

**Find the test sets for Full Statement Coverage and Full Branch Coverage.**

* 1. **int f(int m, int n, boolean x, boolean y) 2. {**

**3. int res=0;**

**4. if(m<0) {res=n-m;}**

**5. else if(x || y) { 6. res= -1;**

**7. if(n==m) {res =1;}**

**8. }**

1. **else {res=n;}**
2. **return res;**
3. **} /\*end of f \*/**

Test Sets:

**Full Statement Coverage:**

To achieve full statement coverage, you need to execute every single statement in the method at least once. Here are the test sets that cover all statements:

* (m = -1, n = 2, x = false, y = false): Executes lines 3, 4 (true), 9, and 10.
* (m = 0, n = 2, x = false, y = true): Executes lines 3, 5 (true), 6, 9, and 10.
* (m = 0, n = 0, x = true, y = false): Executes lines 3, 5 (true), 6, 7 (true), 8, 9, and 10.
* (m = 2, n = 2, x = false, y = false): Executes lines 3, 5 (false), 9, and 10.

**Full Branch Coverage:**

To achieve full branch coverage, you need to execute every possible branch (true and false) of each conditional statement. Here are the test sets that cover all branches:

* (m = -1, n = 2, x = false, y = false): Covers the if (m < 0) branch (true) and the else if (x || y) branch (false).
* (m = 0, n = 2, x = false, y = true): Covers the if (m < 0) branch (false) and the else if (x || y) branch (true).
* (m = 0, n = 0, x = true, y = false): Covers the if (m < 0) branch (false), the else if (x || y) branch (true), and the if (n == m) branch (true).
* (m = 2, n = 2, x = false, y = false): Covers the if (m < 0) branch (false), the else if (x || y) branch (false), and the if (n == m) branch (true).

**Combined Test Sets:**

You can combine the test sets for full statement coverage and full branch coverage to obtain a minimal set:

* (m = -1, n = 2, x = false, y = false)
* (m = 0, n = 0, x = true, y = false)

These two test sets cover all statements and branches, making them a minimal test suite for both full statement coverage and full branch coverage.

1. **Write a program/main function in C/Java to enter a date, month and year and find out if it’s a valid date or not. Write test cases to for Full Code Coverage of the Code for the program you built.**

import java.util.Scanner;

public class ValidateDate {

public static boolean isLeapYear(int year) {

return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;

}

public static boolean isValidDate(int day, int month, int year) {

// Check year validity

if (year <= 0) {

return false;

}

// Check month validity

if (month < 1 || month > 12) {

return false;

}

// Check day validity based on month and leap year

int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int maxDays = daysInMonth[month - 1];

if (month == 2 && isLeapYear(year)) {

maxDays++;

}

return day > 0 && day <= maxDays;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter date (DD/MM/YYYY): ");

int day = scanner.nextInt();

int month = scanner.nextInt();

int year = scanner.nextInt();

if (isValidDate(day, month, year)) {

System.out.println("Valid date.");

} else {

System.out.println("Invalid date.");

}

}

}

**Explanation and Test Cases:**

Key Considerations:

* Year validity: Ensures years are positive integers.
* Month validity: Validates months between 1 and 12.
* Day validity: Checks day range based on month and potential leap year using a days\_in\_month array.
* Clear output: Indicates whether the date is valid or invalid.
* Comprehensive test cases: Cover all branches and boundary conditions:
  + Valid dates: January 1st, June 30th, February 29th (leap year), December 31st.
  + Invalid dates: February 30th (non-leap year), April 31st, negative day, month 0, year 0, future date.

1. **Write a test case in Junit and Test it using Junit framework and libraries. You can use the following tutorials regarding Junit from this links:-**
   1. [**https://www.javatpoint.com/junit-tutorial**](https://www.javatpoint.com/junit-tutorial)
   2. [**https://www.guru99.com/create-junit-test-suite.html**](https://www.guru99.com/create-junit-test-suite.html)
   3. [**https://www.tutorialspoint.com/junit/junit\_suite\_test.htm**](https://www.tutorialspoint.com/junit/junit_suite_test.htm)