SOFTWARE TESTING

UNIT-I [12 Hours]

Introduction: Basic definitions, A testing life cycle, Test Cases, The Traditional Model of Software Testing Process, Levels of Testing, Structural and Behavioral Insights, Fundamental approaches to apply Test Cases, Examples: The NextDate function, Triangle problem and The Commission Problem and The SATM (Simple Automatic Teller Machine) problem.

Boundary Value Testing: Generalizing Boundary Value Analysis, Limitations of BVA, Robustness Testing, Worst-case Testing, Special Value Testing, Test cases for Triangle Problem, Test cases for the NextDate function, Test cases for the Commission Problem, Random Testing and Guidelines for Boundary Value Testing.

What is Software Testing?

Software Testing is the process of evaluating and verifying that a software application or system meets the specified requirements and functions correctly. It involves identifying and fixing defects, ensuring the software performs efficiently under various conditions, and delivering a high-quality product to users.

OR

Software Testing is a method to assess the functionality of the software program. The software checks whether the actual software matches the expected requirements and ensures the software is bug-free. The purpose of software testing id to identify the errors, faults, or missing requirements in contrast to actual requirements. It mainly aims at measuring the specification, functionality, and performance of a software or an application.

Why Software Testing is Important? What are the benefits if Software Testing?

• Defects can be identified early: Software testing is important because if there are any bugs they be identified early and can be fixed before the delivery of the software.

• Improves quality of software: Software Testing uncovers the defects in the software, and fixing them improves the quality of the software.

• Increased customer satisfaction: Software testing ensures reliability, security, and high performance which results in saving time, costs, and customer satisfaction.

• Helps with scalability: Software testing type non-functional testing helps to identify the scalability issues and the point where an application might stop working.

• Cost-Effectiveness: After the application is launched it will be very difficult to trace and resolve the issues, as performing this activity will incur more costs and time. Thus, it is better to conduct software testing at regular intervals during software development.

• Reduced Risks: Testing helps identify and fix bugs early in development. This is crucial because fixing bugs later in the process is much more expensive and time-consuming. Early detection reduces the risk of project failure.

• Enhanced Security: Testing helps identify security vulnerabilities that could be exploited by attackers. This is especially important for software that handles sensitive data.

• Satisfied Customers: Well-tested software is less likely to crash, have bugs, or exhibit unexpected behavior. This leads to a better user experience and increased customer satisfaction.

• Improving Performance: Performance testing evaluates the speed, responsiveness, and stability of the software under different load conditions. By conducting performance testing, developers can optimize the software's performance and ensure it meets performance requirements.

• Compliance and Standards: Testing ensures that the software complies with industry standards regulations, and guidelines. It helps in meeting legal requirements, ensuring data privacy, and maintaining the integrity of the software.

Goals or Objectives of Software Testing

1. To demonstrate that the software meets its requirements and specifications.

2. To identify defects, errors, and bugs in the software and ensure that they are fixed before the software is released to end-users.

3. To ensure that the software is reliable, efficient, and user-friendly.

4. To improve the quality of the software and reduce the risk of failure or malfunction.

5. To ensure that the software is compatible with different hardware, software, and operating systems.

6. To ensure that the software is secure and protects sensitive data from unauthorized access.

7. To ensure that the software performs well under different conditions, such as high traffic, heavy load, or stress.

8. To ensure that the software is easy to maintain and update.

9. To ensure that the software is compliant with industry standards and regulations.

10. To ensure that the software meets the expectations and needs of end-users.

Classification of Software Testing

Software testing can be broadly classified based on its purpose, methodology, and execution. Here are the main classifications:

Based on Testing Objectives:

1. **Functional Testing**: This type of testing ensures that the software functions as intended and meets the specified functional requirements. Test cases are designed to validate the behavior of individual functions, features, and user interactions within the software application.

This testing type answers the question: "Does the software do what it's supposed to do?" It usually involves testing user interfaces, APIs, databases, security features, and any other component that contributes to the software's core functionality.

2. **Non-Functional Testing**: This evaluates the software's performance and other aspects like scalability, usability, reliability, and security.

- **Performance Testing**: Performance testing evaluates how the software performs under various conditions such as load, stress, and scalability. It aims to identify performance bottlenecks and ensure that the software meets performance expectations.
- **Security Testing**: Security testing focuses on identifying vulnerabilities and weaknesses in the software to prevent unauthorized access, data breaches, and other security threats.
- **Usability Testing**: Usability testing assesses the user-friendliness and overall user experience of the software. It involves testing the interface, navigation, and accessibility to ensure a positive user experience.
- **Compatibility Testing**: Compatibility testing checks the software's compatibility with different devices, browsers, operating systems, and environments to ensure seamless operation across various platforms.

3. **Regression Testing**: Conducted after changes, updates, or bug fixes, it ensures that the new code or modifications do not negatively impact the existing functionality of the software. It often involves re-running previous test cases to confirm stability.

Based on Testing Levels:

1. **Unit Testing** is the process of testing individual components, modules, or units of software to ensure that they perform as expected. It focuses on verifying the correctness of specific sections of code, typically the smallest testable parts of an application. Developers often conduct unit testing during the coding phase to identify and fix errors early in the development cycle.
2. **Integration Testing** focuses on verifying the interactions and data flow between integrated components or modules of a software system. It aims to uncover defects that may arise when individual units or modules are combined and ensures that they work together as expected.
3. **System Testing** is the process of testing an entire software system to verify that it meets the specified requirements and performs as expected in a complete and integrated environment. It is conducted after integration testing and focuses on evaluating the software as a whole, including both functional and non-functional aspects.
4. **Acceptance Testing** is the final stage of software testing, conducted to determine whether the developed software meets the business requirements and is ready for delivery to the client or end users. It often involves UAT by end-users to ensure that the software meets business needs.

Based on Testing Techniques:

Manual Testing: This is a testing process where testers manually execute test cases without using automated tools or scripts. It allows for exploratory testing, ad-hoc testing and detailed analysis of software behavior.

Automation Testing: This is a testing process that uses software tools or scripts to execute test cases automatically. It is primarily used for repetitive and extensive testing tasks to improve efficiency and accuracy. It is beneficial for Regression testing, Performance testing and test case execution.

Based on Testing Methods:

Black-Box Testing is a software testing method where the internal structure, design, or implementation of the application being tested is not known to the tester. The focus is solely on validating the functionality of the software based on its specifications, without considering how it operates internally.

White-Box Testing, also known as Clear-Box Testing, Glass-Box Testing, or Transparent-Box Testing, is a software testing method that examines the internal logic, structure, and code of an application. Testers need to have knowledge of the software's internal workings to design and execute test cases that covers different code paths and conditions.

Based on Testing Strategies:

Top-Down Testing is an approach to integration testing where testing begins from the highest-level modules and progressively moves downward to lower-level modules. This method is particularly useful for testing the system's control flow and interactions between hierarchical components.

Bottom-Up Testing is an integration testing approach where testing begins with the lower-level modules and progressively moves upward to the higher-level modules. This method ensures that each module is individually tested and verified before integrating with the rest of the system.

Verification and Validation

Meaning & Definition: Verification (Static Testing)

Verification is the process of discovering the possible failures in the software (not the actual final product) before the commencement of the testing phase. Generally, verification is done during the development phase of the software development life cycle. It involves reviews, inspections, meetings, code reviews, and specifications. Verification is done to determine whether software meet the specified requirements. It answers the question." Are we building the product right?".

Verification is the process, to ensure that whether we are building the product right i.e, to verify the requirements which we have and to verify whether we are developing the product accordingly or not. It's a Low-Level Activity: Verification is a static method of checking documents and files.

Verification Techniques or Static Testing Techniques: Below are the verification techniques

1. Reviews: "A review is a systematic examination of document by one or more people with the main aim of finding and removing errors early in the software development life cycle." There are two types of reviews held in verification. They are Formal Review and Informal Review.

(a) Formal Review: Formal reviews follow a formal process. It is well structured and regulated. It contains: Planning, Kick-off, Preparation, Review meeting, Rework.

(b) Informal Review: Informal reviews are applied many times during the early stages of the life cycle of the document. A two person team can conduct an informal review. The most important thing to keep in mind about the informal reviews is that they are not documented.

2. Inspection: It is the most formal form of reviews, a strategy adopted during static testing phase.

- It is the most formal review type.
- It is led by the trained moderators.
- During inspection the documents are prepared and checked thoroughly by the reviewers before the meeting.

3. Walkthrough: A walkthrough is an informal group or individual review method. In a walkthrough, the author describes and explains work product to his peers or supervisor in an informal meeting to receive feedback. The validity of the proposed work product solution is checked here. It is less expensive to make changes while the design is still on paper rather than during conversion. A walkthrough is a method of quality assurance that is static. Walkthroughs are casual gatherings with a purpose.

Meaning & Definition: Validation (Dynamic Testing)

Validation is the process of evaluating the final product to check whether the software meets the business needs. In simple words, the test execution which we do in our day to day life is actually the validation activity Testing which includes smoke testing, functional testing, regression testing, systems testing, etc.

Validation occurs after the verification process and the actual testing of the product happens at a later stage Defects which occur due to discrepancies in functionality and specifications are detected in this phase. It answers the question, "Are we building the right product?"

Validation is the process, whether we are building the right product i.e., to validate the product which we have developed is right or not.

Activities involved in this is testing the software application. In simple words, Validation is to validate the actual and expected output of the software.

Difference between Verification and Validation

Verification	Validation
Evaluates the intermediary products to check whether it meets the specific requirements of the particular phase.	t the final product to check whether it meets the business needs.
Checks whether the product is built as per the specified requirement and design specification.	It determines whether the software is fit for use and satisfies the business needs.
Checks "Are we building the product right?	Checks "Are we building the right product"?
This is done without executing the software.	Is done with executing the software.
Involves all the static testing techniques.	Includes all the dynamic testing techniques.
Examples include reviews, inspection, and walkthrough	Examples include all types of testing like smoke testing, regression, functional testing, systems and User Acceptance Testing.

Basic Definitions

Error: Errors are mistakes made by people during the software development process. When these mistakes occur while coding, they are commonly d referred to as bugs. Errors can escalate from requirements to design and further to coding stages.

Example: Suppose during the development of the se e-commerce website, a developer mistakenly codes the checkout process to deduct the wrong amount from the customer's account. This coding mistake is an error.

Fault: A fault is the manifestation of an error in the software. It represents the error in the code. Think of a fault as the result of an error made during development. Faults can be elusive and can be categorized as faults of commission (incorrect information entered) or faults of omission (missing information).

Example: The fault in this scenario is the representation of the error in the code. The e incorrect deduction of the amount from the customer's account due to the developer's error is the fault in the system.

Failure: A failure happens when the code associated with a fault is executed during the software operation. Failures are observable and indicate that something went wrong during the execution of the software. Failures are typically linked to faults of commission.

Example: When a customer proceeds to checkout and the system deducts an incorrect amount from their account, it results in a failure. The failure occurs when the faulty code executes during the checkout process.

Incident: An incident is the visible symptom of a failure that alerts users, customers, or testers to the presence of a failure. It is the outward indication that something has gone wrong in the software.

Example: The incident would be when the customer notices that the amount deducted from their account is incorrect. The customer reporting the discrepancy is the incident that alerts the system to the failure.

Test: Testing involves identifying errors, faults, failures, and incidents. A test is the process of executing software with test cases. The primary goals of testing are to find failures or to demonstrate correct execution.

Example: To identify and rectify such errors, faults, and failures, testing is essential. Testing the checkout process with various scenarios to ensure correct deductions is crucial. The act of testing the checkout process is the test.

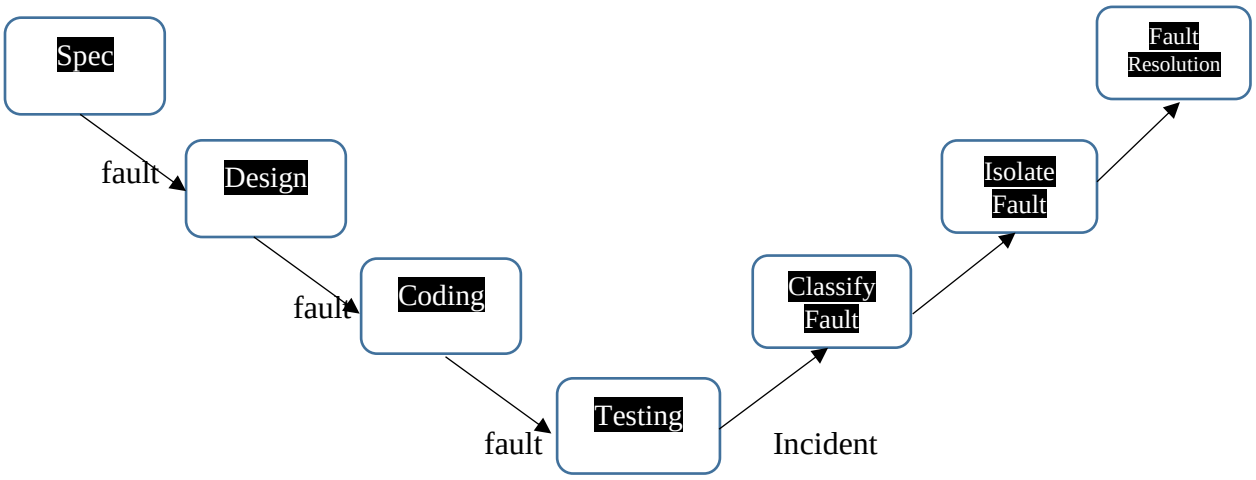
Test Case: A test case is a specific test scenario with defined inputs and expected outputs associated with a Test Case particular program behavior. Test cases help in systematically validating the functionality of the software.

Example: A test case for this scenario could involve a customer placing an order, proceeding to checkout, and verifying that the correct amount is deducted from their account. The test case would include specific inputs (order details, payment information) and the expected output (correct deduction amount).

A Testing Life Cycle

The Testing Life Cycle is a critical framework in software development that systematically identifies diagnoses, and resolves issues within a software product to ensure quality and reliability. By following a structured approach, teams can effectively address defects, enhance functionality, and meet specified requirements. This disciplined process helps prevent defects from reaching production, reducing costs and increasing user satisfaction.

The Testing Life Cycle in the below diagram illustrates the various stages involved in identifying and resolving issues during software development. This cycle is an integral part of ensuring software quality through systematic testing and issue management.



1. Specification (Spec): The testing cycle begins at the specification phase, where requirements are defined and documented. It is crucial to have a clear, thorough, and unambiguous specification because errors at this stage can propagate through to later stages, leading to more severe issues.

2. Design: During the design phase, the specifications are transformed into a design plan that outlines the software architecture and its components. Faults can arise here if the design does not accurately or efficiently implement the specifications. Such design faults could be logical errors in flow or inefficient architecture choices.

3. Coding: In the coding phase, developers write the actual code based on the design documents. This stage is prone to introducing faults due to human error in implementing the design logic, misunderstanding requirements, or syntactical errors in the code.

4. Testing: Once coding is completed, the testing phase begins to check the software for defects (faults) and ensure that it performs as expected. The goal is to identify and document any incidents arising from faults in the software.

5. Classify Fault: When a fault causes an incident, the issue is analyzed and classified. This involves determining the nature of the fault, such as whether it's due to a requirement misunderstanding, design error, or coding mistake. Classifying incidents helps in prioritizing them for fixes.

6. Isolate Fault: Once an incident is reported and classified, the next step is to isolate the specific part of the code or design that is causing the problem. This precise identification is critical for effectively addressing the fault without introducing new issues,

7. Fault Resolution: The final phase in the testing cycle is resolving the fault. This involves modifying the code or design to fix the issue and verifying that the fix resolves the problem without causing additional problems. This stage is crucial as inadequately resolved faults can lead to further errors or even new faults.

Additional Considerations:

• Fault Propagation: Faults in early stages like design can propagate and manifest as more significant problems in later stages like coding and testing. Each phase builds upon the previous one, so errors can become compounded.

• Regression Testing: After resolving faults, regression testing is performed. This ensures that the changes made to fix faults do not adversely affect other parts of the software that were previously working correctly. It's vital to ensure that a fix doesn't lead to new, unexpected behavior or degrade the software's performance.

This testing life cycle not only helps in making the software more reliable and efficient but also structures the testing process to be as thorough as possible. Proper management and execution of each phase are crucial for minimizing the risk of high-severity incidents in production environments, ultimately leading to a higher quality software product.

Test Cases

A test case can be defined as: "A documented set of inputs, execution conditions, and expected results used to verify that a particular feature of a software application is functioning correctly."

Components of a Test Case

Components of a Test Case

A well-structured test case is essential for effective software testing. Each component of a test case plays a crucial role in ensuring the test is executable, measurable, and repeatable. The key components of a test case are listed below.

1. Test Case ID: A unique identifier assigned to each test case. It helps in tracking the test case in test management tools or documentation and referencing specific tests easily.

2. Purpose: A brief description or statement about what the test case is intended to verify or validate.

3. Test Created By: The name of the person who created the test case. This is useful for accountability and for seeking clarification if there are any questions regarding the test case specifics.

4. Test Environment: The specific setup required to run the test, including hardware, operating system, network configurations, and any installed applications.

5. Prerequisites: Any conditions that must be met or steps that need to be performed before the test can be executed. This might include configurations, data setups, or previous test completions.

6. Test Procedure: A detailed step-by-step guide on how to execute the test. It includes actions for setting up, executing, and shutting down the test. It provides clear instructions on how to conduct the test to ensure consistency in test execution and results, regardless of the tester.

7. Test Data: Specific data values or inputs that need to be used during testing.

8. Expected Result: The outcome that should occur if the software functions correctly under the test conditions.

9. Actual Result: What actually happened when the test was executed? The documented outcome that is compared with the expected result to determine software performance and correctness.

10. Verdict: Pass/Fail: The outcome of the test case based on the comparison between expected and actual results.

11. Comments: Additional notes or observations made during the testing, explanations of the results, or issues encountered that might not be covered by the verdict.

Sample Test Case Format

Test Case ID	**Purpose**	**Test Created By**	**Environment**	**Pre-requisites**	**Test Procedure**	**Test Data**	**Expected Result**	**Actual Result**	**Verdict Pass/ Fail**	**Comments**
Serial no assigned to test case	Brief idea about case	Name of test creator	Software or hardware in which the test case is executed	Conditions that should be fulfilled before the test is performed	Steps to be performed in test	Inputs, variables and data	What the program should do	What is actually done	Status of the test	Notes on the procedure

Example of a test case for a Login Functionality

Test Case ID: TC-LOGIN-001

Purpose: Verify login functionality with valid credentials.

Test Created By: John Doe

Environment: Windows 10, Chrome Browser (v.96.0), Web application deployed on staging server.

Pre-Requisites: User account must exist with valid credentials. Access to staging environment.

Test Procedure:

1. Navigate to the login page.
2. Enter valid email (testuser@example.com).
3. Enter valid password (password123).
4. Click the "Login" button.

Test Data - Username: testuser@example.com | Password: password123

Expected Result: The user should be successfully redirected to the dashboard without any errors.

Actual Result: The user was successfully redirected to the dashboard without any errors.

Verdict (Pass/Fail): Pass

Comments: Test executed successfully. No issues observed.

****Note: Preferably write the example in the tabular format itself in the examination.**

What is Test Suite?

A test suite is a collection of test cases. In automated testing, it can mean a collection of test scripts. In a test suite, the test cases / scripts are organized in a logical order. For example, the test case for registration will precede the test case for login.

When we have hundreds / thousands of test cases, a test suite allows to categorize them in a way that matches our planning or analysis needs. For example, we could have a test suite for each of the core features of the software or we could have a separate test suite for a particular type of testing (for example, smoke test suite or security test suite).

An example of a test suite for purchasing a product could comprise of the following test cases:

Test Case 1: Login

Test Case 2: Add Products

Test Case 3: Checkout

Test Case 4: Logout

Note that each of the test cases above are dependent on the success of the previous test cases. For instance, it's no use checking out if one cannot add products. Hence, if we are running a test suite in sequential mode, we can choose to stop the test suite execution if a single test case does not pass.

Process of Executing a Test Case

The execution of a test case involves several steps:

1. Establishing Preconditions: Setting up the system or environment to meet the conditions under which the test should be run.

2. Providing Inputs: Inputting the defined data or actions into the system.

3. Observing Outputs: Monitoring and recording the system's response to the inputs.

4. Comparing Expected vs. Actual Results: Evaluating whether the actual outputs match the expected outputs.

Insights from a Venn Diagram

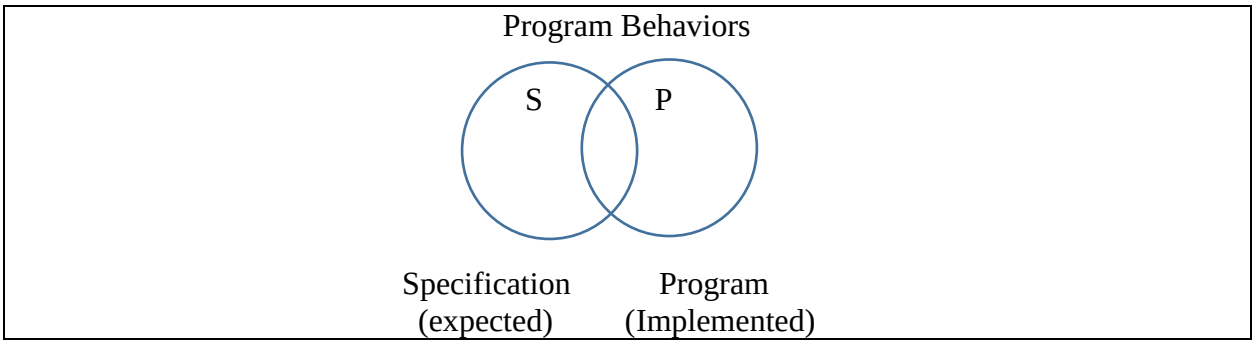
The use of Venn diagrams in software testing indeed provides a visual representation of th relationships between specified, implemented, and tested behaviors in software development. B illustrating the intersections and differences between these aspects, Venn diagrams offer a clear and concise way to analyze the completeness and effectiveness of testing strategies.

Through Venn diagrams, software testers can identify areas where specified behaviors have not been implemented or tested, programmed behaviors that are not covered by test cases, and test case that correspond to unspecified behaviors. This visualization helps in pinpointing potential gaps in testing coverage and understanding the alignment between what is specified, what is implemented and what is actually tested.

This visual tool aids in improving the overall quality and reliability of software systems by providing a structured approach to evaluating the testing completeness and correctness.

Example: Understanding Specified and Implemented Behaviors through Venn Diagrams

Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. The below figure 1.2 shows the relationship between the specified and implemented programmed behaviors.



• Specified Behaviors (S): These represent the behaviors expected from the software, as defined in the requirements or design specifications.

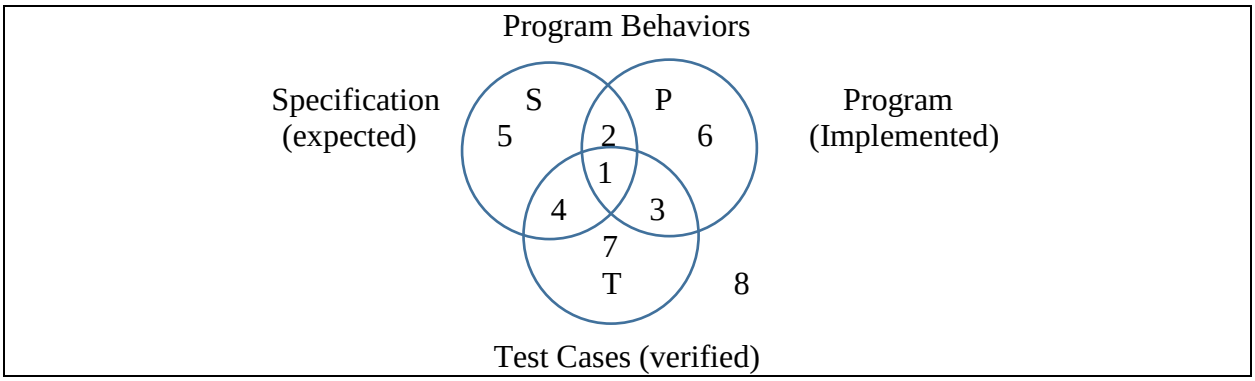
• Implemented Behaviors (P): These are the behaviors actually coded or implemented into the software.

• Detailed Analysis:

- S n P: Behaviors that are both specified and correctly implemented. This is the ideal scenario where implementation aligns perfectly with the specifications.
- S-P: Specified behaviors that are not implemented, known as faults of omission. These gaps highlight deficiencies in the transition from specification to coding
- P-S: Behaviors that are implemented but were not specified, known as faults of commission. These may include additional features added by developers or misinterpretations of the specifications.

Example - Specified, Implemented and Tested Behaviors through Venn Diagrams

Given a program and its specification, consider the set S of specified behaviors, the set P of programmed behaviors and the set T of tested behaviors. The below figure shows the relationship between the specified, implemented and tested programmed behaviors.



• Specified Behaviors (S): These represent the behaviors expected from the software, as defined in the requirements or design specifications.

• Implemented Behaviors (P): These are the behaviors actually coded or implemented into the software.

• Tested Behaviors (T): These are the behaviors that are actually tested.

• Detailed Analysis:

Region 1 (SnPnT): Represents the ideal scenario where behaviors are specified, implemented, and verified by test cases. This is the target region where testing efforts are often focused to ensure that specified and implemented behaviors are correctly tested.

Region 2 (SnP-T): Behaviors that are specified and implemented but not tested. This indicates gaps in the testing coverage.

Region 3 (P-SnT): Behaviors that are implemented and tested but were not specified. This might highlight extra features or potential deviations from the intended design that are being verified.

Region 4 (S-Pn T): Specifies test cases developed for behaviors that were specified but not implemented. This can often occur when tests are designed based on specifications without verifying whether the implementation has been done accordingly.

Region 5 (Only S): Specified behaviors that are neither implemented nor tested. These are missed opportunities or oversights in both development and testing.

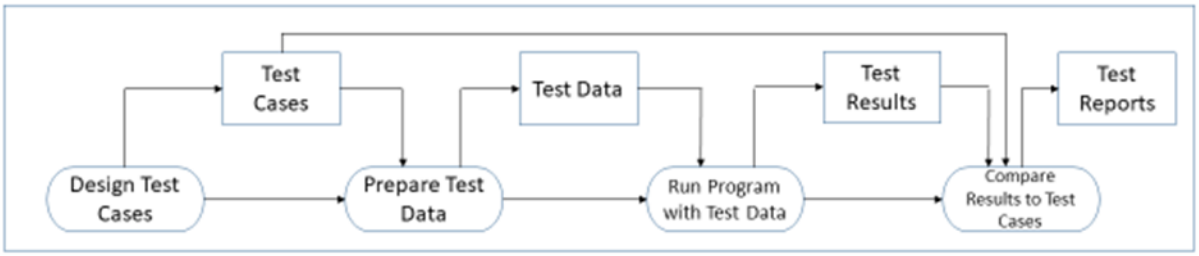
Region 6 (Only P): Implemented behaviors that are neither specified nor tested, indicating possible rogue features or unexpected behaviors that might lead to issues.

Region 7 (Only T): Test cases that do not correlate with any specified or implemented behaviors. These could be erroneous tests or tests designed for functionalities that were removed or never implemented.

Region 8 (P-T): Implemented behaviors that are not tested, suggesting a lack of adequate testing or oversight in ensuring the program's integrity.

These diagrams help to visually organize the complex relationships between what is expected (specified),| what is done (implemented), and what is verified (tested). They assist in identifying discrepancies between these aspects, enabling teams to focus on areas needing improvement, such as enhancing test coverage. Rectifying implementation errors, or updating specifications to reflect the actual system behavior.

Traditional Model of the Software Testing Process



The model comprises the following key stages:

1. Design Test Cases: In this initial phase, specific scenarios or "test cases" are created base on the software's requirements and design documentation. These test cases are designed include typical use cases as well as edge cases that may reveal bugs or inconsistencies.

2. Prepare Test Data: Once the test cases are established, the next step involves preparing appropriate test data. This data is essential for simulating real-world conditions and input that the software will encounter.

3. Run Program with Test Data: The actual testing phase commences with running the software using the prepared test data. The software's behavior and outputs are recorded for subsequent analysis. This step is crucial for identifying any disparities between the expected and actual software performance.

4. Compare Results to Test Cases: The results obtained from running the tests are compare to the expected outcomes outlined in the test cases. Any deviations indicate potential issue that require attention. This comparison is instrumental in verifying that the software meet the specified requirements and behaves as intended.

5. Test Reports: Finally, the outcomes of the testing process are documented in test reports These reports encompass details of the tests performed, the results obtained, and any bugs a problems identified. They serve as critical resources for developers to understand the aspect of the software that require correction.

Levels of Testing

Levels of testing refer to the different stages or phases of software testing that are conducted to ensure the quality and functionality of a software product. The levels of testing are organized hierarchically with each level focusing on specific aspects of the software. The common levels of testing include:

Unit Testing

What is Unit Testing?

1. Unit testing is a software testing technique that involves testing individual units or components or modules of a software application. A unit can be a single function, method, or class. The goal of unit testing is to ensure that each unit functions as expected and meets the specified requirements.

2. Unit testing is typically done by developers during the coding phase of software development. The tests are automated and run frequently to catch errors early in the development process.

Examples Unit Testing

1. Login Feature: Consider an example of testing a login feature in a web application. The test cases would include verifying that the login page is displayed correctly, checking that the user can enter their credentials, and ensuring that the system validates the credentials and logs the user in successfully. Additionally, the test cases would cover scenarios such as incorrect login credentials, expired sessions, and other potential errors that could occur during the login process. By testing each component of the login feature in isolation and then testing the interaction of these components in sequence, developers can ensure that the login feature functions as intended and meets its specified requirements.

2. Messaging Feature: Consider an example of testing a messaging feature in a social media application. The test cases would include verifying that users can compose and send messages, ensuring that messages are delivered to the intended recipients, and checking that the system handles various scenarios such as message formatting, attachments, and error handling for failed message deliveries. By testing each aspect of the messaging feature in isolation and then testing the interaction of these components in sequence, developers can ensure that the messaging feature operates as expected and can handle a variety of user interactions and potential error conditions.

Integration Testing

What is Integration Testing?

Integration testing involves testing the interactions between integrated units or components to ensure they work together as expected. The goal is to detect interface defects and ensure that the integrated components function correctly as a whole. It involves verifying the behavior of the composite component as a whole including its interfaces, interactions, and overall functionality.

Examples Integration Testing

1. Login Feature: Integration testing for the login feature involves testing each individual component while emphasizing the interfaces between them. For instance, when testing the login page component it is essential to verify that the page displays correctly and that the user can enter their credentials. Additionally, the interaction with the user credentials validation component should be tested to ensure that the entered credentials are correctly passed for validation. This emphasizes the interface between the login page and the user credentials validation component. Similarly, testing the session management component should focus on the interface with the validated credentials to initiate and manage user sessions, including handling session timeouts and termination.

2. Messaging Feature: Component testing for the messaging feature involves testing each individual component with a focus on the interfaces between them. When testing the message composition component, it is crucial to verify that users can compose messages with various formatting options and attachments, emphasizing the interface between the user interface and the message composition logic. Testing the message delivery component should emphasize the interface with the message composition component to ensure that composed messages are correctly delivered to the intended recipients.

What is System Testing?

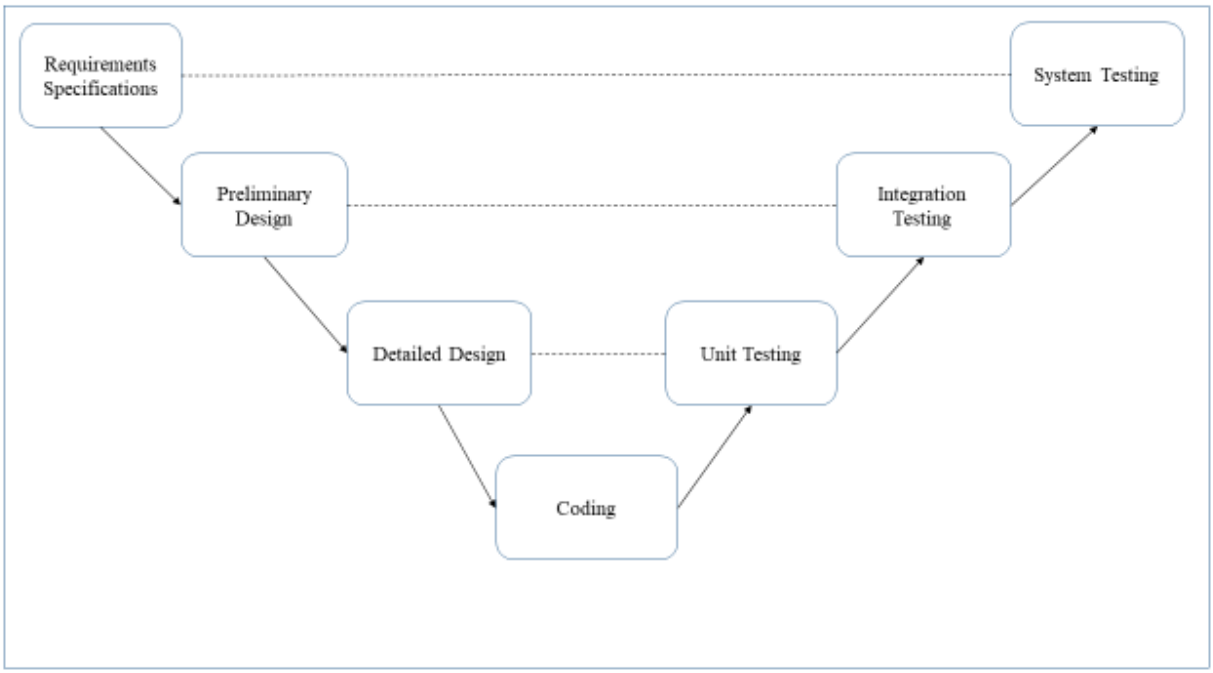
System testing is the phase in the software testing process where the complete and integrated software system is evaluated to ensure that it meets specified requirements. This testing phase focuses on verifying that the system functions correctly as a whole and that it performs according to the specified requirements and design. System testing is typically performed after integration testing and before acceptance testing.

Example System Testing

E-commerce system testing involves testing the entire e-commerce system to ensure that it meets the specified requirements and functions as expected in a real-world environment. This includes testing the system's functionality, performance, reliability, security, and usability. The system is evaluated for its ability to handle various scenarios, such as user registration, product search, adding items to the shopping cart, placing orders, payment processing, and order fulfillment. Integration testing is also performed to ensure that all the components of the system work together seamlessly. By conducting system testing for the e-commerce system, we can ensure that it provides a seamless and secure shopping experience for its users, meets the business and technical requirements, and performs as expected in the real-world environment.

Levels of Testing in V-Model

The V-Model is a variation of the traditional waterfall model and it enhances the alignment of development and testing activities by mirroring each phase of software development with a Corresponding testing phase. This structured approach helps in identifying and executing tests that are particularly relevant to each stage of the software's creation. The below figure illustrates the levels of abstraction and testing in V-Model.



Specification-based testing occurs at three distinct levels which correspond to the different stages software development:

1. System Testing: This is aligned with the "Requirements Specification" phase, system testing is designed to validate the software against the overall business requirements. It is a high level test to ensure that all business processes are functioning as intended, and the software behaves as an integrated whole.

2. Integration Testing: This corresponds to the "Preliminary Design" phase. Integration tests are focused on the interactions between integrated units/modules to detect interface defects. This type of testing is crucial when various software modules are being developed concurrently by different teams, and it ensures that these modules operate together correctly.

3. Unit Testing: This is linked to the "Detailed Design" phase. This involves testing the smallest testable parts of the software, typically individual functions or methods. Unit testing is often conducted by developers themselves and is aimed at ensuring that each function performs a designed.

Structural and Behavioral Insights

Structural and behavioral insights are essential aspects of software testing that focus on different aspects of the software system.

1. Structural Insights:

- ◊ Definition: Structural insights in software testing refer to understanding the internal structure of the software, including the code, components, modules, and their interactions.
- ◊ Focus: It emphasizes analyzing the software at a lower level, such as individual units, classes, and methods, to ensure that the code functions correctly and adheres to design specifications.
- ◊ Techniques: Structural testing techniques such as code based testing or white-box testing focus on examining the internal logic and structure of the software to design test cases that exercise specific paths and conditions within the code.
- ◊ Goal: The goal of structural insights is to verify the correctness of the code implementation, identify defects in the logic, and ensure that the software behaves as expected based on its internal design.

2. Behavioral Insights:

- ◊ Definition: Behavioral insights in software testing involve understanding how the software behaves in response to different inputs, user interactions, and system conditions.
- ◊ Focus: It concentrates on the external behavior of the software, including its functionality, performance, usability, and compliance with requirements.
- ◊ Techniques: Behavioral testing techniques such as specification based testing or black-box testing focus on testing the software based on its external specifications and requirements without knowledge of the internal code structure.
- ◊ Goal: The goal of behavioral insights is to validate that the software meets user expectations, functions correctly in different scenarios, and delivers the intended outcomes as specified in the requirements.

Fundamental Approaches to Apply Test Cases

The concept of identifying test cases in software testing can be effectively illustrated using various methods and perspectives. The two primary approaches to test case identification are:

1. Specification-Based Testing (Functional Testing or Black Box Testing)

2. Code-Based Testing (Structural Testing or White Box Testing)

Definition: Specification-Based Testing (or) Functional Testing (or) Black Box Testing Specification-based testing is the process of testing a software in accordance with pre-determined requirement or specifications. It is a testing technique that tests the functionality of a software application without knowing the internal structure of the code or how it has been implemented. Specification-based testing (function testing) is only concerned with validating if a system works as intended.

It is also called "BlackBox" because software is like a black box inside which tester cannot se The main purpose of Black Box testing is to check whether the software is working as expecte and meeting the customer requirements or not. It was designed as a method of analyzing client requirements, specifications, and high-level design strategies.

Example: In case of Google or any other search engine, the user enters text in the browser. The search engine locates and retrieves the information. The user is not aware of the how Google retrieves the information.

Specification-based testing is the common starting point for designing test cases. Functional test case design can (and should) begin with requirements specification and continue through design and interface specification; it's the only technique with such wide and early applicability Functional testing methodologies can be applied to any description of program behavior, from a informal partial description to a formal specification, from module to system testing. Functional test are cheaper to design and run than white-box tests.

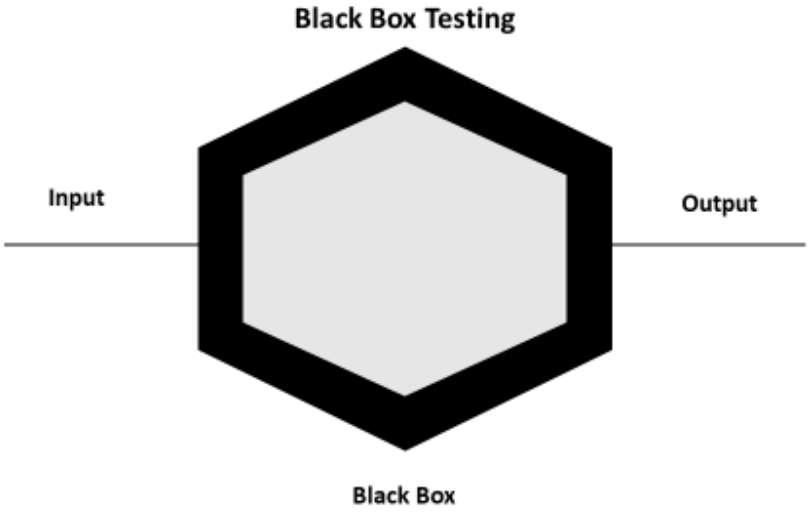
What to test in Specification Based Testing?

The main objective of functional testing is checking the functionality of the software system. It concentrates on:

- Functional Testing involves the usability testing of the system. It checks whether a user can navigate freely without any difficulty through screens.
- Functional testing test the accessibility of the function.
- It focuses on testing the main feature.
- Functional testing is used to check the error condition. It checks whether the error message displayed.

Example: Specification Based Testing

Testing search engine is a good example for specification based or functional testing. We are not aware of the processes that work behind the search engine to provide the desired information. While testing search engine we provide input in the form of words or characters, and check for output parameters such as relevance of the search result, time taken to perform the search or the order of listing the search result.



Characteristics of Specification Based Testing

1. It focuses on testing the software system from the outside, without knowledge of how the internal code or system architecture works.

2. It is a functional testing method that verifies whether the software system meets the functional requirements specified by the customer or user.

3. It involves testing the software system based on its inputs and outputs without examining the internal workings of the system.

4. It is user-oriented and focuses on ensuring that the software system works as expected from the perspective of the end-user.

5. It is an independent testing method that can be performed by testers who have no knowledge of the programming language or platform used to develop the system.

6. It is based on the requirements and specifications provided by the customer or user, and the test cases are designed to cover all possible scenarios and inputs that the user might encounter while using the system.

7. It can identify defects or errors that may have been missed during the design or coding phase of the software development life cycle.

8. It can be performed manually or using automated tools, depending on the complexity of the software being tested and available resources.

9. It is often performed during later stages of testing, after code based testing or white-box testing has been completed.

10. It can ensure that the software meets regulatory or compliance requirements by verifying that it behaves as expected under different conditions.

Definition: Code Based Testing (or) Structural Testing (or) White Box Texting

Code Based Testing can be defined as a type of software testing that tests the code's structure and intended flows. For example, verifying the actual code for aspects like the correct implementation of conditional statements, and whether every statement in the code is correctly executed. It is also known as Structural Testing (or) White Box testing or Glass Box testing. This type of testing requires knowledge of the code, usually done by the developers.

In simple words, Structural testing is the type of testing carried out to test the structure of code.

Is Structural Testing White Box?

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing).

Characteristics of Code Based Testing

1. It is focused on the internal workings of the software application, including its cod architecture, and design.
2. It is based on how the system carries out the operations instead of how it is perceived by the users or how functions are carried out.
3. It involves testing all possible paths through the software application to ensure that every line of code is executed at least once during testing. This ensures that errors related to control flow structures such as loops and conditional statements are identified and corrected.
4. It requires technical expertise in software development and coding to be carried out effectively. Developers must have a deep understanding of the software's internal workings to identify potential issues and defects.
5. It can be automated using tools such as unit test frameworks or code coverage tools to ensure thorough test coverage. This can help reduce the time and effort required for manual testing.
6. It provides better coverage than many other types of testing approaches because it tests the whole code in detail, ensuring that errors involved can easily be removed. The chances missing out on any error become very low.
7. It is particularly useful for identifying complex defects that may be difficult to detect using other types of testing approaches.
8. It can be time-consuming because it involves testing all possible paths through the software application. This can make it difficult to achieve complete test coverage in a reasonable amount of time.
9. It may require specialized tools such as code coverage analysis tools or static analysis tools to identify potential issues and defects in the software application. These tools can help automat some aspects of structural testing and make it more efficient.
10. It complements other types of testing approaches such as functional testing and integration testing by providing additional coverage of the internal workings of the software application.

Examples

Triangle Problem

The Triangle Problem is a famous example in software testing. It shows how important it is to test software carefully. Many experts have used this example to explain different testing methods and why it's crucial to have clear instructions when developing software.

> Problem Statement Simple Version:

The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides Equilateral, Isosceles, Scalene, or NotATriangle.

The program's task is to determine the type of triangle based on these sides: Equilateral (all sides equal), Isosceles (exactly one pair of sides equal), Scalene (no pair of sides equal), or NotATriangle (conditions not met).

> Improved Version:

The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

Condition c1: $1 \leq a \leq 200$

Condition c2: $1 \leq b \leq 200$

Condition c3: $1 \leq c \leq 200$

Condition c4: $a < b + c$

Condition c5: $b < a + c$

Condition c6: $c < a + b$

In improved version, specific conditions are set for the input values a, b, and c. These conditions require that a, b, and c fall within the range of 1 to 200. If the input values meet these criteria, the program proceeds to classify the triangle type based on the defined rules. If any input value fails to satisfy the specified conditions, the program provides an output message indicating the issue encountered. For example, "Value of b is not in the range of permitted values."

If values of a, b, and c satisfy conditions c4, c5, and c6, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.

2. If exactly one pair of sides is equal, the program output is Isosceles.

3. If no pair of sides is equal, the program output is Scalene.

4. If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

Discussion:

The Triangle Problem is popular because it's easy to understand but also has some tricky parts. It shows why it's important for everyone involved in making software to have clear instructions. The idea of the triangle inequality rule, which says the sum of two sides must be greater than the third side, adds a bit of complexity. Choosing 200 as the highest number is just a convenient way to test different scenarios. By setting boundaries for the input values, developers can create test cases that cover a wide range of possibilities to ensure thorough testing of the program's functionality under different conditions.

What are the Complexities of Testing a Triangle Problem?

Testing a Triangle Problem can involve several complexities that need to be addressed to ensure the accuracy and reliability of the software. Some complexities in testing a Triangle Problem are:

1. Boundary Testing: Testing the program with boundary values such as the minimum and maximum allowed side lengths can be complex. Ensuring that the program behaves correctly at these boundaries is crucial for comprehensive testing.

2. Handling Invalid Inputs: Testing how the program handles invalid inputs such as negative side lengths or non-numeric values is essential. Verifying that the program provides appropriate error messages and handles exceptions correctly adds complexity to testing.

3. Validation of Triangle Inequality: The triangle inequality theorem states that the sum of the lengths of any two sides must be greater than the length of the third side. This must be validated under all input conditions to ensure that this condition is checked correctly and consistently for all side lengths.

4. Edge Case Testing: Testing edge cases such as triangles with very small or very large side lengths, is crucial. These edge cases can reveal potential issues in the program's logic or calculations. Testing need to ensure that the program behaves as expected in these scenarios adds complexity to testing.

5. Complex Logic Verification: The logic involved in determining the type of triangle based on the side lengths, including considerations like the triangle inequality rule, can introduce complexities in testing. Validating the correctness of the logic and ensuring all possible outcomes are covered require thorough testing strategies.

The NextDate Function

The NextDate function is designed to handle the calculation of the date following a given input date by considering three variables: month, day, and year. This function showcases a specific type of complexity related to logical relationships among input variables. It is distinct from the complexity observed in the Triangle Problem. It highlights the logical relationships between days, months, a years, where each input variable (month, day, year) influences the output in distinct ways. This function serves as a clear example of how a simple and straight forward tasks can become complex due to the various rules and exception in calendar calculations.

Problem Statement:

The NextDate function operates on three integer variables with specific ranges:

- month: Valid values range from 1 to 12.
- day: Acceptable values are between 1 and 31.
- year: The year should fall within the accepted range like (1812 to 2024).

The NextDate function must handle invalid input values and combinations. For example, an input like "June 31" would be considered invalid due to June having only 30 days. In case any of the above conditions (month, day, year) are violated, NextDate signals an out-of-range error specific to that variable, such as "Value of month not in the range 1...12." For invalid date combinations, a generic message "Invalid Input Date" is provided.

Discussion

The complexity of the NextDate function arises from two main aspects:

- Complex Input Domain: Managing a wide range of input values and their valid combinations poses a significant challenge. It require a thorough validation to ensure accurate date calculations. Ut
- Leap Year Calculation: The function must accurately handle leap years, which introduce an extra day in February every four years, with exceptions for certain century years. This leap year rule is crucial for determining the correct date transition, especially in February.

The implementation of the function reflects this complexity in two key areas:

- Leap Year Handling: A considerable portion of the code is dedicated to precisely determining leap years to ensure accurate date calculations, particularly in February.
- Input Validation: Substantial code is allocated to validate input values, checking for out- of-range errors and incorrect day-month combinations. This validation is essential for the function to provide correct results.

What are the Complexities of Testing the NextDate Function?

Testing the NextDate function involves several complexities due to the nature of date calculations and the specific requirements of the function. Some key complexities in testing the NextDate function are:

1. Input Domain Complexity: The NextDate function operates within specific ranges for month, day, and year inputs. Testing all possible combinations within these ranges can be challenging and time- consuming, especially considering edge cases and boundary conditions.

2. Leap Year Handling: Testing the function's behavior around leap years adds complexity. Ensuring that the function correctly identifies leap years and adjusts the date calculation accordingly requires thorough testing to cover all scenarios, including leap day (February 29th) considerations.

3. Invalid Input Scenarios: Testing for invalid inputs, such as providing a day beyond the valid range for a specific month or entering an incorrect month number, requires comprehensive test cases to validate the function's error-handling mechanisms.

4. Boundary Testing: Testing at the boundaries of the input ranges (e.g., the last day of a month, the last month of the year) is crucial to verify the function's accuracy in handling critical transition points.

5. Combination Testing: Verifying the function's behavior for various combinations of valid and invalid inputs adds complexity. Testing scenarios where multiple input variables interact to determine the output date is essential to ensure comprehensive coverage.

6. Output Verification: Validating the correctness of the output date generated by the NextDate function against expected results for a wide range of input scenarios is a key aspect of testing complexity.

7. Error Handling: Testing the function's ability to handle errors gracefully, such as providing informative messages for invalid inputs or exceptional cases, requires thorough testing to ensure robust error management.

The Commission Problem

The Commission Problem involves complex computational and decision-making elements. It is commercial computing scenarios particularly in management information systems (MIS). This scenario sets the stage for applying advanced software testing techniques like data flow and slice-based testing.

Problem Statement:

The Commission Problem involves a scenario where a salesperson sells rifle components (locks, stocks, and barrels) manufactured by a gunsmith in Missouri. The problem statement includes the following key elements:

Product Costs:

- Locks cost $45 each.
- Stocks cost $30 each,
- Barrels cost $25 each.

Sales Requirements:

- The salesperson must sell at least one lock, one stock, and one barrel each month, but they do not necessarily need to be sold as part of a complete rifle.
- There are maximum sales limits due to production constraints: 70 locks, 80 stocks, and 90 barrels per month.

Sales Reporting:

- After visiting each town, the salesperson sends a telegram to the gunsmith about the number of locks, stocks, and barrels sold.
- At the end of the month, a final telegram with the figures "-1 locks sold" signals the completion of that month's sales, prompting the gunsmith to compute the salesperson's commission.

Commission Calculation:

The commission structure is tiered:

- 10% commission on sales up to and including $1000.
- 15% commission on the next $800 of sales.
- 20% commission on any sales beyond $1800.

The Commission Problem revolves around managing sales of rifle components, ensuring minimum sales requirements are met, reporting sales data accurately, and calculating the salesperson's commission based on a tiered structure of sales revenue.

> Discussion

This problem highlight the computational and logical aspects involved in processing sales data and calculating commissions. The use of a sentinel value (-1 locks sold) in the communication process is a classic technique in MIS for signaling the end of data input.

Components of the Problem:

- Input Data Handling: This involves managing the sales data received per town. While the problem statement omits explicit input data validation, ensuring accurate and valid data input is critical in real-world applications.
- Sales Calculation: Summing up the total sales from the number of locks, stocks, and barrels sold, multiplied by their respective prices, to determine the total sales revenue for the month.
- Commission Calculation: Applying a tiered commission structure to the total sales revenue to compute the salesperson's earnings for the month. This requires careful calculation to ensure commissions are accurately applied at each tier.

Testing Considerations:

Testing this application would involve validating the correct execution of each step:

- Ensuring accurate calculations of total sales.
- Correctly implementing the commission tiers.
- Proper handling of the sentinel value to terminate monthly data input.

The Commission Problem exemplifies the challenges in processing sales data and commission calculations. Testing methodologies play a vital role in validating the system's functionality and accuracy, ensuring reliable operations in real-world scenarios.

What are the Complexities of Testing the Commission Problem?

Testing the Commission Problem involves various complexities due to the nature of the problem and the requirements involved. Some of the complexities of testing the Commission Problem are:

1. Data Validation: Ensuring that input data such as the number of locks, stocks, and barrels sold is validated correctly to prevent errors in calculations.

2. Boundary Conditions: Handling boundary conditions such as reaching the maximum sales limits for locks, stocks, and barrels, and ensuring that the commission calculation is accurate in such scenarios.

3. Commission Tiers: Testing the commission calculation logic for different tiers (10%, 15%, and 20%) based on the total sales revenue, including scenarios where sales fall within multiple tiers.

4. Sentinel Value Handling: Validating the correct handling of the sentinel value (-1 locks sold) to signal the end of data input for the month, ensuring it triggers the commission calculation accurately.

5. Integration Testing: Testing the integration of different components of the problem, such as input data handling, sales calculation, and commission calculation, to ensure they work seamlessly together.

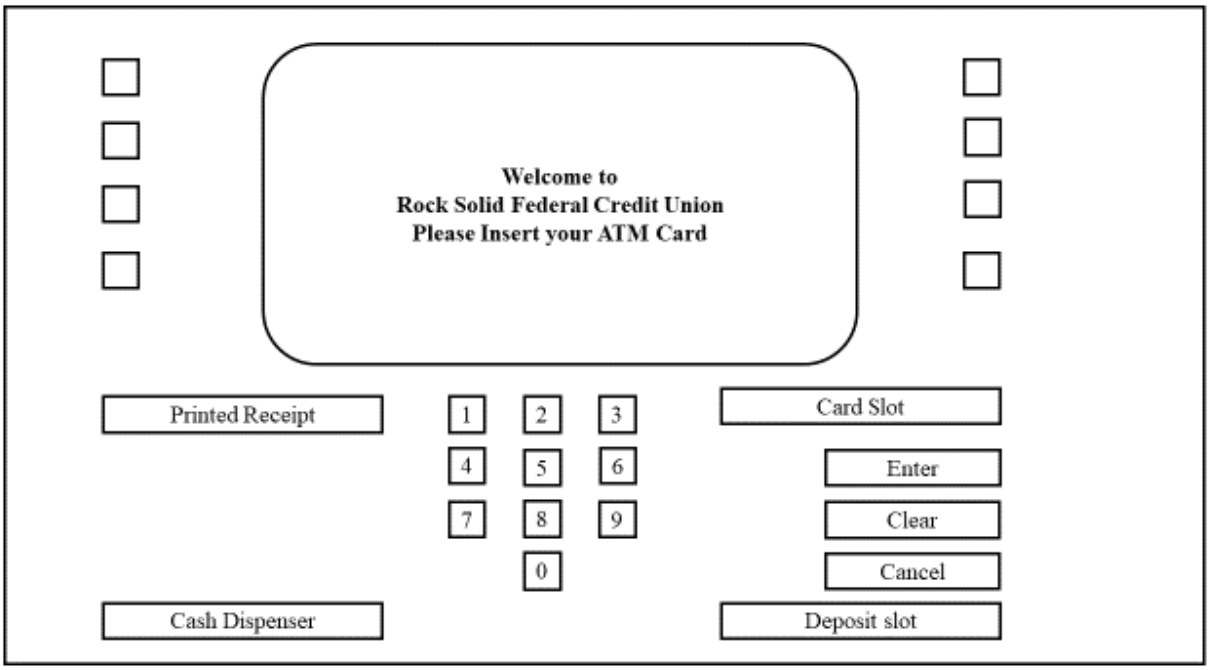
6. Error Handling: Testing error-handling mechanisms for scenarios like invalid input data, exceeding maximum sales limits, or unexpected data formats in the input.

The SATM (Simple Automatic Teller Machine) Problem

The Simple ATM System (SATM) serves as a practical example to illustrate the complexities involved in integration and system testing of a client-server architecture. With a set of functionalities captured in a series of interactive screens, the SATM system provides an ideal case to examine how different components within an ATM interface work together to handle user transactions seamlessly.

Problem Statement:

The Simple ATM system simulates real-world banking transactions via an interface shown in Figure 1.6. It demonstrates how various components interact to complete user-driven tasks.



A terminal is equipped with various user interaction components like a card slot, keypad, and screens for displaying messages and options. Customers, interact with the SATM by using a plastic card encoded with a personal account number (PAN). The system progresses through multiple screens, each corresponding to different stages of transaction insertion, PIN entry, transaction selection, to the final transaction execution.

The SATM interacts with bank customers using 15 interactive screens as depicted in the below figure. Each screen represents a different stage of the transaction process, capturing all necessary user interactions and system responses.

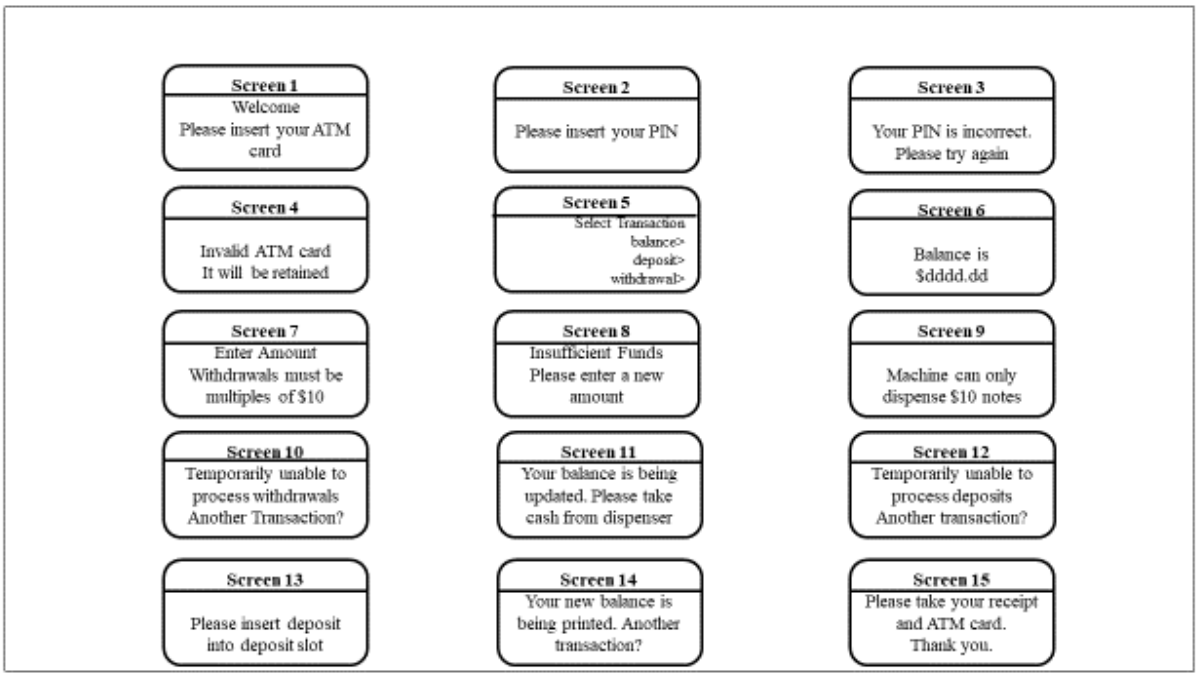


Fig. SATM Screens

Initial Interaction: When a customer approaches the SATM,) the interface shown on screen 1 prompts them to insert their ATM card into the card slot (This triggers a verification process where the system checks the personal account number (PAN) encoded on the card against an internal database.

Authentication If the PAN is verified, the system advances the customer to screen 2, asking for a PIN. If the PAN does not match, screen 4 appears, indicating the card is invalid and will be retained. After correct PIN entry, the customer moves to screen 5 incorrect entries after three attempts lead to screen 4 where the card is retained.

Transaction Selection On screen 5, the customer selects from available transaction, balance, deposit, or withdrawal) The system then navigates to different screen based on the selection:

Balance: Leads directly to screen 14 showing the account balance.

Deposit: (If the deposit slot is operational (as per the terminal control file), the system proceeds to screen 7 to accept the deposit amount. If there is an issue, it moves to screen 12. Following the deposit, the system processes the transaction and updates the balance on screen 14

Withdrawal: The system first checks the status of the withdrawal. If it's jammed screen 10 is displayed. If it's operational, screen 7 appears for entering the withdrawal amount. Post this, if the funds are insufficient, screen 8 is shown otherwise, the processes the withdrawal and displays the new balance on screen 14.

The SATM's workflow is designed to ensure secure and efficient transaction processing. Each step of the interaction from user authentication to the final transaction is handled through specific

screens that guide the user through the process. This system's design allows for clear separation of functionalities, which is crucial for both integration testing, where different system components are tested together, and system testing, where the entire system functionality is evaluated in a real-world scenario.

Integration Challenges:

- Ensuring seamless data flow between screens and the backend database for real-tim updates on user transactions and account balances.
- Handling hardware interactions, such as card reading and cash dispensing, which requin the physical components and the software to work in unison.

System Testing Focus:

- Verifying that all transaction types are processed correctly and under various scenarios including edge cases like maximum withdrawal limits or operational failures (e.g., jammed deposit slot).
- Testing the system's response to user inputs across different screens to ensure consistent and secure handling of transactions.

What are the Complexities of Testing the SATM Problem?

Testing the Simple ATM System (SATM) involves a range of complexities due to its interactive, multi-step nature and the critical need for security and reliability in financial transactions. Here are several key complexities involved in testing the SATM:

1. Integration of Hardware and Software Components: Testing must ensure that the hardware (card reader, keypad, cash dispenser, deposit slot) and software components interact flawlessly. Each component must respond correctly to user inputs and system commands under various scenarios.

2. User Interface and Experience: The system involves a series of screens, each designed for specific functions like entering a PIN, choosing a transaction, or handling errors. Tests need to verify that each screen correctly displays the expected information and that transitions between screens are smooth and logical.

3. Transaction Logic Accuracy: Each transaction type-withdrawals, deposits, and balance inquiries— has specific logical paths that need to be thoroughly tested for accuracy. For example, the system must correctly calculate and update balances, validate transaction conditions (e.g., sufficient funds, correct PIN), and handle transaction limits.

4. Error Handling and Exception Management: The system should gracefully handle errors such as incorrect PIN entries, unrecognized card information, hardware malfunctions (like a jammed cash dispenser), and other operational issues. Testing should include scenarios where these errors are triggered to ensure the system responds appropriately.

5. Security Testing: Given the sensitive nature of financial transactions, security is most important. Testing must cover data encryption, secure communication between the ATM and the bank's servers, protection against physical and cyber threats, and compliance with financial regulations.

6. Concurrency and Session Management: The ATM may handle multiple users sequentially or almost concurrently. Testing needs to ensure that session data from one customer does not leak into another session, and that the system can manage simultaneous transactions safely and correctly.

7. Performance and Reliability: The SATM should perform reliably under varying conditions, including high usage periods, network connectivity issues, and under different operational conditions. Stress and performance testing help verify that the system operates efficiently without crashes or slowdowns.

8. Usability and Accessibility: Testing must also cover the usability and accessibility of the ATM for all types of users, including those with disabilities. This involves checking the clarity of instructions, the responsiveness of the interface, the physical accessibility of the machine, and the overall user experience.