SOFTWARE TESTING

As per the New NEP Syllabus for BCA 6th Semester Course of

Bengaluru City University and Bangalore University

By

Srikanth S

Ravikiran R.K	**Murulidhara C**
Associate Professor	Associate Professor
Department of Computer Science & Application	Department of Computer Science & Application
SRN Adarsh College, Chamarajpet,	SRN Adarsh College, Chamarajpet,
Bengaluru	Bengaluru

Skyward Publishers
#157, 7th Cross, 3rd Main Road, Chamarajpet,
Bengaluru-18. Phone : 080-43706620 / 080-26603535
Mob: 9611185999
E-mail: skyward.publishers@gmail.com
Website: www.skywardpublishers.co.in

A Text Book of "Software Testing" - by Srikanth S, Ravikiran R.K & Murulidhara C as per the New NEP Syllabus for VI Semester BCA, Bengaluru City University & Bangalore University.

First Edition : 2024

ISBN : 978-93-95085-99-1

Price : ₹ 300/-

Published by:

Skyward Publishers

#157, 7th Cross, 3rd Main Road, Chamarajpet
Bangalore-18. Phone : 080-26603535 / 43706620,
Mob: 9611185999
E-mail: skyward.publishers@gmail.com
Website: www.skywardpublishers.co.in

DTP By

Mary & Nirmala, Skyward Team

PREFACE

We are delighted to present this comprehensive book on **"Software Testing"** meticulously crafted to align with the National Education Policy (NEP) syllabus for the 6th Semester BCA course of Bengaluru City University and Bangalore University. This book is designed to meet the specific needs of students and educators by providing thorough coverage of the syllabus, explained in a systematic and clear manner.

Key Highlights of This Book:

- **Strict Adherence to Syllabus :** This book strictly follows the NEP syllabus for the 6th Semester BCA course to ensure that all required topics are covered comprehensively.
- **Real-Time Examples :** To facilitate a deeper understanding of software testing concepts, we have included numerous real-time examples. These practical insights help bridge the gap between theoretical knowledge and its application in real-world scenarios.
- **Detailed Descriptions :** Every topic is described in detail to ensure that students gain a thorough understanding of software testing principles, techniques, and tools.
- **Review Questions :** At the end of each chapter, review questions are provided to help students assess their understanding and reinforce their learning.
- **Model Question Papers :** Four model question papers are included to give students ample practice and prepare them for their examinations effectively.
- **Systematic and Clear :** The content is organized systematically, presented in a clear and easy-to-understand manner, making it accessible to all students regardless of their prior knowledge of the subject.

This book is not just a learning resource but a comprehensive guide that aims to equip students with the necessary skills and knowledge to excel in the field of software testing. We believe that the systematic approach, detailed explanations, and practical examples will make this book an invaluable tool for students.

We welcome your comments and suggestions to help us improve future editions of this book. Please send your feedback to **[skyward.publishers@gmail.com]**

Thank you for choosing this book as your companion in learning software testing. We wish you the best in your academic and professional endeavors.

Happy Learning!

- Authors

SYLLABUS

UNIT – 1 :

Introduction: Basic definitions, A testing life cycle, Test Cases, Fundamental approaches to apply Test Cases, Levels of Testing, 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 Boundary Value Analysis, Robustness Testing, Worst-Case Testing, Special Value Testing, Test cases for the Triangle problem, Test cases for the NextDate function, Test cases for the Commission Problem, Random Testing and Guidelines for Boundary Value Testing.

UNIT – II :

Equivalence Class Testing: Equivalence Classes, Weak Normal Vs Strong Normal Equivalence Class Testing, Weak Robust Vs Strong Robust Equivalence Class Testing, Equivalence Class Test Cases for the Triangle Problem, Equivalence Class Test Cases for the Next Date Function and Equivalence Class Test Cases for the Commission Problem, Guidelines for Equivalence Class Testing.

Decision Table Based Testing: Decision tables, Test cases for the triangle problem, Test cases for the Next Date function, Test cases for the commission problem, Guidelines and observations.

Data flow Testing: Definition Use Testing, Example- The Commission Problem, Slice-Based Testing, Guidelines and Observations.

UNIT – III :

Levels of Testing: The SATM System, Structural and Behavioural Insights. **Integration Testing:** A Closer Look at the SATM System, Decomposition-Based Integration, Top-Down Vs Bottom-Up Integration, Sandwich Integration, Call Graph-Based Integration, Pair wise Integration, Neighborhood Integration, Path-Based Integration.

System Testing: Threads, Basic concepts for requirements specification, Finding threads, Structural strategies and functional strategies for thread testing,

Interaction Testing: A Taxonomy of Interactions, Static Interaction in a Single Processor, Static Interaction in Multiple Processors, Dynamic Interaction in a Single Processor, Dynamic Interaction in Multiple Processors, Client-Server Testing.

UNIT – IV :

Object Oriented Testing: Issues in Object Oriented Testing, Implication of Composition and Encapsulation, Implications of Inheritance, Implications of Polymorphism, GUITesting, Object-Oriented Integration Testing.

Exploratory Testing: The context-driven school, Exploring exploratory testing, Exploring a familiar example, Exploratory and context-driven testing observations.

Model-Based Testing: Testing based on models, Appropriate models, Use case-based testing, Commercial tool support for model-based testing.

Test-Driven Development: Test-then-code cycles, Automated test execution, Java and JUnit example, Remaining questions, Pros, cons, and open questions of TDD, Retrospective on MDD versus TDD,

Software Testing Excellence: Craftsmanship, Best practice of software testing, Top 10 best practices for software testing excellence.

CONTENTS

UNIT - I

CHAPTER 1

INTRODUCTION TO SOFTWARE TESTING

Contents

1.1 Introduction

Nowadays, software runs all aspects of modern life and accounts for a large and increasing share of the world economy. This trend started slowly with the advent of computing in the middle of the twentieth century and was further precipitated by the emergence of the World Wide Web at the end of the twentieth and the beginning of the twenty-first century. This phenomenon has spawned a great demand for software products and services and generated a market pressure that the software industry takes great pains to cater to.

The success of any software product or application is greatly dependent on its quality. Today, testing is seen as the best way to ensure the quality of any product. Quality testing can greatly reduce the cascading impact of rework of projects, which have the capability of increasing the budgets and delaying the schedule. The need for testing is increasing, as businesses face pressure to develop sophisticated applications in shorter time frames. Testing is a method of investigation conducted to assess the quality of the software product or service. It is also the process of checking the correctness of a product and assessing how well it works.

Generally we hear a software problems like a bank reporting incorrect account balances, software printing 120 out of 100 marks in marks card, a grocery store scanner charging too much for grocery items. Why does this happen? Can't computer programmers figure out ways to make software just plain work? Unfortunately, no. As software gets more complex, gains more features, and is more interconnected, it becomes more and more difficult to create a glitch-free program. Despite how good the programmers are and how much care is taken, there will always be software problems.

That's where software testing comes in. Many large software companies are so committed to quality they have one or more testers for each programmer. These jobs span the software spectrum from computer games to factory automation to business applications.

Software bugs can cause potential monetary and human loss. There are many examples in history that clearly depicts that without the testing phase in software development lot of damage was incurred. Below are some examples:

- **1985:** Canada's Therac-25 radiation therapy malfunctioned due to a software bug and resulted in lethal radiation doses to patients leaving 3 injured and 3 people dead.
- **1994:** China Airlines Airbus A300 crashed due to a software bug killing 264 people.
- **1996:** A software bug caused U.S. bank accounts of 823 customers to be credited with 920 million US dollars.
- **1999:** A software bug caused the failure of a $1.2 billion military satellite launch.
- **2015:** A software bug in fighter plan F-35 resulted in making it unable to detect targets correctly.
- **2015:** Bloomberg terminal in London crashed due to a software bug affecting 300,000 traders on the financial market and forcing the government to postpone the 3bn pound debt sale.
- Starbucks was forced to close more than 60% of its outlet in the U.S. and Canada due to a software failure in its POS system.

- Nissan cars were forced to recall 1 million cars from the market due to a software failure in the car's airbag sensory detectors.

1.1.1 What is Software Testing?

A software bug usually occurs when the software does not do what it is intended to do or does something that it is not intended to do. Flaws in specifications, design, code or other reasons can cause these bugs. Identifying and fixing bugs in the early stages of the software is very important as the cost of fixing bugs grows over time. So, the goal of a software tester is to find bugs and find them as early as possible and make sure they are fixed.

The process of testing identifies the defects in a product by following a method of comparison, where the behavior and the state of a particular product is compared against a set of standards which include specifications, contracts, and past versions of the product. Software testing is an incremental and iterative process to detect a mismatch, a defect or an error. As pointed by Myers, "Testing is a process of executing a program with the intent of finding errors".

? What is Software Testing?

Software Testing is a method to assess the functionality of the software program. The process checks whether the actual software matches the expected requirements and ensures the software is bug-free. The purpose of software testing is 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 program or an application.

Definitions: Software Testing

- **According to Glenford J. Myers**, software testing is defined as "the process of executing a program or system with the intent of finding errors."
- **According to IEEE (Institute of Electrical and Electronics Engineers)**, software testing is defined as "the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."
- **According to Roger S. Pressman**, software testing is defined as "the process of executing a software application with the intent to uncover defects, validate that the system meets its requirements, and ensure that it operates correctly in its intended environment."
- **According to Paul C. Jorgensen**, software testing is defined as "The process of reducing uncertainty about the correctness of a software system by executing the system with the purpose of finding its defects, if any."

1.1.2 Why Software Testing is Important ?

Software Testing is important because if there are any bugs, errors, in completed requirements in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

? Why Software Testing is Important? or What are the Benefits of Software Testing?

- **Defects can be identified early:** Software testing is important because if there are any bugs they can 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 behaviour. 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.

1.1.3 Goals or Objectives of Software Testing

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.

1.1.4 Importance of Software Testing

Importance of Software Testing

1. Software testing ensures quality by identifying defects and errors.
2. Testing improves customer satisfaction by meeting user needs and expectations.
3. Testing reduces costs by identifying and fixing issues early in the development process.
4. Testing improves reliability by ensuring consistent performance under different conditions.
5. Testing minimizes risks by identifying potential issues before they become costly to fix.
6. Testing facilitates compliance with industry standards and legal requirements.
7. Testing enhances performance by identifying and addressing performance issues early on.
8. Well-tested software is easier to maintain and update.

1.1.5 Classification of Software Testing

Software testing can be classified into various types based on different criteria and objectives. Some common classifications of software testing are listed below.

1. Based on Testing Objectives

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.

Non-Functional Testing:

- **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.

2. Based on Testing Levels

- **Unit Testing:** Unit testing involves testing individual units or components of the software in isolation. It helps identify defects early in the development cycle and ensures that each unit functions correctly.

- **Integration Testing:** Integration testing verifies the interactions between integrated components to detect interface issues and ensure that the integrated modules work together as expected.
- **System Testing:** System testing tests the entire software system as a whole to validate that it meets the specified requirements and functions correctly in its intended environment.
- **Acceptance Testing:** Acceptance testing validates whether the software meets user requirements and is ready for deployment. It often involves User Acceptance Testing (UAT) by end-users to ensure that the software meets business needs.

3. Based on Testing Techniques

- **Manual Testing:** Manual testing involves human testers executing test cases manually without the use of automation tools. It allows for exploratory testing, ad-hoc testing, and detailed analysis of software behavior.
- **Automated Testing:** Automated testing uses automation tools to execute test scripts, compare actual outcomes with expected results, and improve testing efficiency and repeatability. It is beneficial for regression testing, performance testing, and test case execution.

4. Based on Testing Methods

- **Black Box Testing:** Black box testing focuses on testing the software's functionality without knowledge of its internal code structure. Testers validate inputs and outputs based on the software's specifications and expected behavior.
- **White Box Testing:** White box testing tests the internal logic, code structure, and paths within the software. Testers have knowledge of the internal code to design test cases that cover different code paths and conditions.

5. Based on Testing Strategies

- **Top-Down Testing:** Top-down testing starts testing from the highest-level modules and progresses downwards. It emphasizes early integration testing and ensures that higher-level modules function correctly before integrating with lower-level modules.
- **Bottom-Up Testing:** Bottom-up testing begins testing from the lowest-level modules and moves upwards. It focuses on early unit testing, identifying defects in individual units, and gradually integrating them to form higher-level modules.

Examples **Software Testing Scenarios**

The below examples illustrate various scenarios where software testing is essential to ensure the functionality, usability, and performance of software applications:

1. **Testing a game to see if you can beat all the levels without the game crashing:** This example highlights the importance of testing the stability and performance of a game application. Testers would play through different levels, scenarios, and interactions to identify any bugs, crashes, or performance issues that may impact the player experience.
2. **Trying out a new drawing app to see if all the colors and tools work properly:** This scenario emphasizes the need for functional testing to validate the features and capabilities of a drawing application. Testers would explore different drawing tools, colors, brushes, and functionalities to ensure they work as intended and deliver the expected user experience.
3. **Making sure a college website displays information correctly on phones and laptops:** This example showcases the significance of compatibility testing to verify that a website functions seamlessly across various devices and screen sizes. Testers would access the school website on different devices such as phones and laptops to check for responsiveness, layout consistency, and content display.

1.1.6 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 2. Inspections 3. Walk through

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 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.

When Do Verification and Validation (V&V) Start and End?

Although some primitive software development processes concentrate testing and analysis at the end of the development cycle, and the job title "tester" in some organizations still refers to a person who merely executes test cases on a complete product, today it is widely understood that execution of tests is a small part of the verification and validation process required to assess and maintain the quality of a software product.

- Test is not a (late) phase of software development.
  - Execution of tests is a small part of the verification and validation process.
- V&V start as soon as we decide to build a software product, or even before
- V&V last far beyond the product delivery as long as the software is in use, to cope with evolution and adaptations to new conditions.

Difference between Verification and Validation

Verification	Validation
Evaluates the intermediary products to check whether it meets the specific requirements of the particular phase.	Evaluates 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 walk through.	Examples include all types of testing like smoke testing, regression, functional testing, systems and User Acceptance Testing.

1.2 Basic Definitions

Testing software is a critical component of software development that ensures the quality and functionality of the final product. Essentially, testing involves evaluating software by learning how it behaves under various conditions to discover any errors and ensure the software performs as intended. It is important to understand the key testing terms in software testing process. These terms are derived from standardized definitions by the International Software Testing Qualification Board (ISTQB) and the Institute of Electronics and Electrical Engineers (IEEE):

Term	Description	Example
Error	Errors are mistakes made by people during the software development process. When these mistakes occur while coding, they are commonly referred to as bugs. Errors can escalate from requirements to design and further to coding stages.	Suppose during the development of the 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).	The fault in this scenario is the representation of the error in the code. The 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.	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.	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.	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 particular program behavior. Test cases help in systematically validating the functionality of the software.	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).

In software development and testing, errors in the code or system design can lead to faults when the flawed code is executed. These faults, if not detected and rectified, can result in failures where the system deviates from its expected behavior or produces incorrect outcomes. Failures, when experienced by users or stakeholders, can escalate into incidents that impact the system's functionality, performance, or security. Understanding this progression from errors to faults, faults to failures, and failures to incidents is essential for testers to effectively identify, address, and prevent issues throughout the software development lifecycle, ensuring the delivery of reliable and high-quality software products.

1.3 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.

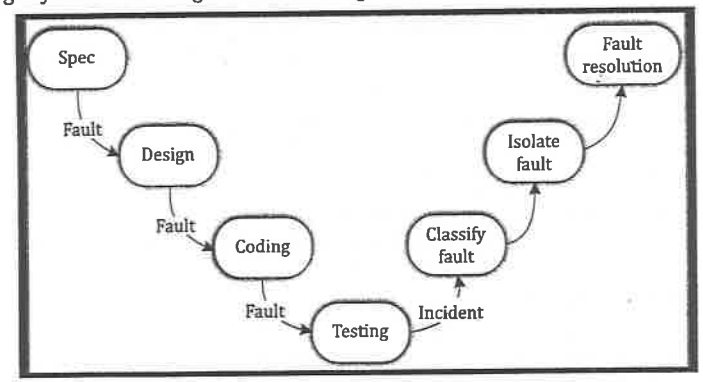


Fig 1.1 : A Testing Life Cycle

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.

1.4 Test Cases

A test case is a vital component of the testing process. It contain all the necessary elements to perform a test, validate functionality, and ensure compliance with the specifications. A Test Case is a set of actions performed to verify a particular feature of the Software Application. It contains Pre-conditions, Test Steps, Expected Results, Actual Results, Test Data etc. With the help of Test Cases, a Test Engineer can compare the Expected Results and Actual Results to determine whether a Software Application is working as per the Client's requirements.

Definitions : Test Case

- **According to IEEE Standard 610,** Test cases are defined as "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement."
- **According to Ron Patton,** "Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software."
- **Boris Beizer defines a** test as "A sequence of one or more sub tests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next."

The Value of Test Cases

Test cases are as critical to software development as the source code itself. They need to be carefully crafted, reviewed, and maintained. Properly developed test cases can:

- Serve as a documentation of what tests have been performed.
- Provide a basis for performing regression testing to ensure that changes do not adversely affect existing functionality.
- Help new team members understand testing procedures and the software's intended behavior.

1.4.1 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	Test Environment	Prerequisites	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

Designing test cases can be time-consuming in a testing schedule, but they are worth for spending time, because they can prevent unnecessary retesting or debugging or at least lower the rate of such operations. Organizations can take the test case approach in their own context and according to their own perspectives. Some follow a general approach while others may opt for a more detailed and complex approach.

? 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.

1.4.2 Insights from a Venn Diagram

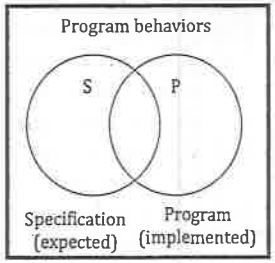
The use of Venn diagrams in software testing indeed provides a visual representation of the relationships between specified, implemented, and tested behaviors in software development. By illustrating the intersections and differences between these aspects, Venn diagrams offer a clear and concise way 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 cases 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**

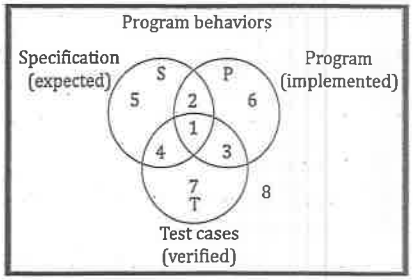
Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. The below figure 1.2 shows the relationship between the specified and implemented programmed behaviors.



- **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 ∩ 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 1.3 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 (S ∩ P ∩ T):** 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 (S ∩ P - T):** Behaviors that are specified and implemented but not tested. This indicates gaps in the testing coverage.
  - **Region 3 (P - S ∩ T):** 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 - P ∩ 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.

1.5 The Traditional Model of the Software Testing Process

The traditional model of the software testing process for plan-driven development like waterfall includes a structured sequence of steps to ensure that software behaves as expected and identifying and rectifying any defects. The process is shown in below diagram.

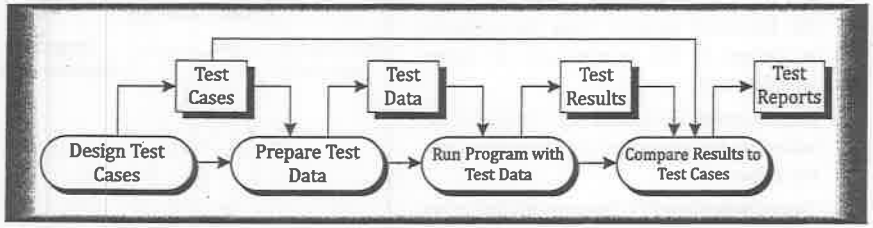


Fig 1.2 : Traditional Model of the Software Testing Process

The model comprises the following key stages:

1. **Design Test Cases :** In this initial phase, specific scenarios or "test cases" are created based on the software's requirements and design documentation. These test cases are designed to 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 inputs 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 compared to the expected outcomes outlined in the test cases. Any deviations indicate potential issues that require attention. This comparison is instrumental in verifying that the software meets 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 or problems identified. They serve as critical resources for developers to understand the aspects of the software that require correction.

1.6 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:

1. **Unit Testing:** This is the lowest level of testing where individual units or components of the software are tested in isolation. Unit testing verifies that each unit works correctly as per the design specifications. It is typically performed by developers and automated testing tools.

2. **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 work correctly as a whole.
3. **System Testing:** System testing evaluates the behavior of the entire software system as a whole. It tests the system against the functional and non-functional requirements to verify that it meets the specified criteria. System testing is usually performed by independent testers.

1.6.1 Unit Testing

Unit testing is a foundational aspect of the software development process where individual components or modules such as methods or object classes are tested to ensure they work correctly. This involves calling functions or methods with various parameters and checking for the correct output to ensure all operations and attributes associated with the object are functioning as expected, and verifying the object's behavior across different states.

The goal of unit testing is to find errors within that part of the program by testing important control paths using a description of how the component works. Unit testing is limited in scope, which means it only tests a small part of the program at a time. The focus of unit testing is on how the component processes information and stores data.

? 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.

1.6.2 Integration Testing

Integration testing or component testing is a software testing technique that focuses on verifying the behavior of individual software components or modules. However, in complex software systems, components are often composite components that are made up of several interacting objects. In such cases, testing the composite component is essential to ensure that the interactions and interfaces between the objects within the component behave as expected.

When testing composite components, the focus is on the interface of the combined components rather than on the individual objects. This is because interface errors in the composite component may result from interactions between the objects, which may not be detectable by testing the individual objects.

Consider a scenario where a software system has a component responsible for processing customer orders. This component interacts with a pricing component to calculate the total cost of the order. Testing the composite component (order processing) involves verifying that the parameters passed to the pricing component are correct and that the interface behaves as expected. An error in the way the order processing component passes parameters to the pricing component could lead to incorrect pricing calculations, which may not be apparent when testing the pricing component in isolation.

Therefore, integration testing involves verifying that the interactions between the objects within the component are correct and that the component meets its functional and non-functional requirements. It assumes that unit tests for individual objects within the component have been completed, and the focus is on testing the behavior of the composite component as a whole.

? 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.

Furthermore, testing the error handling component should emphasize its interface with the message delivery component to validate the system's ability to handle errors related to failed message deliveries and provide appropriate user feedback.

1.6.3 System Testing

System testing is a type of software testing that involves testing the entire system as a whole, rather than testing individual components or modules in isolation. It is a critical phase in the software development life cycle, where the system is tested to ensure that it meets the specified requirements and performs as expected in the real-world environment.

The main objective of system testing is to verify that the system is functioning correctly and meets the business and technical requirements. It involves testing the system's functionality, performance, reliability, security, and usability. System testing is usually performed after the completion of component and integration testing, where individual components are integrated and tested as a group.

System testing is typically performed in a real-world environment that simulates the actual usage of the system. It involves testing the system from end-to-end, including all the interfaces and interactions between different components. Test scenarios are created to cover all the possible use cases and scenarios that the system may encounter in the real world. Test data is also created to simulate the real-world data that the system will encounter.

? 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.

1.6.4 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.

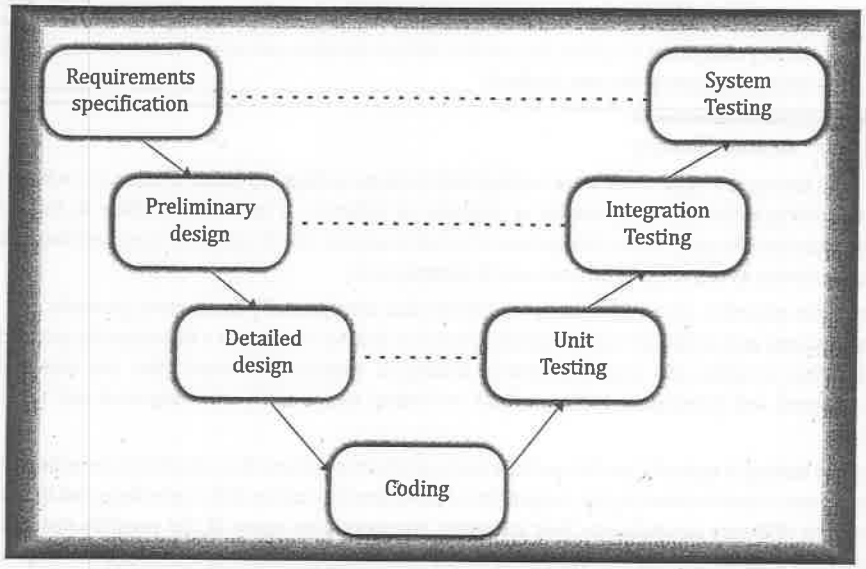


Fig 1.3 : Levels of Abstraction and Testing in V-Model

Specification-based testing occurs at three distinct levels which correspond to the different stages of 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 as designed.

This systematic approach to testing at each level of the software development process not only helps in detecting errors at the earliest possible stage but also aids in maintaining a clear focus on meeting the predefined design and specification objectives. The correspondence between the levels of design and testing in the V-Model provides a robust framework for quality assurance throughout the software development lifecycle, making it possible to refine and validate the software continuously as it progresses from conception to completion. This methodical alignment of design and testing is instrumental in improving the quality, reliability, and performance of the final software product.

1.7 Structural and Behavioural 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.

1.8 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)**

1.8.1 Specification-Based Testing (Functional Testing or Black Box Testing)

Specification-Based Testing (Functional Testing or Black Box Testing) is a common activity that we perform in our daily lives, often without even realizing it. We interact with the system being tested as if it were a mystery box. We may not know how the system works internally, but we know how

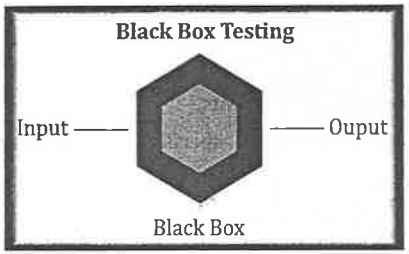
it should behave. For example, when we test our car or bike, we drive it to ensure that it behaves as expected. This is an example of black-box testing.

Specification-Based Testing (Functional Testing or Black Box Testing) 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. In other words, we treat the software as a "black box" and tests its functionality based on inputs and expected outputs without considering how the software processes those inputs. Specification-based testing, relies solely on the software specification (requirement specification, design specification etc.,) to identify test cases.

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 requirements 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 (functional 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 see. The main purpose of Black Box testing is to check whether the software is working as expected and meeting the customer requirements or not. It was designed as a method of analyzing client's 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 programme behaviour, from an informal partial description to a formal specification, from module to system testing. Functional tests 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 a 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.
- Testing the functions of an ATM is a good example of specification based or functional testing The tester acts as a customer who is using the ATM and checks the functions of the machine. He/She does not know the internal working of the logic. The test cases are developed to check the functions through the GUI of the ATM such as change in display of the GUI when card is detected, masking the password or navigating from main menu to a specific function.

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.

Specification-Based Test Case Identification Methods

The below Venn Diagram (Figure 1.4) for specification-based methods provides a visual representation of how different specification-based testing methods cover the set of specified behaviors. In specification-based or functional testing, the focus is primarily on ensuring that the software meets its outlined specifications without regard to how these functionalities are implemented.

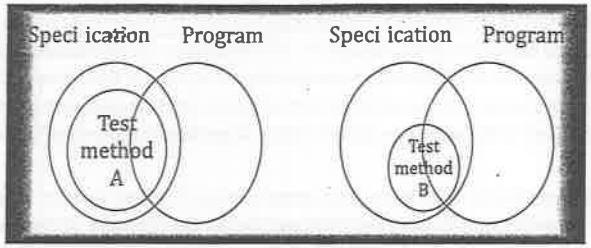


Fig 1.4 : Comparing specification based test case identification methods

The two circles within the diagram represents the different specification-based test method:

- Circle A (Test Method A)
- Circle B (Test Method B)

Both circles are within a larger set labeled "Specification," which include all the behaviors that the software is expected to exhibit according to its requirements documentation.

Example of Specified Behaviors:

Let's say a software application is supposed to handle user authentication, data processing, and report generation. The specifications would detail the requirements for each of these functionalities, like:

- Users must be able to log in with a username and password.
- The system must process data inputted by the user and provide output within 2 seconds.
- Users should be able to generate reports based on selected data.

How Test Methods Cover Specified Behaviors

- Test Method A might be designed to cover basic functionalities such as:
  - **User Authentication:** Validating correct user credentials and rejecting invalid attempts.
  - **Basic Data Processing:** Ensuring that data input is accepted and correctly processed to generate an expected output.
- Test Method B might focus on more detailed aspects or additional features such as:
  - **Advanced Data Processing:** Checking the system's handling of edge cases in data processing.
  - **Report Generation**: Verifying that all types of reports can be generated accurately based on user-selected parameters.
- The overlap between A and B would represent the common functionalities both methods test, perhaps basic data processing, which is a core requirement.
- Parts of A not overlapping with B could include specific tests unique to basic functionalities not covered by B, like specific user authentication tests.
- Parts of B not overlapping with A would highlight more complex or additional functionalities that B tests but A does not, such as comprehensive checks on report generation.

The Venn diagram serves to illustrate visually how different testing methods based on the same specifications can vary in their focus and coverage. By analyzing these diagrams, stakeholders can identify potential gaps in testing (where certain specified behaviors are not covered by either method), redundancies (where the same functionalities are being tested by both methods unnecessarily), and ensure a comprehensive testing strategy that covers all critical specified behaviors effectively.

1.8.2 Code-Based Testing (Structural Testing or White Box Testing)

Code-Based Testing is considered more technical than specification based testing (functional testing). It attempts to design test cases from the source code and not from the specifications. The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details. It also gives insight in to the source code which may be used as an essential knowledge for the design of test cases. **Code-Based Testing** is also known as **White Box Testing** or **Structural Testing.**

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).

To carry out this type of testing, we need to thoroughly understand the code. This is why this testing is usually done by the developers who wrote the code as they understand it better.

It is more concerned with how system does it rather than the functionality of the system. It provides more coverage to the testing. For example, to test certain error message in an application, we need to test the trigger condition for it, but there must be many trigger for it. It is possible to miss out one while testing the requirements drafted in requirement specification. But using this testing, the trigger is most likely to be covered since structural testing aims to cover all the nodes and paths in the structure of code.

The intention behind the testing process is finding out how the system works not the functionality of it. To be more specific, if an error message is popping up in an application there will be a reason behind it. Structural testing can be used to find that issue and fix it

Code Based Testing is complementary to Specification Based Testing. Using this technique the test cases drafted according to system requirements can be first analyzed and then more test cases can be added to increase the coverage. It can be used on different levels such as unit testing, component testing, integration testing, functional testing etc. Its helps in performing a thorough testing on software. The structural testing is mostly automated.

Characteristics of Code Based Testing

- It is focused on the internal workings of the software application, including its code, architecture, and design.
- 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.
- 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..
- 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.
- 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.
- 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 of missing out on any error become very low.
- It is particularly useful for identifying complex defects that may be difficult to detect using other types of testing approaches.
- 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.
- 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 automate some aspects of structural testing and make it more efficient.
- 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.

Code-Based Test Case Identification Methods

The below Venn diagram show a comparison of two code-based test case identification methods. Each diagram consists of two overlapping circles, representing "Specification" and "Program."

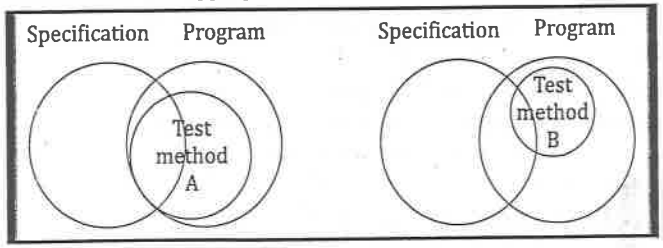


Fig 1.5 : Comparing Code-Based Test Case Identification Methods

- **Test Method A (Left Diagram):** This method seems to focus on identifying test cases that are relevant to both the program and the specification. The overlap, where "Test Method A" is placed, indicates that the chosen test cases are those that meet both the criteria set by the program and the specification.
- **Test Method B (Right Diagram):** In this method, the identified test cases overlap slightly differently with the specification and the program. "Test Method B" is located mostly within the "Program" circle but still overlaps with the "Specification" circle. This suggests that this method prioritizes test cases that are primarily relevant to the program but also considers their applicability to the specification.

These methods approach the selection of test cases from different perspectives. Method A might be seen as more balanced or conservative, ensuring that tests cover aspects important to both specification and program equally. Method B seems more program-oriented, but ensuring compliance or adherence to predefined requirements.

? How to Select a Effective Testing Method?

Choosing the right testing methods depends on understanding the specific needs of the project and the types of errors most likely to occur. Testers should aim to employ a mix of methods that cover both functional and structural aspects of the software, providing a balanced view of its reliability and correctness.

1.9 Examples

Software testing is vital for ensuring software systems work reliably. In the upcoming chapters, various examples will be used to explain different testing methods and difficulties at both the unit and system levels. These examples include basic problems like triangles, more complex functions like NextDate, and practical applications like commission problems and automated systems such as ATMs. By looking at these examples, testers can learn how to test different types of software and make sure they work well.

Examples for Unit Testing

1. **Triangle Problem**
   - A classic example in the testing community.
   - This problem involves categorizing a triangle based on the lengths of its sides.
   - It illustrates fundamental testing concepts at the unit level.
2. **NextDate Function**
   - This function is a complex task that calculates the subsequent date from a given date.
   - Represents a logically complex function.
   - Highlights challenges in managing input constraints and leap year calculations.
3. **Commission Problem**
   - It is typical scenario in Management Information Systems (MIS).
   - Involves a mix of computation and decision-making.
   - This problem is utilized to discuss testing strategies like data flow and slice-based testing.

Examples for Higher-Level Testing (System Testing)

1. **Simple ATM System (SATM)**
   - A simplified version of an automated teller machine.
   - Demonstrates testing at a higher level of abstraction.
   - Focuses on higher-level testing concerns and their practical implications.
   - Focuses on system-level testing considerations.

1.9.1 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: $1 \le a \le 200$	Condition c4: $a < b + c$
Condition c2: $1 \le b \le 200$	Condition c5: $b < a + c$
Condition c3: $1 \le c \le 200$	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.

➢ **Traditional Programming Implementation :**

The traditional way of solving the Triangle Problem, often resembling older programming styles like FORTRAN, involves using a series of steps to figure out the type of triangle based on its sides. This includes keeping track of side equality using a some variable called **match.** It checks if the sides follow the triangle inequality rule to confirm if it's not a triangle, and then using the match value to decide if the triangle is Equilateral, Isosceles, or Scalene.

How the match Counter Works ?

The match counter is initialized to zero at the start of the program. As the program evaluates the equality of the triangle's sides, it updates the match counter in the following ways:

- If side a equals side b, the match counter is incremented by 1.
- If side a equals side c, the match counter is incremented by 2.
- If side b equals side c, the match counter is incremented by 3.

Using the match Counter to Determine Triangle Type

- **Equilateral Triangle:** This is correctly determined if match equals 3, indicating that all three conditions ($a = b$, $a = c$, and $b = c$) are true.
- **Isosceles Triangle:** The triangle is classified as isosceles if match equals 1 or 2, indicating that exactly two sides are equal.
- **Scalene Triangle:** If the match value remains 0 after all checks, it indicates that no sides are equal ($a \neq b \neq c$).
- **Not a Triangle:** The validation against the triangle inequality theorem ($a + b > c$, $a + c > b$, $b + c > a$) still applies separately to determine if a valid triangle can be formed. If any of these conditions fail, it's not a triangle, irrespective of the match value.

Challenges with the Traditional Programming Implementation

The approach of using batch couter variable is efficient for quickly determining the type of triangle, but it mixes different logical checks (equality and triangle inequality) in ways that can be confusing. This can complicate understanding the flow of the program and debugging. Isolating issues becomes more challenging because changes to how match is calculated or interpreted might affect multiple parts of the triangle classification logic.

Thus, while the match approach is computationally efficient and clever in using unique sums to identify side equalities, it exemplifies a style of programming where clarity and maintainability could be compromised, making it harder for new programmers or testers to follow or modify the code without introducing errors.

➢ **Structured Implementations:**

The "structured programming" approach (like C, C++) breaks down the program into clearer, more manageable parts, making it easier to understand, maintain, and test:

- **Input Validation:** First, the program checks if the input values (the sides of the triangle) are within a valid range, say 1 to 200. This ensures that the sides are not only real, meaningful numbers but also within expected limits.
- **Triangle Validation:** Next, the program checks if the three sides meet the triangle inequality conditions (the sum of the lengths of any two sides must be greater than the length of the third side). This step confirms whether the given sides can actually form a triangle.
- **Type Determination:** Finally, the program determines the type of triangle based on how many sides are equal—if all three sides are the same, it's Equilateral; if two are the same, it's Isosceles; if none are the same, it's Scalene. If the sides don't form a triangle, it'll classify it as "Not a Triangle."

These structured steps make the program more robust and maintainable because each part does a specific task and does it well. It also makes testing easier. Testers can check each part independently to ensure it works correctly, without worrying about the other parts. This means bugs can be found and fixed more efficiently, leading to a more reliable program.

? 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.

1.9.2 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, and 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 straightforward tasks can become complex due to the various rules and exceptions 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.
- **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.

This complexity underscores the importance of thorough software testing to cover all possible scenarios by aligning with Zipf's law where a small portion of the code (e.g., leap year calculation) can significantly impact the function's complexity and operational focus.

? What are the Complexities of Testing the NextData 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 error messages for invalid inputs or exceptional cases, requires thorough testing to ensure robust error management.

1.9.3 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.

This problem, although somewhat simplified for clarity, reflects the complexities found in real-world MIS applications where multiple variables and conditional logic must be carefully managed to ensure accurate computations.

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.

1.9.4 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.

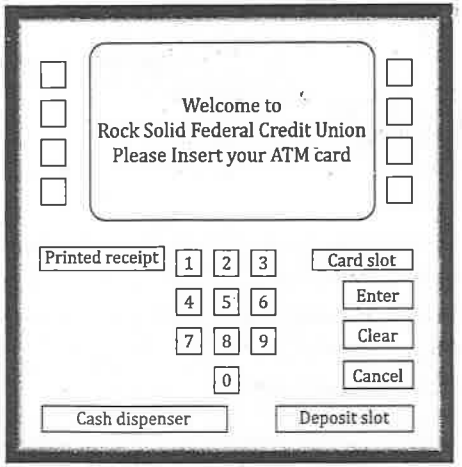


Fig 1.6 : SATM Terminal

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 processing—from card insertion, PIN entry, transaction selection, to the final transaction execution.

The SATM interacts with bank customers using a sequence of 15 interactive screens as depicted in Figure 1.7. Each screen represents a different stage of the transaction process, capturing all necessary user interactions and system responses.

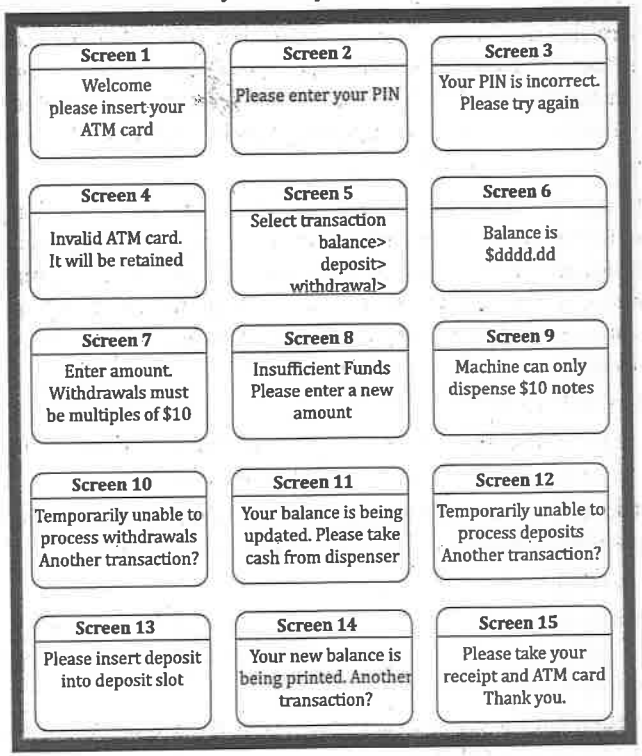


Fig 1.7 : SATM Screens

- **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 transactions: balance inquiry, deposit, or withdrawal. The system then navigates to different screens based on the selection:
- **Balance Inquiry:** 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 chute. 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 system processes the withdrawal and displays the new balance on screen 14.

➢ **Discussion**

The SATM's workflow is designed to ensure secure and efficient transaction processing. Each step of the interactio from user authentication to the final transactio 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's functionality is evaluated in a real-world scenario.

Integration Challenges:

- Ensuring seamless data flow between screens and the backend database for real-time updates on user transactions and account balances.
- Handling hardware interactions, such as card reading and cash dispensing, which require 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., a 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.

1.10 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Software Testing?
2. Define Software Testing.
3. What is Verification and Validation in Software Testing?
4. What is an Error? Give an example.
5. What is a Fault? Give an example.
6. What is a Failure? Give an example.
7. What is an Incident? Give an example.
8. Define Test and Testcase.
9. What is Test Suite?

10. Write the Process of Executing a Test Case.
11. Write the Levels of Testing.
12. What is Unit Testing? Give an example.
13. What is Integration Testing ? Give an example.
14. What is System Testing ? Give an example.
15. What is Specification-Based Testing? Give an example.
16. What is Code Based Testing? Give an example.

Section - B

Each Question Carries Five Marks

1. What are the Benefits of Software Testing?
2. Write the Goals or Objectives of Software Testing.
3. Explain the Importance of Software Testing.
4. What is Verification and Validation in Software Testing? Explain the techniques of Verification and Validation.
5. Mention the Differences between Verification and Validation.
6. Explain the Components of a Test Case.
7. Explain the use of Venn diagrams in software testing.
8. Explain the Traditional Model of the Software Testing Process.
9. Explain the Levels of Testing in V-Model.
10. Write a note on Structural and Behavioural Insights.
11. What is Specification-Based Testing? Write the Characteristics of Specification Based Testing.
12. What is Code-Based Testing? Write the Characteristics of Code Based Testing.
13. What is Triangle Problem ? What are the Complexities of Testing a Triangle Problem?
14. What is NextDate function ? What are the Complexities of Testing the NextDate function?
15. What is Commission Problem ? What are the Complexities of Testing the Commission Problem?
16. What is SATM Problem ? What are the Complexities of Testing the SATM Problem ?

Section - C

Each Question Carries Eight Marks

1. Explain the Classification of Software Testing.
2. Explain the Testing Life Cycle.
3. Explain Fundamental Approaches to Apply Test Cases with examples.

❖❖❖❖❖

UNIT - I

CHAPTER 2

BOUNDARY VALUE TESTING

Contents

2.1 Boundary Value Testing

Boundary value testing is an important method in software testing that involves testing the extreme values at the edges of input ranges. The idea is that errors are more likely to occur at these boundary points, so testing them can help find potential issues efficiently. By focusing on these edge cases, testers can uncover defects that may not be visible with typical test inputs within the normal range. This approach is valuable because it aligns with how a program functions, mapping inputs to outputs. it a fundamental component of specification-based testing strategies. We will explore different strategies to adapt this technique to various testing situations to identify errors and enhance software quality effectively.

Boundary Value Testing (BVT) is a specification based testing method that involves creating test cases based on the boundary values of input domains. Boundary values are the values at the edges of an input domain, just inside and just outside the boundaries, where the behavior of a system might change. This technique is based on the observation that errors tend to occur at the boundaries of input values rather than in the center.

Key Concepts of Boundary Value Testing:

- **Boundary Values:** These are the values at both ends of input ranges. For example, if an input field accepts values from 1 to 100, the boundary values would be 0, 1, 2, 99, 100, and 101.
- **Test Cases:** Boundary Value Testing focuses on creating test cases for these boundary values rather than testing with any value within the range. This approach helps to efficiently detect errors that are related to incorrect handling of data at the edges.
- **Function Mapping:** Just as a mathematical function maps inputs (domain) to outputs (range), a program takes specific inputs and generates outputs based on those inputs. Understanding this functional nature helps in designing effective test cases by considering the inputs and expected outputs.
- **Cross Products:** When a program's inputs or outputs are combinations of different variables, these can be treated as cross products, which are sets formed by combining each possible value of one variable with each possible value of another.

Example **Boundary Value Testing**

Suppose a function is designed to accept an integer value from 1 to 100 inclusive. Boundary Value Testing would generate test cases for values at and around the boundaries:

- Just below the minimum boundary (e.g., 0)
- At the minimum boundary (e.g., 1)
- Just above the minimum boundary (e.g., 2)
- Just below the maximum boundary (e.g., 99)
- At the maximum boundary (e.g., 100)
- Just above the maximum boundary (e.g., 101)

? What is Input Domain Function ?

An input domain function refers to the range of valid input values that can be accepted by a function or program. In the context of software testing, the input domain function is defined by the boundaries within which input variables must fall to ensure the correct functioning of the program.

Example : If we consider a function F that takes two variables x1 and x2, the input variables x1 and x2 are constrained by certain boundaries:

$a \leq x1 \leq b$

$c \leq x2 \leq d$

These boundaries [a, b] and [c, d] define the valid ranges for x1 and x2

? What is Boundary Value Testing ?

Boundary Value Testing (BVT) also known as **Input domain testing** is a specification-based testing technique that focuses on the inputs a program can accept. This type of testing is based on the idea that errors are most frequent at the edges of an input range, hence testing these boundary values can be more effective in finding bugs.

? Why Use Boundary Value Testing ? What is an Importance of Boundary Value Testing?

Boundary value testing is a software testing technique that involves creating test cases based on the boundary values of input domains. This method is particularly useful and frequently employed because it effectively identifies errors that occur at the edges of input ranges, where bugs are most likely to appear.

1. **High Error Detection Rate at Boundaries:** Many errors in software occur at the boundaries of input ranges due to off-by-one errors and other boundary-related issues. Boundary value testing specifically targets these potentially problematic areas, which increases the likelihood of catching bugs that might not be detected by other testing methods that use values well within the range.
2. **Efficiency:** Boundary value testing is a cost-effective method in terms of the number of test cases generated versus the potential defects found. By focusing on the edge cases, it reduces the number of test cases needed compared to exhaustive testing, which would require much more time and resources.
3. **Common Requirement Specifications:** Requirements often define operations or behaviors at the limits of input ranges (e.g., "the age should be between 18 and 60"). Testing these boundary conditions directly checks the system's adherence to its specified requirements.
4. **Usability and Reliability:** By ensuring that the software behaves correctly at boundary values, developers can improve the usability and reliability of their software. This is because handling boundary conditions gracefully often reflects the software's ability to handle unexpected or extreme inputs, which are critical in real-world operations.
5. **Early Defect Identification:** Identifying defects at the boundaries early in the testing process can lead to more efficient debugging and resolution, reducing the likelihood of critical issues in later stages of development.
6. **Integrates with Other Test Methods:** This method can be effectively combined with other testing strategies such as equivalence partitioning (where inputs are divided into logically similar groups), further refining the efficiency and effectiveness of the testing process.

2.2 Types of Boundary Value Testing

Boundary value testing is a critical technique in software testing where special focus is placed on the values at the edge of input domains. The four types of boundary value testing are:

1. **Normal Boundary Value Testing :** Normal Boundary Value Testing focuses on testing values at the boundaries within the valid range.
2. **Robust Boundary Value Testing :** Robust Boundary Value Testing extends Normal Boundary Value Testing by including values just outside the valid range. It tests the system's ability to handle inputs slightly beyond the expected boundaries.
3. **Worst-Case Boundary Value Testing :** Worst-Case Boundary Value Testing examines the effects of all combinations of boundary values across multiple variables. It explores interactions between variables at their boundary conditions.
4. **Robust Worst-Case Boundary Value Testing:** Robust Worst-Case Boundary Value Testing combines out-of-range values for multiple variables to stress test the system. It includes extreme combinations, even those outside the valid input ranges.

2.3 Normal Boundary Value Testing

Normal Boundary Value Testing (NBVT) is a technique that focuses on testing the boundaries of the input space to uncover potential errors that often occur near extreme values of input variables. The rationale behind NBVT is to test input values at their minimum, just above the minimum, at a nominal value, just below the maximum, and at the maximum value. This approach helps to identify common errors such as off-by-one errors, incorrect conditional checks (using < instead of ≤), and misunderstandings about where counting should start (from zero or one). Normal boundary value test cases for two variables are shown in Figure 2.1.

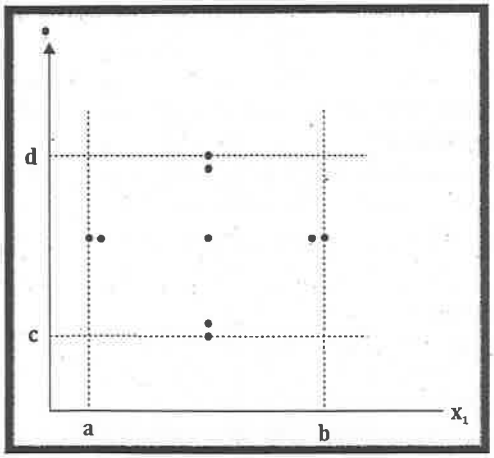


Fig 2.1 : Boundary value analysis test cases for a function of two variables

Methodology of Normal Boundary Value Testing

The testing focuses on the boundary values of input variables. This includes:

- **Min:** The minimum value the variable can take.
- **Min+:** Just above the minimum value.
- **Nom (Nominal):** A typical or expected value (often the midpoint).
- **Max–:** Just below the maximum value.
- **Max:** The maximum value the variable can take.

Application of NBVT

1. **Single Fault Assumption:** NBVT often operates under the "single fault" assumption of reliability theory, which indicates that system failures are usually due to a single fault rather than the interaction of multiple faults. This assumption simplifies the testing process by allowing the focus to be on individual variables one at a time.
2. **Test Cases Generation:** For a function with two variables, for example, the test cases would keep one variable at its nominal value and vary the other through its boundary values. For example, If we have two variables x1 and x2, then
   - Variable x1 is held at its nominal value, and x2 is tested at its min, min+, nom, max-, and max.
   - Similarly, x2 is held at its nominal value, and x1 is tested at its min, min+, nom, max-, and max.

Example 1 **Noraml Boundary Value Testing - Single Variable**

Scenario: Consider a system that grants access based on age, where only individuals aged 18 to 65 are allowed entry.

Boundary Values for Age:

- **Min (18):** Test with age 18 to verify that the system grants access.
- **Min+ (19):** Test with age 19 to ensure access is consistently granted just above the minimum age.
- **Nom (42):** A nominal test with age 42 (midpoint of the range) to check normal operation.
- **Max– (64):** Test with age 64, just below the maximum age limit to ensure access is still granted.
- **Max (65):** Test with age 65 to check that the system still grants access at the upper edge.

Testing at these boundaries helps ensure that the system accurately enforces age restrictions by allowing access to eligible individuals while denying it to those outside the age range (under 18 or over 65).

Example 2 **Noraml Boundary Value Testing - Two Variables**

Scenario: Businesses are required to pay GST monthly by submitting their total sales and selecting the applicable GST rate. The rates are variable depending on the type of goods or services provided, commonly 5%, 12%, 18%, and 28%. The system calculates the tax payable and allows businesses to submit their payments online.

Boundary Values for Variables:

- **Total Sales (x1):** Input is expected to range from ₹0 (minimum) to ₹10,00,000 (maximum), representing the sales amount for the month.
- **GST Rate (x2):** Standard GST rates applicable: 5% (min), 12%, 18%, and 28% (max).

Testing Boundary Values for Total Sales:

- **(₹0, 5%)** - Test case with no sales and the lowest GST rate (5%).
- **(₹1, 5%)** - Minimum positive sales amount with the lowest GST rate.
- **(₹5,00,000, 18%)** - Midpoint of the sales range with a commonly used GST rate (18%).
- **(₹9,99,999, 28%)** - Just below the maximum sales limit with the highest GST rate (28%).
- **(₹10,00,000, 28%)** - Maximum sales limit with the highest GST rate.

Testing across different GST rates (using a typical sales amount, e.g., ₹50,000):

- **(₹50,000, 5%)** - Testing with a lower GST rate applied to a typical sales figure.
- (₹50,000, 12%)
- (₹50,000, 18%)
- **(₹50,000, 28%)** - Testing with higher GST rates applied.

By testing the system with these specific boundary values, we can ensure that the GST calculation and payment submission process works correctly under various scenarios, including minimum, maximum, and critical points of the input ranges. This approach helps identify potential issues related to calculations, tax rates, and system behavior at the edges of the expected input values.

Example 3 **Noraml Boundary Value Testing - Three Variables**

Scenario: In the Indian railway ticket booking system, passengers can book tickets for different types of trains (such as Express, Superfast, Rajdhani) and different classes (Sleeper, 3AC, 2AC, 1AC). Each train type and class combination has a specific number of tickets available per day.

Boundary Values for Variables:

- **Train Type (x1):** Express (min), Rajdhani (max)
- **Class of Travel (x2):** Sleeper (min), 1AC (max)
- **Number of Tickets Available (x3):** Let's assume a range from 0 to 100. (Max 100 tickets)

Testing Boundary Values for Express Train:

- **(Express, Sleeper, 0)** - Test scenario with no tickets available.
- **(Express, Sleeper, 1)** - Testing just above the scenario with no availability.
- **(Express, Sleeper, 50)** - Nominal availability scenario.
- **(Express, Sleeper, 99)** - Testing just below full capacity.
- **(Express, Sleeper, 100)** - Test scenario at full capacity.

Test Cases for Rajdhani Train:

- **(Rajdhani, 1AC, 0)** - Test scenario with no tickets available.
- **(Rajdhani, 1AC, 1)** - Minimal available tickets scenario.
- **(Rajdhani, 1AC, 50)** - Nominal ticket availability scenario.
- **(Rajdhani, 1AC, 99)** - Testing almost full capacity.
- **(Rajdhani, 1AC, 100)** - Test scenario at full capacity.

By executing these test cases, we can ensure that the ticket booking system handles different scenarios related to ticket availability for Express and Rajdhani trains across various classes of travel. This approach helps in verifying the system's functionality and its ability to manage ticket availability based on the specified constraints and boundaries.

2.3.1 Generalizing the Boundary Value Analysis

Generalizing Boundary Value Analysis (BVA) in software testing refers to extending the traditional boundary value analysis technique to handle a wider range of scenarios, variables, or types of data. This generalization aims to enhance the applicability and effectiveness of BVA by adapting it to different contexts and testing requirements. The normal boundary value analysis technique can be generalized based on the number of variables and the types of ranges.

1. **Generalizing by Number of Variables:**
   - For a function with multiple variables, BVA can be generalized by holding all but one variable at their nominal (typical or expected) values while the remaining variable is tested at its boundary values. This is repeated for each variable independently.
   - For example, in a function with three variables, applying boundary value analysis by varying each variable through min, min+, nom, max–, and max values results in 4n + 1 unique test cases.
   - This approach ensures comprehensive coverage of different variable combinations and their boundary conditions.

Example **Generalizing the BVA by Number of Variables**

Suppose a function calculates a fee based on three variables: age (10 to 65 years), distance traveled (0 to 100 kilometers), and hours of service (1 to 24 hours). Testing might involve:

- Holding distance and hours at their nominal values (50 km and 12 hours), and varying age through its boundary values (10, 11, 32.5, 64, 65).
- Repeating this process for each variable, leading to a series of tests that comprehensively cover the boundary conditions for each variable.

This method ensures each variable's influence on the outcome is thoroughly examined, with 4n+1 unique test cases generated where n is the number of variables.

2. **Generalizing by Types of Ranges:**

   Variables can have different types and ranges. The nature of variables determines the ranges for boundary value analysis.
   - **Discrete and Bounded Variables:** Such as months in a year or days in a month, where boundaries are inherently defined by the domain (e.g., January to December for months).
   - **Variables without Explicit Bounds:** These require artificial boundaries. For example, if there is no upper limit specified for a numeric input, the maximum might be set as the largest representable integer.
   - **Boolean and Logical Variables:** BVA becomes less useful because they usually have only two states (True and False). These types of variables are better suited to other testing techniques like decision table testing.
   - **Context-Specific Adjustments:** For variables like a customer's PIN or transaction type in an ATM system, conventional BVA may not be very insightful or practical because these are typically categorical or have a restricted range of valid inputs.

Example **Generalizing the BVA by Types of Ranges**

- **Discrete and Bounded Variables:** A movie ticket booking system allows customers to choose a month for a special monthly screening event. The variables are Months of the year (January to December).Test cases would typically include the first month (January), the last month (December), and a mid-year month like June to cover the boundaries and a nominal value.
- **Variables without Explicit Bounds:** In the context of the triangle problem where side lengths are the variables, determining boundary values involves setting the lower bound at 1 (as negative side lengths are invalid) and selecting an upper bound, such as 200 or MAXINT. This generalization ensures that the testing covers a wide range of scenarios, including extreme values and boundary conditions, to validate the behavior of the triangle classification algorithm accurately.
- **Boolean and Logical Variables:** software application has a feature that can be toggled on or off.

  Variables: Feature state (True or False).

  Since the variable is Boolean, boundary testing directly applies to testing both states: True and False. BVA is straightforward as the tests will explicitly check the system's behavior when the feature is enabled (True) and disabled (False).

2.3.2 Limitations of Boundary Value Analysis

Boundary Value Analysis (BVA) is a fundamental software testing technique used to identify errors at the boundaries of input domains. By testing values at the edges of valid ranges, BVA aims to uncover faults that may arise due to boundary conditions. While BVA is effective for functions with independent variables representing bounded physical quantities, it may have limitations when dealing with complex dependencies or non-physical variables. Understanding the principles and constraints of BVA is essential for testers to design comprehensive test cases and ensure the reliability and quality of software systems.

Let us understand the limitations of Boundary Value Analysis (BVA) and how certain conditions can make it less effective or inappropriate

1. **Requirement for Ordering Relations :** BVA is most effective when the variables involved have a natural ordering, meaning it is logical to determine that one value is greater than, less than, or equal to another. This is crucial for defining boundary values meaningfully.

   Example: Temperature and pressure have a natural ordering. For instance, 0°C can be logically compared to 100°C (0°C ≤ 100°C). In contrast, sets of colors or names of football teams do not have an intrinsic order. It's not logical to assert that "Red" is less than "Blue" or that "Team A" is greater than "Team B".

2. **Independence of Variables :** BVA assumes variables are independent, but this is not always the case. Dependencies between variables can lead to complex interactions that BVA might not adequately test.

   Example: In the context of the NextDate function, the validity of a date depends on the interactions between day, month, and year. February 29 is a valid date but only in a leap year, highlighting a dependency between the day and year that traditional BVA might overlook.

3. **Focus on Physical Quantities :** Boundary value analysis is most suitable for variables representing physical quantities like temperature, pressure, or air speed, where physical boundaries play a crucial role.

   Example: The closure of Bangalore International Airport due to temperatures exceeding the maximum value shows how critical physical boundaries are. Here, BVA could have identified potential issues with instrument settings at extreme temperatures.

4. **Challenges with Logical Variables :** Logical or categorical variables, such as PIN numbers or telephone numbers, do not benefit from BVA due to the lack of physical boundaries or meaningful extremities.

   Example: Consideration of logical variables such as Personal Identification Numbers (PINs) or telephone numbers may not reveal significant faults through boundary value testing. Testing PIN values like 0000, 0001, 5000, 9998, and 9999 may not uncover substantial issues due to the nature of logical, non-physical variables.

5. **Inadequacy in Handling Complex Dependencies :** BVA may fail to account for complex dependencies within the system, which could lead to significant oversights in testing.

   Example: Imagine a digital thermostat that controls both heating and cooling in a smart home system. The thermostat is programmed to switch on heating when the temperature drops to 18°C or lower and activate cooling when the temperature rises to 26°C or higher.

   In typical BVA, we might test the thermostat's response at 18°C and 26°C separately to ensure it triggers the heating and cooling systems correctly. However, suppose the thermostat experiences rapid temperature changes, fluctuating between 17°C and 27°C in a short period due to unusual weather conditions or HVAC issues.

   This scenario could test the thermostat's ability to handle quick switching between heating and cooling, a condition not covered by simple boundary tests for individual temperatures. If there's a delay or failure in switching modes under rapid fluctuation, the system might fail to maintain a stable room temperature, potentially causing discomfort or even damaging the HVAC system due to the rapid cycling of heating and cooling.

2.4 Robust Boundary Value Testing

Robust boundary value testing (RBVT) is a simple extension of normal boundary value testing: in addition to the five boundary value analysis values of a variable, we see what happens when the inputs are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min–).

Robust Boundary Value Testing (RBVT) is an extension of Normal Boundary Value Testing (NBVAT) that aims to enhance test coverage by considering values beyond the boundaries. RBVT includes values slightly outside the boundary limits to ensure the software behaves robustly even with inputs that are close to the edges. This approach helps identify potential vulnerabilities and corner cases that may not be captured by Normal Boundary Value Testing.

This approach is designed to assess the robustness of a system by observing how it handles inputs that fall outside the expected input range. It aims to uncover potential failures that could occur due to inputs that users may not typically provide but could potentially be used either maliciously or by mistake.

Key Components of RBVT

1. **Standard Boundary Values:**
   - **Min:** The smallest value within the acceptable range.
   - **Min+:** A value just above the minimum to verify edge cases within the operational range.
   - **Nominal:** A typical value expected during regular use.
   - **Max-:** A value just below the maximum to test the upper limits of normal operation.
   - **Max:** The largest value within the acceptable range.
2. **Extended Test Values:**
   - **Min-:** A value slightly less than the minimum accepted input, testing the system's error handling or validation processes.
   - **Max+:** A value slightly greater than the maximum accepted input, similarly aimed at probing the robustness of error handling and input validation.

Importance of RBVT

Robust Boundary Value Testing is crucial for systems where input validation directly impacts functionality and security. By including tests for inputs just outside the accepted ranges, RBVT helps ensure that the application is secure against unusual or unexpected inputs, enhancing the overall resilience and reliability of the system. This testing approach is particularly valuable in protecting against errors that could lead to exceptions, system crashes, or security breaches.

Robust boundary value test cases for two variables are shown in Figure 2.2.

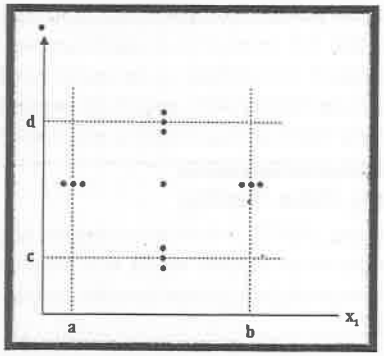


Fig 2.2 : Robustness test cases for a function of two variables

Example **Robust Boundary Value Testing (RBVT) - Single Variable**

Scenario: An online application form requires users to enter their age, which should be between 18 and 65 years inclusive.

Define Boundary and Extended Values:

- **Min (18 years):** Check that the form accepts the minimum age.
- **Max (65 years):** Ensure that the form accepts the maximum age.

- **Min- (17 years):** The form should reject this input, ideally with a clear error message.
- **Max+ (66 years):** Similar to Min-, the form should not accept this age and should provide an error message.

Test Cases:

- **At Minimum (18 years):** Verify the form processes this input correctly.
- **Just Above Minimum (19 years):** Confirm the form continues to function correctly slightly above the minimum.
- **Nominal (42 years):** A common age to test the form under typical conditions.
- **Just Below Maximum (64 years):** Test the upper operational limits.
- **At Maximum (65 years):** Ensure the maximum boundary is respected.
- **Below Minimum (17 years):** The system should identify and reject this out-of-bounds input.
- **Above Maximum (66 years):** Similarly, this should be rejected to confirm robust boundary handling.

Example **Robust Boundary Value Testing (RBVT) - Two Variables**

Scenario: This system allows users to book a certain number of rooms for a specified number of nights. The valid range for the number of rooms is from 1 to 10, and for the number of nights from 1 to 30.

Variables:

Number of Rooms (x1): Min: 1 room **Max:** 10 rooms

Number of Nights (x2): Min: 1 night **Max:** 30 nights

Test Cases:

1. **Extended Boundary Values:**
   - **For Number of Rooms:**
     Min-: 0 rooms (test system's reaction to an invalid lower boundary)
     Max+: 11 rooms (test system's reaction to exceeding the maximum room limit)
   - **For Number of Nights:**
     Min-: 0 nights (similarly tests for invalid input below the minimum)
     Max+: 31 nights (tests for exceeding the maximum night limit)
2. **Standard Boundary Values:**
   - **For Number of Rooms:**
     Min (1 room), Min+ (2 rooms), Nominal (5 rooms, assuming a mid-point or typical selection), Max- (9 rooms), Max (10 rooms)
   - **For Number of Nights:**
     Min (1 night), Min+ (2 nights), Nominal (15 nights, assuming a mid-point or typical stay length), Max- (29 nights), Max (30 nights)

Test Cases Combinations:

By applying RBVT, test cases would be generated by combining these boundary and beyond-boundary conditions for both variables. Examples of specific test cases might include:

- (1 room, 1 night) – Minimum of both variables.
- (10 rooms, 30 nights) – Maximum of both variables.
- (0 rooms, 31 nights) – Beyond the minimum and maximum for both variables.

- (11 rooms, 0 nights) – Beyond the maximum for rooms and below minimum for nights.
- (1 room, 31 nights) – Minimum rooms but above maximum nights.
- (10 rooms, 0 nights) – Maximum rooms but below minimum nights.

These tests ensure that the booking system can handle not only typical scenarios but also react appropriately to edge cases and improper inputs. It tests the system's error handling capabilities and validates that improper inputs do not cause crashes or incorrect behavior, which is critical for maintaining system integrity and user trust.

2.5 Worst-Case Boundary Value Testing

Worst-Case Boundary Value Testing (WCBVT) is a testing approach that goes beyond traditional boundary value testing by considering extreme values for multiple variables simultaneously. This method aims to explore scenarios where more than one variable reaches its boundary limits to assess the software's behavior under such conditions. By analyzing worst-case scenarios, testers can uncover potential vulnerabilities that may not be evident with single-variable boundary testing.

Unlike Robust Boundary Value Testing, which tests each variable independently at and just outside its boundaries, WCBVT involves the Cartesian product of the boundary values of all variables. This approach is designed to detect issues that may arise specifically from interactions between variables at their extreme operational limits.

Key Points

1. **Rejecting Single-Fault Assumption:** Worst-case boundary value testing challenges the single-fault assumption by examining the impact of extreme values on multiple variables. This approach is particularly useful in scenarios where the failure of the software due to extreme conditions can have severe consequences.
2. **Generating Test Cases:** To conduct worst-case boundary value testing, testers start with a five-element set for each variable, including the minimum, slightly above minimum, nominal, slightly below maximum, and maximum values. By taking the Cartesian product of these sets for multiple variables, a comprehensive set of worst-case test cases is generated.
3. **Comparison with Normal Boundary Value Testing:** Worst-case boundary value testing is more exhaustive than normal boundary value testing as it considers extreme values for multiple variables simultaneously. The number of test cases generated for worst-case testing is significantly higher (5^n for n variables) compared to normal boundary value testing (4n + 1 test cases).

Key Components of WCBVT

1. **Standard Boundary Values:**
   - **Min:** The smallest value within the acceptable range for each variable.
   - **Min+:** A value just above the minimum, within the operational range.
   - **Nominal:** A typical or average value expected during regular usage.
   - **Max-:** A value just below the maximum, still within the operational limits.
   - **Max:** The largest value within the acceptable range for each variable.

2. **Cartesian Product of Boundary Values:**

   This approach multiplies the boundary scenarios of each variable with every boundary scenario of the other variables, producing a comprehensive set of test cases that explore interactions between variables at their boundary conditions.

Importance of WCBVT

1. WCBVT is important in environments where multiple variables interact in complex ways, potentially impacting the system's behavior under extreme conditions. By systematically testing all combinations of boundary values, WCBVT can uncover issues that might not be visible when variables are tested in isolation or only within their normal operational ranges.
2. This type of testing is particularly useful in critical systems where failure can result in significant consequences, ensuring that the system is robust against a wide range of inputs and conditions. It's essential for ensuring the reliability and stability of systems in real-world scenarios where multiple factors may affect outcomes simultaneously.

The result of the two-variable version of this is shown in Figure 2.3.

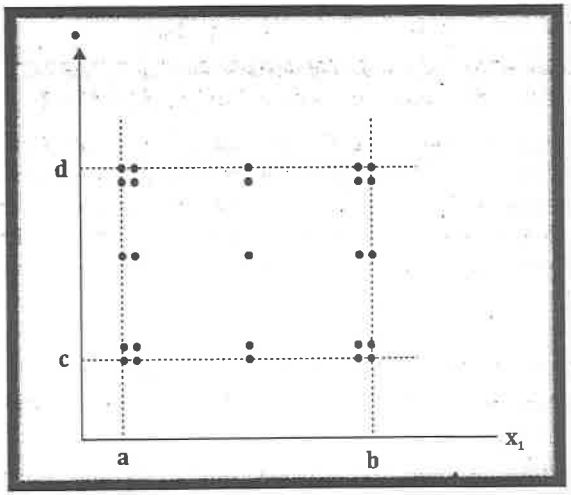


Fig 2.3 : Worst-case test cases for a function of two variables

Example **Worst Case Boundary Value Testing (WCBVT)**

Scenario: A system manages an online promotional campaign where users can enter the number of items they wish to purchase and select a delivery option. The valid range for the number of items is from 1 to 20, and the delivery options are categorized into regular (1) and express (2).

Variables:

- **Number of Items (x1):**

  Min: 1 item

  Max: 20 items

- **Delivery Option (x2):**

  Min: 1 (regular)

  Max: 2 (express)

Test Cases: Apply the boundary values for each variable and create combinations using the Cartesian product:

- **Boundary Values for Number of Items:** [1 (Min), 2 (Min+), 10 (Nominal), 19 (Max-), 20 (Max)]
- **Boundary Values for Delivery Options:** [1 (Min, regular), 2 (Max, express)]

Test Case Combinations: Each combination of the above boundary values forms a test case, such as:

- (1 item, regular)
- (1 item, express)
- (20 items, regular)
- (20 items, express)
- (2 items, regular)
- (19 items, express)

And so on, through all possible combinations of these boundary values.

2.6 Robust Worst-Case Boundary Value Testing (RWCBVT)

Robust Worst-Case Boundary Value Testing (RWCBVT) takes the principles of both Robust Boundary Value Testing and Worst-Case Boundary Value Testing to create an even more stringent testing environment. This approach focuses on evaluating the software's behavior under extreme conditions while also testing for robustness against unexpected inputs and variations in the boundary values of multiple variables simultaneously.

Key Points

1. **Incorporation of Non-valid Inputs:** RWCBVT includes values beyond the normal operating range for each variable, testing how the system handles inputs that are typically considered invalid.
2. **Cartesian Product Including Out-of-Range Values:** The approach involves taking the Cartesian product of extended boundary values (which include values beyond the typical range) for multiple variables. This extensive combination aims to simulate potential extreme scenarios that could arise in actual operations. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in 7^n test cases.
3. **Comprehensive Testing Scope:** This method is the most exhaustive form of boundary value testing, examining not only the interactions between variables at their defined limits but also how these variables behave when pushed beyond these points. The number of test cases for RWCBVT can be significantly higher, especially when extended boundary conditions are considered.

Key Components of RWCBVT:

1. **Standard Boundary Values:**
   - **Min:** The smallest value within the acceptable range for each variable.
   - **Min+:** A value just above the minimum, to verify the system's behavior just within the operational range.
   - **Nominal:** A typical or average value expected during regular usage.
   - **Max-:** A value just below the maximum, testing the near-upper limit operations.
   - **Max:** The largest value considered normal for each variable.
2. **Extended Boundary Values:**
   - **Min-:** A value slightly less than the minimum, testing how the system handles inputs below the acceptable range.
   - **Max+:** A value slightly above the maximum, probing the robustness of the system's upper limits
3. **Cartesian Product of All Boundary Values:**

   This methodology multiplies every scenario, including out-of-range scenarios, across all variables, creating a comprehensive set of test cases to explore how variable interactions might affect the system under extreme and unexpected conditions.

Importance of RWCBVT:

1. **Detecting Hidden Vulnerabilities:** By pushing the system beyond its intended operational limits, RWCBVT can uncover hidden issues that might not be evident during standard testing. This is critical for systems where safety and security are paramount.
2. **Ensuring System Resilience:** This testing is crucial for ensuring that the system can handle erroneous inputs without crashing or behaving unpredictably, which is especially important in high-stakes environments like medical, aerospace, and financial systems.

The below Figure 2.4 shows the robust worst-case test cases for our two-variable function.

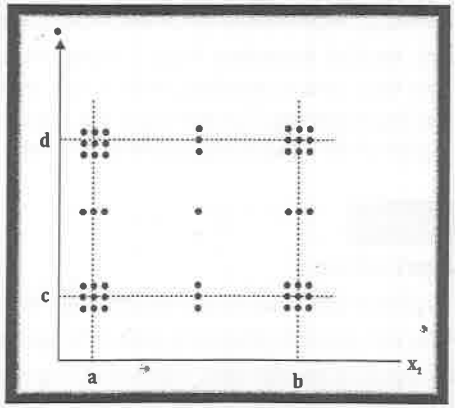


Fig 2.4 : Robust worst-case test cases for a function of two vriables

Example **Robust Worst Case Boundary Value Testing (RWCBVT)**

Scenario: A flight booking system that allows users to select the number of travelers and class of service. The system typically handles up to 5 travelers and offers three classes of service (Economy, Business, First Class).

Variables:

- **Number of Travelers (x1):**
  - **Min:** 1 traveler
  - **Max:** 5 travelers
- **Class of Service (x2):**
  - **Min:** 1 (Economy)
  - **Max:** 3 (First Class)

Test Cases: Apply both the typical and extended boundary values for each variable and create combinations using the Cartesian product:

- **Boundary Values for Number of Travelers:** [0 (Min-), 1 (Min), 3 (Nominal), 5 (Max), 6 (Max+)]
- **Boundary Values for Class of Service:** [0 (Min-), 1 (Economy), 2 (Business), 3 (First Class), 4 (Max+)]

Test Case Combinations: Each combination of the above boundary values forms a test case, such as:

- **(0 travelers, Economy)** - Testing below minimum travelers with the lowest class.
- **(1 traveler, 4 classes)** - Minimum travelers with beyond maximum class.
- **(6 travelers, 0 class)** - Above maximum travelers with non-valid class.
- **(5 travelers, First Class)** - Maximum valid travelers and maximum valid class.

And so on, through all possible combinations of these boundary values.These combinations ensure that every possible extreme and out-of-bound scenario is tested, providing insights into how well the system can maintain functionality and reliability under adverse conditions.

2.7 Special Value Testing

Special Value Testing also known as ad hoc testing is a form of functional testing that relies on the tester's domain knowledge, experience with similar programs, and understanding of potential weak points in the software to design test cases. This approach is highly dependent on the tester's judgment and expertise as no specific guidelines are followed other than best engineering practices.

Special Value Testing can be valuable in uncovering faults that may not be easily detected by other testing methods. Testers using this approach often consider unique or critical scenarios that may not be covered by traditional boundary value testing.

Characteristics of Special Value Testing:

1. **Tester-Driven:** Special Value Testing heavily relies on the tester's judgment, experience, and expertise. The effectiveness of this testing approach is highly dependent on the individual tester's capabilities and insights in identifying critical test scenarios.

2. **Domain Knowledge:** Testers leverage their in-depth understanding of the application's domain to anticipate potential error-prone areas. By applying domain knowledge, testers can pinpoint specific functionalities or modules where defects are more likely to occur and focus their testing efforts accordingly.
3. **Ad Hoc Approach:** Special Value Testing follows an ad hoc approach, lacking standardized procedures or guidelines. Testers have the flexibility to design test cases based on their intuition, experience, and knowledge without strict adherence to predefined testing methodologies. This creative freedom allows testers to explore unique scenarios that may not be covered by traditional testing techniques.
4. **Targeting "Soft Spots":** Special Value Testing aims to target "soft spots" within the software, which are areas known to be prone to errors or vulnerabilities. These soft spots may include complex calculations, unusual input types, historically problematic modules, or functionalities with a higher likelihood of defects. By focusing on these critical areas, testers can uncover hidden issues that might not be revealed through standard testing methods.

Importance of Special Value Testing:

This method is particularly valuable for:

1. **Uncovering Rare Issues:** By focusing on specific, often rare conditions that are not typically covered by other testing methods.
2. **Highly Contextual Applications:** Effective in complex systems where the tester's deep understanding of the application can guide the testing process.
3. **Areas Prone to Errors:** Particularly useful in areas known for their susceptibility to bugs, where testers can apply their insights to explore specific scenarios thought to be risky.

Example **Special Value Testing**

Scenario: The "NextDate" function calculates the next day's date given a specific day, month, and year. This function can be particularly tricky around the boundaries of months, changes in year, and especially during leap years.

In the context of the NextDate function, Special Value Testing may involve creating test cases related to specific dates such as February 28, February 29 in leap years, and other scenarios that are not covered by standard boundary value testing. While this method may lack the systematic approach of boundary value testing, it can be effective in revealing faults and vulnerabilities in the software.

Special Value Test Cases:

- **February 28 in a Non-Leap Year:** Test what happens on the day after February 28. The expected result should be March 1 of the same year.
- **February 28 in a Leap Year:** Here, the next day should be February 29.
- **February 29 in a Leap Year:** Testing the transition from February 29 to March 1 in a leap year is crucial since February 29 does not exist in non-leap years.
- **December 31:** Testing the transition from December 31 to January 1 of the following year to verify that the function handles year changes correctly.

2.8 Examples

2.8.1 Test Cases for the Triangle problem

IIn the context of the triangle problem, boundary value testing is a crucial testing technique that focuses on testing the boundaries of input ranges to uncover potential defects. When conducting robust boundary value testing for the triangle problem, it is essential to consider the lower and upper bounds of the input ranges for the sides of the triangle.

In this case, the lower bound for the sides is set at 1, and an arbitrary upper bound of 200 is chosen. The test values selected for each side are {1, 2, 100, 199, 200}. Additionally, robust boundary value test cases will include values just outside the boundaries, such as {0, 201}, to ensure thorough testing.

It is important to note that when designing boundary value test cases, care should be taken to avoid redundancy and ensure comprehensive coverage of all possible scenarios. Test cases should be diverse and cover all possible outcomes, including scenarios for equilateral, isosceles, scalene triangles, and cases where a triangle is not formed.

The cross-product of test values can result in a large number of test cases, some of which may be repeated. It is crucial to manage the test cases effectively to avoid duplication and ensure efficient testing coverage. Table 2.2 likely presents a subset of the most critical boundary value test cases, providing a snapshot of the comprehensive testing approach applied to the triangle problem.

Table 2.1 Normal Boundary Value Test Cases

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	200	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

Table 2.2 (Selected) Worst-Case Boundary Value Test Cases

Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	100	Not a triangle
4	1	1	199	Not a triangle
5	1	1	200	Not triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	100	Not a triangle
9	1	2	199	Not a triangle
10	1	2	200	Not a triangle
11	1	100	1	Not a triangle
12	1	100	2	Not a triangle
13	1	100	100	Isosceles
14	1	100	199	Not a triangle
15	1	100	200	Not a triangle
16	1	199	1	Not a triangle
17	1	199	2	Not a triangle
18	1	199	100	Not a triangle
19	1	199	199	Isosceles
20	1	199	200	Not a triangle
21	1	200	1	Not a triangle
22	1	200	2	Not a triangle
23	1	200	100	Not a triangle
24	1	200	199	Not a triangle
25	1	200	200	Isosceles

2.8.2 Test cases for the next date function

When designing test cases for the NextDate function, it is essential to consider various scenarios to ensure comprehensive testing coverage. Here are some example test cases that can be used to validate the functionality of the NextDate function:

1. **Normal Boundary Value Test Cases:**

   Test Case 1: Inputting the date "01 01 1812" should return "01 02 1812" as the next date.

   Test Case 2: Inputting the date "12 31 2012" should return "01 01 1813" as the next date.

   Test Case 3: Inputting the date "02 28 2020" should return "02 29 2020" as the next date (leap year).

Test Case 4: Inputting the date "02 29 2020" should return "03 01 2020" as the next date.

Test Case 5: Inputting the date "04 30 2021" should return "05 01 2021" as the next date.

2. **Invalid Input Test Cases (Robust Boundary Value Testing)**

   Test Case 6: Inputting the date "13 01 2020" should return an error message indicating an invalid month.

   Test Case 7: Inputting the date "02 30 2021" should return an error message indicating an invalid day for February.

   Test Case 8: Inputting the date "02 29 2021" should return an error message indicating an invalid date for a non-leap year.

3. **Edge Cases Test Cases:**

   Test Case 9: Inputting the date "01 01 1812" should return "01 02 1812" as the next date.

   Test Case 10: Inputting the date "12 31 2012" should return "01 01 1813" as the next date.

   Test Case 11: Inputting the date "06 30 2020" should return "07 01 2020" as the next date.

   Test Case 12: Inputting the date "09 30 2021" should return "10 01 2021" as the next date.

4. **Leap Year Test Cases:**

   Test Case 13: Inputting the date "02 28 2021" should return "03 01 2021" as the next date.

   Test Case 14: Inputting the date "02 29 2020" should return "03 01 2020" as the next date.

These test cases cover a range of scenarios including boundary values, invalid inputs, edge cases, and leap year considerations to ensure thorough testing of the NextDate function. Additional test cases can be designed based on specific requirements and functionalities of the function.

There are three variables and worst case boundary value testing requires $5^3 = 125$ test cases. All 125 worst-case test cases for NextDate are listed in Table 2.3.

Table 2.3 Worst-Case Test Cases

Case	Month	Day	Year	Expected Output
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012
16	1	30	1812	1, 31,1812
17	1	30	1813	1, 31, 1813
18	1	30	1912	1, 31, 1912
19	1	30	2011	1, 31, 2011
20	1	30	2012	1, 31, 2012
21	1	31	1812	2, 1, 1812
22	1	31	1813	2, 1, 1813
23	1	31	1912	2, 1, 1912
24	1	31	2011	2, 1, 2011
25	1	31	2012	2, 1, 2012
26	2	1	1812	2, 2, 1812
27	2	1	1813	2, 2, 1813
28	2	1	1912	2, 2, 1912
29	2	1	2011	2, 2, 2011
30	2	1	2012	2, 2, 2012
31	2	2	1812	2, 3, 1812
32	2	2	1813	2, 3, 1813
33	2	2	1912	2, 3, 1912
34	2	2	2011	2, 3, 2011
35	2	2	2012	2, 3, 2012
36	2	15	1812	2, 16, 1812
37	2	15	1813	2, 16, 1813
38	2	15	1912	2, 16, 1912
39	2	15	2011	2, 16, 2011
40	2	15	2012	2, 16, 2012
41	2	30	1812	Invalid date
42	2	30	1813	Invalid date
43	2	30	1912	Invalid date
44	2	30	2011	Invalid date
45	2	30	2012	Invalid date
46	2	31	1812	Invalid date

47	2	31	1813	Invalid date
48	2	31	1912	Invalid date
49	2	31	2011	Invalid date
50	2	31	2012	Invalid date
51	6	1	1812	6, 2, 1812
52	6	1	1813	6, 2, 1813
53	6	1	1912	6, 2, 1912
54	6	1	2011	6, 2, 2011
55	6	1	2012	6, 2, 2012
56	6	2	1812	6, 3, 1812
57	6	2	1813	6, 3, 1813
58	6	2	1912	6, 3, 1912
59	6	2	2011	6, 3, 2011
60	6	2	2012	6, 3, 2012
61	6	15	1812	6, 16, 1812
62	6	15	1813	6, 16, 1813
63	6	15	1912	6, 16, 1912
64	6	15	2011	6, 16, 2011
65	6	15	2012	6, 16, 2012
66	6	30	1812	7, 1, 1812
67	6	30	1813	7, 1, 1813
68	6	30	1912	7, 1, 1912
69	6	30	2011	7, 1, 2011
70	6	30	2012	7, 1, 2012
71	6	31	1812	Invalid date
72	6	31	1813	Invalid date
73	6	31	1912	Invalid date
74	6	31	2011	Invalid date
75	6	31	2012	Invalid date
87	11	15	1813	11, 16, 1813
88	11	15	1912	11, 16, 1912
89	11	15	2011	11, 16, 2011
90	11	15	2012	11, 16, 2012
91	11	30	1812	12, 1, 1812
92	11	30	1813	12, 1, 1813
93	11	30	1912	12, 1, 1912
94	11	30	2011	12, 1, 2011

95	11	30	2012	12, 1, 2012
96	11	31	1812	Invalid date
97	11	31	1813	Invalid date
98	11	31	1912	Invalid date
99	11	31	2011	Invalid date
100	11	31	2012	Invalid date
101	12	1	1812	12, 2, 1812
102	12	1	1813	12, 2, 1813
103	12	1	1912	12, 2, 1912
104	12	1	2011	12, 2, 2011
105	12	1	2012	12, 2, 2012
106	12	2	1812	12, 3, 1812
107	12	2	1813	12, 3, 1813
108	12	2	1912	12, 3, 1912
109	12	2	2011	12, 3, 2011
110	12	2	2012	12, 3, 2012
111	12	15	1812	12, 16, 1812
112	12	15	1813	12, 16, 1813
113	12	15	1912	12, 16, 1912
114	12	15	2011	12, 16, 2011
115	12	15	2012	12, 16, 2012
116	12	30	1812	12, 31, 1812
117	12	30	1813	12, 31, 1813
118	12	30	1912	12, 31, 1912
119	12	30	2011	12, 31, 2011
120	12	30	2012	12, 31, 2012
121	12	31	1812	1, 1, 1813
122	12	31	1813	1, 1, 1814
123	12	31	1912	1, 1, 1913
124	12	31	2011	1, 1, 2012
125	12	31	2012	1, 1, 2013

2.8.3 Test Cases for the Commission Problem

When creating test cases for the Commission Problem, it is important to cover various scenarios to ensure the accuracy and reliability of the program. Here are some example test cases that can be used to validate the functionality of the Commission program:

1. **Boundary Value Test Cases:**

   Test Case 1: Inputting the sales of locks as 0, stocks as 0, and barrels as 0 should result in zero total sales and zero commission.

   Test Case 2: Inputting the sales of locks as 200, stocks as 200, and barrels as 200 should test the upper bounds of the sales values.

   Test Case 3: Inputting the sales of locks as 201, stocks as 201, and barrels as 201 should test values just outside the upper bounds.

2. **Valid Input Test Cases:**

   Test Case 4: Inputting the sales of locks as 50, stocks as 30, and barrels as 20 should calculate the total sales and commission accordingly.

   Test Case 5: Inputting the sales of locks as 100, stocks as 150, and barrels as 75 should test different sales combinations.

3. **Invalid Input Test Cases:**

   Test Case 6: Inputting negative values for sales of locks, stocks, or barrels should result in an error message.

   Test Case 7: Inputting non-numeric values for sales should prompt the user to enter valid numeric inputs.

4. **Edge Cases Test Cases:**

   Test Case 8: Inputting the sales of locks as 1, stocks as 1, and barrels as 1 should test the lower bounds of the sales values.

   Test Case 9: Inputting the sales of locks as 200, stocks as 0, and barrels as 0 should test the scenario where only one type of product is sold.

5. **Commission Calculation Test Cases:**

   Test Case 10: Inputting the sales of locks as 100, stocks as 50, and barrels as 25 should verify the commission calculation based on the given prices for each product.

These test cases cover a range of scenarios including boundary values, valid inputs, invalid inputs, edge cases, and commission calculations to ensure thorough testing of the Commission program. Additional test cases can be designed based on specific requirements and functionalities of the program.

2.9 Random Testing

Random Testing is a methodology within the field of software testing where inputs are generated randomly to test the system rather than selecting them based on any predetermined criteria such as boundary conditions or typical values. This approach is unique in its reliance on randomness to uncover errors. It a potentially powerful tool for identifying hidden issues in the software.

Random testing is a valuable approach to software testing that involves using a random number generator to select test case values. This method helps in avoiding bias in testing and can uncover unexpected issues in the software.

Overview of Random Testing:

1. **Statistical Basis:** Random testing is often discussed within academic due to its statistical nature. By using random inputs, the testing process attempts to simulate a broad spectrum of user interactions, potentially uncovering less obvious faults.
2. **Use of Random Number Generators:** Inputs for testing are selected using random number generators to ensure that the values are spread across the entire input domain of the variable being tested. This method helps in mitigating tester biases that might influence the choice of test data.
3. **Challenge of Determining Test Adequacy:** One of the critical challenges with random testing is deciding how many test cases are sufficient to confidently assert the software's reliability. This decision can be somewhat subjective and often requires statistical or risk-based approaches to resolve.
4. **Considerations for Random Testing:** Random testing can be particularly useful for exploring a wide range of inputs and scenarios that may not be covered by traditional test cases. It is important to ensure that the random number generator used is truly random and provides a uniform distribution of values. Random testing can be combined with other testing techniques to achieve comprehensive test coverage. The effectiveness of random testing can be influenced by the quality of the random number generator and the size of the input domain.
5. **Determining the Number of Random Test Cases:** One common question in random testing is how many random test cases are sufficient to provide adequate test coverage. The number of random test cases needed can vary depending on factors such as the complexity of the software, the size of the input domain, and the quality of the random number generator. In practice, running a large number of random test cases can help increase confidence in the robustness of the software.

Advantages of Random Testing:

1. **Comprehensive Coverage:** Random testing can potentially cover a wide range of input scenarios than manual selection methods.
2. **Unbiased Testing:** It reduces the likelihood of unconscious bias in choosing test cases, which might overlook certain types of errors.

Limitations:

1. **Less Efficiency:** It may require a large number of tests to achieve sufficient coverage, especially for variables with a wide range of possible values.
2. **Difficulty in Reproducing Errors:** Randomly generated test cases can make it hard to reproduce failures unless the specific inputs causing the failure are recorded.

Example **Random Testing for "NextDate" Function**

Scenario: The "NextDate" function calculates the next day based on given input of day, month, and year..

Implementation:

- **Random Date Generation:** Use a random number generator to select the day (1-31), month (1-12), and year (e.g., 1900-2100), without considering the actual number of days in each month for simplicity in this illustrative example.
- **Test Execution:** For each randomly generated date, the NextDate function is executed to determine if it correctly calculates the following day's date.
- **Analysis:** Analyze outcomes to see if any date calculations break the expected logic, particularly around critical dates such as year-end, month-end, and for leap years.

Example **Random Testing for Traingle Problem**

The below table represents the sample distribution of test cases for different types of triangles (Nontriangles, Scalene, Isosceles, Equilateral) along with the total number of test cases for each scenario. Additionally, the percentage distribution of each type of triangle is also given.

Test Cases	Nontriangles	Scalene	Isoscles	Equilateral
1289	663	593	32	1
15,436	7696	7372	367	1
17,091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

1. **Total Test Cases:** The total number of test cases varies for each set of inputs, ranging from 1289 to 17091.
2. **Distribution of Triangle Types:**
   - **Nontriangles:** The majority of test cases fall under the category of Nontriangles, with counts ranging from 663 to 8556.
   - **Scalene:** The Scalene triangles have a significant number of test cases, with counts ranging from 593 to 8164.
   - **Isosceles:** The number of Isosceles triangles is lower compared to Nontriangles and Scalene, with counts ranging from 32 to 367.
   - **Equilateral:** Equilateral triangles have the least number of test cases, with counts of 1 in most scenarios.
3. **Percentage Distribution:** The percentage distribution provides insights into the relative frequency of each type of triangle in the test cases.

4. **Analysis:** The data indicates that the test cases are skewed towards Nontriangles and Scalene triangles, which are more common in general triangle scenarios. The low percentage of Isosceles and Equilateral triangles suggests that these types of triangles are less frequently encountered in the test cases provided. It is important to ensure a balanced distribution of test cases across different types of triangles to thoroughly test the program's functionality for all possible scenarios.

2.10 Guidelines for Boundary Value Testing

Boundary value testing is a crucial aspect of software testing that focuses on testing the boundaries of input and output ranges of a program. This technique can be especially useful when testing for edge cases, error message handling, and robustness of internal variables like loop controls and pointers.

Key Guidelines for Boundary Value Testing:

1. **Understand Variable Relationships:** Make sure that variables in the software are independent of each other. If they are not independent, consider how they affect each other to avoid unrealistic test scenarios. For example, ensure that dates like June 31 are not accepted.
2. **Apply Testing Widely:** Extend boundary value testing to not only input ranges but also output values and internal variables like indices. Test error messages and output values to ensure they are within expected limits. Use robust testing for internal variables to catch common errors.
3. **Use Semantic Understanding:** Understand the real-world purpose of the software function being tested. This helps in creating test cases that are more relevant and avoid impossible scenarios. For instance, knowing that a function calculates mileage per litre petrol can help in avoiding negative values or division by zero.
4. **Create Diverse Test Cases:** Develop test cases that cover a range of values, including minimum, maximum, values just inside these boundaries, typical values, and values slightly beyond the boundaries. This comprehensive approach ensures thorough testing.
5. **Avoid Strict Technical Focus:** Move away from solely technical testing methods and consider the practical application of the software. By incorporating semantic information, test scenarios become more meaningful. For example, when testing a banking application, ensuring that a user cannot transfer a negative amount of money or transfer funds without having sufficient balance adds practical value to the testing process.

By following these guidelines, testers can conduct effective boundary value testing that covers a wide range of scenarios, considers real-world applications, and ensures the software functions correctly in various situations.

Example **Practical Example: Adjusting Test Cases Based on Semantic Knowledge**

Scenario: A banking application calculates interest earned on a savings account based on the average daily balance and the annual interest rate.

Function: I = (A * R)/100

Where:

A = average daily balance

R = annual interest rate (as a percentage)

I = interest earned per day

Revised Boundary Value Test Cases Based on Semantic Insights:

- **Avoid Negative Balances:** Ensure A≥0. Test cases where A is zero, slightly above zero, and typical savings balance but not negative.
- **Avoid Negative Interest Rates:** Interest rates should not be negative in this context. Test for R=0%,0.1%, and a higher typical rate like 5%, but ensure no test case includes a negative rate.
- **Handle Zero Interest Rate:** Include scenarios where R=0% to ensure that the function correctly calculates zero interest earned.
- **Test Boundary Values for Average Balance:** Introduce test cases that specifically examine the impact of very low and very high balances on the interest calculation to evaluate its accuracy across the range of possible balances**Example Test Cases:**

1. **Zero Balance Test:** A=0, R=5%; Expect I=0 since there's no balance to accrue interest.
2. **Minimum Interest Rate Test:** A=10,000, R=0.1%; Check interest calculation for an extremely low interest rate.
3. **High Balance Test:** A=100,000, R=5%; Expect a higher interest calculation reflecting the significant balance.
4. **Zero Interest Rate Test:** A=50,000, R=0%; Ensure that the interest earned is correctly calculated as zero despite a substantial average balance.

These tailored test cases help ensure that the banking application reliably calculates interest under various realistic scenarios, from typical to edge cases, thus ensuring accuracy and robustness in its financial computations. This methodology leverages semantic knowledge to create more relevant and impactful tests, focusing on likely real-world usage scenarios that could significantly impact user experience and financial accuracy.

2.11 Review Questions

Section A

Each Question Carries Two Marks

1. What is Input Domain Function ?
2. What is Boundary Value Testing ?
3. Mention the Types of Boundary Value Testing.
4. What is Normal Boundary Value Testing (NBVT)?
5. What is Robust Boundary Value Testing (RBVT)?
6. What is Worst-Case Boundary Value Testing (WCBVT)?
7. What is Robust Worst-Case Boundary Value Testing (RWCBVT)?
8. What is Special Value Testing? Give an example.
9. What is Random Testing? Give an example

Section B

Each Question Carries Five Marks

1. What is Boundary Value Testing ? Explain The Key Concepts of Boundary Value Testing.
2. Why Use Boundary Value Testing ? What is an Importance of Boundary Value Testing?
3. Explain Generalizing Boundary Value Analysis (BVA).
4. Explain the Limitations of Boundary Value Analysis.
5. What is Normal Boundary Value Testing (NBVT)? Explain its methadology.
6. What is Robust Boundary Value Testing (RBVT)? Explain Key Components and Importance of RBVT.
7. What is Worst-Case Boundary Value Testing (WCBVT)? Explain Key Components and Importance of WCBVT.
8. What is Robust Worst-Case Boundary Value Testing (RWCBVT)? Explain Key Components and Importance of WCBVT.
9. Explain the Generation of Test Cases for the Triangle problem in Boundary Value Testing.
10. Explain the Generation of Test Cases for the NextDate Function in Boundary Value Testing.
11. Explain the Generation of Test Cases for the Commission Problem in Boundary Value Testing.
12. Explain Random Testing. Mention its advantages and disadvantages.
13. Explain Random Testing for "NextDate" Function.
14. Explain Random Testing for Traingle Problem.
15. Explain the Key Guidelines for Boundary Value Testing.

Section C

Each Question Carries Eight Marks

1. Explain Normal Boundary Value Testing (NBVT) with an example.
2. Explain Robust Boundary Value Testing (RBVT) with an example.
3. Explain Worst-Case Boundary Value Testing (WCBVT) with an example.
4. Explain Robust Worst-Case Boundary Value Testing (RWCBVT) with an example.
5. What is Special Value Testing? Explain Characteristics and Importance of Special Value Testing.

❖❖❖❖❖

Note

UNIT - II

CHAPTER 3

EQUIVALENCE CLASS TESTING

Contents

3.1 Introduction

In the previous chapter, we explored boundary value testing as a technique to ensure thorough test coverage by focusing on boundary conditions. In this chapter, we will discuss equivalence class testing as another essential method for functional testing. Equivalence class testing involves grouping input values into classes to streamline test case design and enhance testing efficiency. By categorizing inputs that are treated similarly by the software system, equivalence class testing aims to achieve comprehensive testing coverage while minimizing redundancy. This chapter will provide a detailed examination of equivalence class testing, its principles, and its various forms to further enhance our understanding of effective testing strategies.

Traditional equivalence class testing originated during the dominance of languages like FORTRAN and COBOL, where errors related to invalid inputs were common, ultimately influencing the evolution of strongly typed languages to address such issues.

3.2 Equivalence Classes

Equivalence Class Testing (ECT) is a method used in software testing where the input domain is divided into classes of data from which test cases are derived. Each class is expected to be representative of a group of inputs that behave similarly in the system, hence testing just one input from each class should be representative of the entire class. This approach helps optimize the number of test cases, aiming to cover all possible scenarios with minimal redundancy. **Equivalence Class Testing (ECT) is also called as Equivalence Partition Testing (EPT).**

Example : In the context of the triangle problem, for instance, testing for an equilateral triangle can be effectively represented by using the input values (5, 5, 5). In this scenario, additional test cases like (6, 6, 6) or (100, 100, 100) would not provide significant new insights as they would essentially yield the same outcome. This intuitive understanding of redundancy in test cases is crucial for optimizing testing efforts.

Motivations Behind Equivalence Class Testing

1. **Sense of Complete Testing:** ECT aims to ensure every functional aspect of the application is tested by covering all equivalence classes.
2. **Avoid Redundancy:** By focusing on one representative from each class rather than multiple similar inputs, ECT reduces unnecessary test cases.

? What do you mean by Equivalence Classes ?

Equivalence classes refer to groups of input values that are treated the same way by the software system being tested. Test cases are designed to represent each equivalence class to ensure that the system behaves consistently for all values within that class. This approach helps in reducing the number of test cases needed while maintaining thorough test coverage. This allows for efficient test case creation while achieving good test coverage.

? What is an Equivalence Class Testing?

Equivalence Class Testing (ECT) is a method used in software testing that helps to efficiently partition the input or output spaces into classes that are treated equivalently by the system under test. By identifying and utilizing representative samples from these classes, testers can effectively reduce redundancy while ensuring comprehensive coverage.

Understanding Equivalence Classes

1. **Partitioning:** The concept of partitioning in the context of equivalence classes means dividing a set into exclusive and exhaustive subsets. Each element of the set belongs to one and only one subset. This partitioning is key to ensuring that tests are both comprehensive and non-redundant.
2. **Mutual Disjointness:** The subsets are mutually disjoint, meaning no two subsets share an element. This property ensures that each test case derived from each subset is unique, thereby reducing redundancy in testing.
3. **Common Properties:** Each subset in an equivalence class contains elements that are assumed to have something in common—typically, how the software behaves when presented with these elements as inputs. This assumption allows testers to use a single test case from each subset to infer the behavior for all elements of that subset.

Core Idea

- Divides the entire range of possible input values for a program input into distinct partitions called equivalence classes.
- Each equivalence class represents a group of input values where the program's behaviour is expected to be the same.
- Test cases are designed to target each equivalence class with at least one representative value.

Equivalence Class Testing Assumptions

Equivalence class testing considers two primary factors:

- **Robustness:** Tests are designed to handle both valid and invalid inputs, checking the system's ability to handle unexpected or erroneous data.
- **Single/Multiple Fault Assumption:** Determines whether the testing assumes that errors are caused by a single fault or multiple faults simultaneously.

Importance of Equivalence Class Testing

Equivalence class testing is crucial in software testing for several reasons:

1. **Comprehensive Test Coverage:** By organizing input values into equivalence classes, testers can ensure that representative test cases are selected to cover different scenarios. This approach helps in identifying defects across various input conditions, leading to more thorough testing coverage.

2. **Efficiency in Test Case Design:** Equivalence class testing allows testers to reduce the number of test cases needed while maintaining effective coverage. By focusing on representative values from each equivalence class, redundant test cases can be minimized, optimizing testing efforts and resources.
3. **Effective Bug Detection:** Equivalence class testing helps in uncovering defects and vulnerabilities in the software system by testing different equivalence classes. By exploring how the system handles inputs within each class, testers can identify potential issues and ensure the system behaves as expected under various conditions.
4. **Risk Mitigation:** By systematically categorizing input values into equivalence classes, testers can prioritize testing efforts based on the criticality of each class. This approach helps in mitigating risks associated with different input scenarios and ensures that high-risk areas are thoroughly tested.
5. **Alignment with Testing Principles:** Equivalence class testing aligns with fundamental testing principles such as robustness and the single/multiple fault assumption. By focusing on how the system treats inputs within each class, testers can validate the system's behavior and identify potential weaknesses or inconsistencies.

Four Forms of Equivalence Class Testing

- **Weak Normal :** Assumes a single fault and focuses on valid inputs.
- **Strong Normal:** Assumes multiple faults can occur simultaneously and focuses on valid inputs.
- **Weak Robust:** Assumes a single fault but includes both valid and invalid inputs.
- **Strong Robust:** Assumes multiple faults and includes both valid and invalid inputs.

Aligning Equivalence Class and Boundary Value Testing

- The equivalence class testing reduces gaps and redundancies in functional testing. By combining it with boundary value testing (especially for bounded variables) can enhance test coverage. This hybrid approach, sometimes referred to as **"edge testing,"**. This helps to identify edge cases at the boundaries of equivalence classes. For example, testing right at the transition between valid and invalid triangles can uncover boundary-specific issues that might otherwise go undetected.
- Equivalence class testing, particularly when integrated with boundary value testing, provides a robust framework for ensuring that all functional paths are explored and that the software behaves as expected across all theoretically possible inputs. This strategy is critical for effective software testing.

Example 1 **Equivalence Class Testing**

1. Suppose we have an application that accepts a user's age as input, and the valid age range is from 18 to 60. We can apply Equivalence Partitioning to divide the input data into three partitions:
   - **Partition 1:** Invalid values below 18 - This partition includes all values less than 18, such as -10, 0, and 17. Testing these values will verify that the system correctly rejects invalid inputs.
   - **Partition 2:** Valid values between 18 and 60 - This partition includes all values between 18 and 60, such as 25, 35, and 50. Testing these values will verify that the system correctly accepts valid inputs.

   - **Partition 3:** Invalid values above 60 - This partition includes all values greater than 60, such as 75, 100, and 200. Testing these values will verify that the system correctly rejects invalid inputs.
2. For each partition, we can create one or more test cases to cover all possible scenarios. For example, we can test the following inputs for each partition:
   - **Partition 1:** -10, 0, 17
   - **Partition 2:** 18, 25, 35, 50, 60
   - **Partition 3:** 75, 100, 200
3. By applying Equivalence Partitioning, we have reduced the number of test cases required to test the software system, while still ensuring that all possible scenarios are covered. This technique is useful for testing complex systems where testing all possible inputs would be impractical or impossible.

Example 2 **Equivalence Class Testing**

In the triangle classification problem, equivalence classes can be based on the types of triangles:

- **Equilateral:** All sides are equal.
- **Isosceles:** Two sides are equal.
- **Scalene:** No sides are equal.
- **Invalid:** Combinations of side lengths that do not form a triangle.

For each class, a single test case is chosen:

- **Equilateral:** (5, 5, 5)
- **Isosceles:** (5, 5, 3)
- **Scalene:** (4, 5, 6)
- **Invalid:** (1, 2, 3) – where the sum of two sides does not exceed the third side.

These choices reduce test redundancy, as testing with other numbers that still fit these definitions (e.g., (6, 6, 6) for equilateral) is unlikely to provide additional insights since the application treats all instances of each class equivalently.

3.2.1 Traditional Equivalence Class Testing

Traditional Equivalence Class Testing (TECT) is a foundational method in software testing that focuses on categorizing input data into equivalence classes based on their validity. This approach is rooted in the principle of **"Garbage In, Garbage Out"** (GIGO), which underscores the importance of input validation to ensure accurate and expected outputs from software systems.

Most of the standard testing texts of equivalence classes based on valid and invalid variable values. Traditional equivalence class testing is nearly identical to weak robust equivalence class. This traditional form focuses on invalid data values.

During the early programming time, the principle of "Garbage In, Garbage Out" (GIGO) underscored the significance of providing valid data for program execution, as results based on invalid data were unpredictable. In response to GIGO, programmers implemented extensive input validation mechanisms within programs to ensure data integrity. However, with the evolution of modern programming languages featuring strong data typing and the adoption of graphical user interfaces (GUIs), the need for extensive input data validation has diminished. User-friendly interface elements like drop-down lists and slider bars have contributed to reducing the occurrence of erroneous input data.

Traditional equivalence class testing aligns with the process of boundary value testing and involves testing functions for valid values of variables first. Subsequently, testing is conducted with invalid values for individual variables while keeping the remaining variables valid. This iterative approach helps in identifying faults related to invalid data values.

Process of Traditional Equivalence Class Testing

The testing process typically follows these steps, which aim to isolate issues related to input validation:

1. **Test with Valid Inputs:** Begin by testing the function F with all variables set to their valid values. This initial step ensures that the function behaves as expected under normal conditions.
2. **Test with Invalid Inputs:** After confirming that the function performs correctly with valid inputs, the next step involves introducing invalid values for one variable at a time while keeping other variables at their valid states.

   For instance, if testing a function F(x1,x2), start by substituting x1 with its invalid values while keeping x2 within its valid range. This helps determine if the function can handle errors gracefully when one input is incorrect.

   This step is repeated for each variable, ensuring that each one is tested for its response to invalid data.
3. **Identify Faults Due to Invalid Data:** Any failures detected during the testing of invalid inputs can typically be attributed to issues handling these inputs. This pinpointed approach helps in identifying specific vulnerabilities associated with the processing of invalid data.

The below Figure 3.1 shows test cases for a function F of two variables x1 and x2.

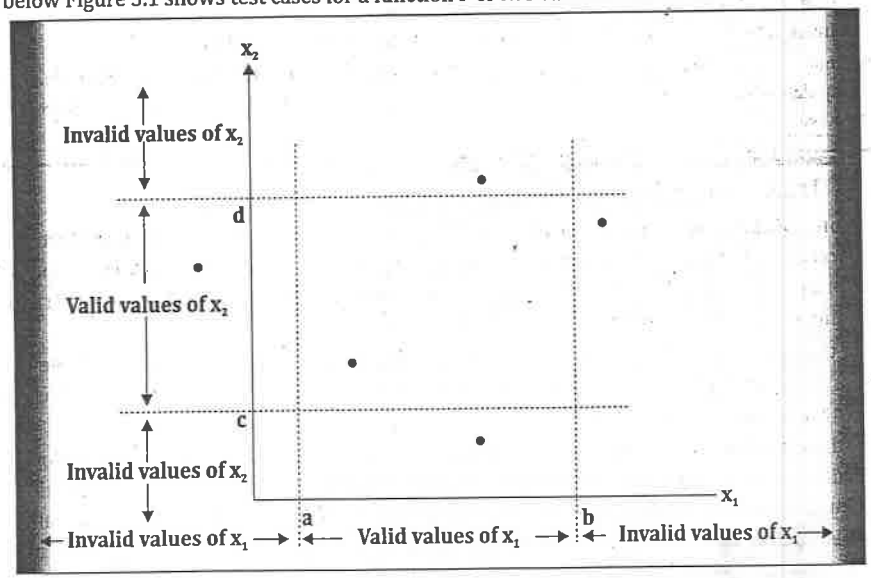


Figure 3.1 : Traditional Equivalence Class Test Cases

Advantages of Traditional Equivalence Class Testing

1. **Focused Fault Identification:** By systematically testing valid and then invalid inputs, TECT efficiently identifies how well the program handles erroneous data, which is crucial for robust error handling and validation logic in software applications.
2. **Reduction in Redundant Testing:** Since each variable is tested independently for validity, there is minimal redundancy, making the testing process more efficient.

3.3 Forms or Variations of Equivalence Class Testing

Equivalence class testing include four main forms, each with its own focus and assumptions.

1. **Weak Normal Equivalence Class Testing:**
   - Assumes a single fault and concentrates on testing valid inputs only.
   - Designed to verify the system's behavior under normal operating conditions with valid input values.
2. **Strong Normal Equivalence Class Testing:**
   - Assumes the possibility of multiple faults occurring simultaneously and emphasizes testing valid inputs.
   - Aims to validate the system's response to various valid input scenarios, considering the potential for multiple faults.
3. **Weak Robust Equivalence Class Testing:**
   - Assumes a single fault but includes both valid and invalid inputs in the testing process.
   - Focuses on evaluating the system's resilience to faults by testing both valid and invalid input values.
4. **Strong Robust Equivalence Class Testing:**
   - Assumes the presence of multiple faults and incorporates both valid and invalid inputs in the testing strategy.
   - Seeks to uncover system vulnerabilities by testing a combination of valid and invalid input values under the assumption of multiple faults.

Each form of equivalence class testing serves a specific purpose in software testing, ranging from validating system behavior under normal conditions to assessing its robustness in the face of faults and invalid inputs. By employing these different forms of equivalence class testing, testers can enhance test coverage, identify potential defects, and ensure the reliability and quality of the software system.

3.3.1 Weak Normal Equivalence Class Testing

Weak Normal Equivalence Class Testing (WNECT) is a fundamental technique in software testing, particularly within the domain of equivalence class testing. This method is termed "weak" because it operates under the assumption of a single fault, implying that any failure is attributed to an issue within a single input variable at a time. WNECT aims to simplify the testing process by reducing the number of test cases to a manageable few, each representing different equivalence classes or intervals of input variables.

WNECT is particularly useful in situations where the input domain is large and can be clearly segmented into meaningful classes based on the functionality and requirements of the system. It is widely used in regression testing, where the goal is to quickly validate that no new errors have been introduced in previously tested parts of the software.

? What is Weak Normal Equivalence Class Testing (WNECT) ?

Weak Normal Equivalence Class Testing (WNECT) is a software testing technique that assumes a single fault and concentrates on testing valid inputs only. It is called "weak" because it assumes that any failure is caused by a problem in just one input variable at a time. It is specifically designed to verify the system's behavior under normal operating conditions with valid input values. It simplifies the testing process and ensure comprehensive coverage of different input categories.

Key Characteristics of WNECT

1. **Equivalence Classes:** Inputs are divided into groups (or classes) where each group represents a set of values that the system should theoretically treat the same. These classes are defined based on both the input value ranges (valid or invalid) and their expected behaviors.
2. **Single Fault Assumption:** WNECT operates under the premise that failures are due to issues with one specific input variable at a time. This approach simplifies the analysis of test results and helps focus on isolating faults in distinct areas of the system.
3. **Representative Sampling:** From each equivalence class, one representative sample is chosen for testing. The idea is that testing this single value is sufficient to infer the behavior for all values within that class, assuming the system treats all of them equivalently.

Implementation Steps in WNECT

1. **Identify Equivalence Classes:** Determine the sets of values that make up the equivalence classes based on input specifications. These can include ranges of valid values and separately, ranges of invalid values that are expected to trigger error handling mechanisms.
2. **Select Test Cases:** Choose one representative value from each class to be used in testing. This selection should ideally cover the spectrum of expected behaviors from the system when given inputs from these classes.
3. **Construct and Execute Tests:** Formulate test cases that include these selected values. Each test case will typically involve inputs from different equivalence classes to ensure coverage across the input domain.

Benefits of WNECT

1. **Efficiency:** Reduces the number of test cases needed by focusing only on representative values rather than exhaustive testing of all possible inputs.
2. **Effectiveness:** Provides a systematic approach to testing by ensuring that all defined classes of inputs are checked, thus covering different scenarios the software might encounter.

Limitations of WNECT

1. **Isolation of Faults:** While it is efficient, the single fault assumption may not always hold true, especially in complex systems where interactions between different inputs can lead to failures. This can make fault isolation challenging if a test case fails.
2. **Depth of Testing:** WNECT might not sufficiently test the interactions between various input values, potentially overlooking multi-variable defects.

Example : Weak Normal Equivalence Class Testing (WNECT)

Let's consider an example to illustrate Weak Normal Equivalence Class Testing.

Scenario : A banking application calculates interest on a savings account using the account balance and interest rate.

Equivalence Classes:

1. **Account Balance:**
   - Class A (Low Balance): ₹500 or less
   - Class B (Medium Balance): ₹501 to ₹5000
   - Class C (High Balance): ₹5001 or more
2. **Interest Rate:**
   - Class X (Low Interest Rate): 0% to 3%
   - Class Y (Medium Interest Rate): 4% to 7%
   - Class Z (High Interest Rate): 8% to 12%

Weak Normal Equivalence Class Test Cases:

1. **Test Case 1:**
   - Account Balance: ₹300 (Class A - Low Balance)
   - Interest Rate: 2% (Class X - Low Interest Rate)
2. **Test Case 2:**
   - Account Balance: ₹2500 (Class B - Medium Balance)
   - Interest Rate: 5% (Class Y - Medium Interest Rate)
3. **Test Case 3:**
   - Account Balance: ₹8000 (Class C - High Balance)
   - Interest Rate: 10% (Class Z - High Interest Rate)

Analysis:

- Weak Normal ECT focuses on individual equivalence classes with one value from each class to ensure basic coverage. It targets specific scenarios within each class to identify potential faults associated with those ranges.
- In Weak Normal Testing, each test case focuses on a single equivalence class with one value from that class. For example, Test Case 1 considers a low account balance and a low interest rate. This approach aims to identify potential faults within individual input ranges.
- Weak Normal Testing is aligned with the concept of a single fault because it targets one specific equivalence class at a time, testing for potential issues within that range. Each test case is designed to validate the system's response to valid inputs within a particular class, aiming to uncover faults associated with that specific scenario.

- Assume that if Test Case 2 fails, indicating a discrepancy between the expected interest calculation and the actual calculated interest amount, the issue could be related to the calculation of interest based on the medium balance and interest rate. The failure in Test Case 2 highlights a potential problem in the application's handling of medium balance and interest rate scenarios.
- The ambiguity in fault isolation in Weak Normal Equivalence Class Testing is evident in this scenario. While the failure identifies a problem, it does not pinpoint whether the issue lies with the medium balance, medium interest rate, or their interaction. This level of ambiguity is acceptable in certain testing scenarios, such as regression testing, where the focus is on broader system validation rather than detailed fault isolation.
- For more precise fault identification, stronger forms of equivalence class testing, like Strong Normal or Weak Robust, may be employed to delve deeper into the potential causes of failures and ensure comprehensive testing coverage.

3.3.2 Strong Normal Equivalence Class Testing

Strong Normal Equivalence Class Testing (SNECT) is an advanced method within the framework of equivalence class testing used in software testing. This approach builds on the concepts of Weak Normal Equivalence Class Testing by incorporating multiple variables simultaneously in each test case, rather than examining them individually. SNECT is designed to detect more complex interactions between variables that might not be evident when variables are tested in isolation.

? What is Strong Normal Equivalence Class Testing (SNECT) ?

Strong Normal Equivalence Class Testing (SNECT) assumes the possibility of multiple faults occurring simultaneously and emphasizes testing valid inputs. It aims to validate the system's response to various valid input scenarios, considering the potential for multiple faults. By systematically testing all unique combinations of input values, SNECT ensures comprehensive coverage of input scenarios to identify and address potential defects in the system.

Key Characteristics of Strong Normal Equivalence Class Testing

1. **Multiple Variable Integration:** Unlike weak testing, which might consider one variable at a time, strong testing involves creating test cases that combine representative values from multiple equivalence classes across different variables. This approach helps identify issues arising from the interactions between these variables.
2. **Normal Equivalence Classes:** This form of testing focuses on normal (valid) equivalence classes, meaning it uses combinations of values that are all expected to be handled correctly by the system. The purpose is to confirm that the system behaves as expected under various combinations of normal conditions.
3. **No Single Fault Assumption:** SNECT moves away from the single fault assumption prevalent in weak testing methods. By integrating multiple variables in each test case, it acknowledges that faults might be caused by complex interactions between variables rather than issues with individual inputs.

Implementation Steps in SNECT

1. **Identify Equivalence Classes:** As with other forms of equivalence class testing, the first step involves identifying all relevant equivalence classes for each input variable based on their valid value ranges and behavioral characteristics.
2. **Select Representative Samples:** Choose representative samples from each equivalence class. These selections should capture a broad range of behaviors and potential interactions between the variables.
3. **Construct Comprehensive Test Cases:** Develop test cases that include combinations of selected samples from the identified equivalence classes across all variables. This method ensures that the interactions between variables are thoroughly tested.
4. **Execute and Analyze Tests:** Execute the formulated test cases and carefully analyze the outcomes. The complexity of analyzing results increases as the interactions between multiple variables are considered.

Advantages (or) Benefits of Strong Normal Equivalence Class Testing

1. **Comprehensive Interaction Testing:** Provides a more thorough examination of how different parts of the system interact with each other, potentially uncovering hidden bugs that occur only under specific conditions involving multiple inputs.
2. **Increased Fault Detection Capabilities:** By testing combinations of inputs across multiple variables, SNECT can identify faults that may be missed by testing variables in isolation,

Limitations (or) Challenges of Strong Normal Equivalence Class Testing

1. **Increased Complexity:** The need to consider multiple combinations of inputs significantly increases the complexity of test planning and execution.
2. **Higher Resource Requirements:** The comprehensive testing requires more time and computational resources and hence the cost and duration of the testing phase is higher.

Example Strong Normal Equivalence Class Testing (SNECT)

Let's consider an example to illustrate Strong Normal Equivalence Class Testing.

Scenario : A banking application calculates interest on a savings account using the account balance and interest rate.

Equivalence Classes:

1. **Account Balance:**
   - Class A (Low Balance): ₹500 or less
   - Class B (Medium Balance): ₹501 to ₹5000
   - Class C (High Balance): ₹5001 or more
2. **Interest Rate:**
   - Class X (Low Interest Rate): 0% to 3%
   - Class Y (Medium Interest Rate): 4% to 7%
   - Class Z (High Interest Rate): 8% to 12%

Strong Normal Equivalence Class Test Cases:

1. Strong Normal Test Case 1:
   Account Balance: ₹500 (Low Balance)
   Interest Rate: 2% (Low Interest Rate)
2. Strong Normal Test Case 2:
   Account Balance: ₹500 (Low Balance)
   Interest Rate: 5% (Medium Interest Rate)
3. Strong Normal Test Case 3:
   Account Balance: ₹500 (Low Balance)
   Interest Rate: 10% (High Interest Rate)
4. Strong Normal Test Case 4:
   Account Balance: ₹2000 (Medium Balance)
   Interest Rate: 2% (Low Interest Rate)
5. Strong Normal Test Case 5:
   Account Balance: ₹2000 (Medium Balance)
   Interest Rate: 5% (Medium Interest Rate)
6. Strong Normal Test Case 6:
   Account Balance: ₹2000 (Medium Balance)
   Interest Rate: 10% (High Interest Rate)
7. Strong Normal Test Case 7:
   Account Balance: ₹10000 (High Balance)
   Interest Rate: 2% (Low Interest Rate)
8. Strong Normal Test Case 8:
   Account Balance: ₹10000 (High Balance)
   Interest Rate: 5% (Medium Interest Rate)
9. Strong Normal Test Case 9:
   Account Balance: ₹10000 (High Balance)
   Interest Rate: 10% (High Interest Rate)

Analysis:

- In Strong Normal Equivalence Class Testing, all possible combinations of input equivalence classes are tested to ensure thorough coverage. For instance, Test Case 4 examines a medium account balance with a low interest rate, while Test Case 6 tests a medium balance with a high interest rate. This approach aims to uncover potential faults arising from the interactions of multiple input variables.
- This approach provides a more detailed examination of the system's behavior under various scenarios, including interactions between different input variables.
- The difference between Weak Normal and Strong Normal testing lies in the level of coverage and depth of testing, with Strong Normal testing offering a more comprehensive and exhaustive evaluation of the system.
- Strong Normal Testing is associated with the concept of multiple faults because it explores various combinations of input ranges, allowing for the identification of potential issues resulting from the interplay between different factors. By testing multiple combinations, this approach helps uncover faults that may arise from the complex interactions of valid inputs.

3.3.3 Weak Robust Equivalence Class Testing

Weak Robust Equivalence Class Testing (WRECT) is a method within software testing that expands on the principles of Weak Normal Equivalence Class Testing by explicitly including invalid inputs alongside valid inputs in the test cases. While the weak normal approach focuses solely on valid input scenarios under the assumption of a single fault, WRECT aims to validate how the system handles both valid and invalid inputs simultaneously to assess the system's robustness and error handling capabilities.

The robust part comes from consideration of invalid values, and the weak part refers to the single fault assumption. The process of weak robust equivalence class testing is a simple extension of that for weak normal equivalence class testing—pick test cases such that each equivalence class is represented.

? What is Weak Robust Equivalence Class Testing (WRECT) ?

Weak Robust Equivalence Class Testing (WRECT) is a testing methodology that focuses on evaluating how a system handles both valid and invalid inputs, with a specific emphasis on scenarios where unexpected or erroneous inputs are provided. This approach aims to uncover vulnerabilities related to error handling, boundary conditions, and the system's robustness against various types of input. WRECT operates under the assumption that a single fault in handling invalid inputs can lead to system vulnerabilities.

Key Characteristics of Weak Robust Equivalence Class Testing

1. **Inclusion of Invalid Inputs:** WRECT introduces invalid input values as separate equivalence classes. This is done to test the system's resilience and error-handling mechanisms, ensuring that invalid inputs do not cause crashes or undesired behaviors.
2. **Single Fault Assumption:** Similar to weak normal testing, WRECT operates under the assumption that any failure is likely due to a single problematic input—whether valid or invalid—rather than complex interactions between multiple inputs.
3. **Combination of Valid and Invalid Inputs:** Test cases are designed to include both valid and invalid inputs but typically focus on changing one variable at a time to maintain simplicity and clarity in identifying the source of any issues.

Implementation Steps in WRECT

1. **Identify Equivalence Classes:** Define equivalence classes for both valid and invalid input ranges for each variable based on the system's requirements and expected behavior.
2. **Select Representative Samples:** Choose samples from both valid and invalid equivalence classes. The selection should ideally cover a broad spectrum of expected behaviors and potential error scenarios.
3. **Construct Test Cases:** Develop test cases that integrate the selected samples. Although the focus is on a single fault assumption, incorporating invalid inputs provides insights into the system's robustness.
4. **Execute and Analyze Tests:** Perform testing and meticulously analyze the outcomes to determine how well the system handles erroneous inputs alongside normal operations.

Benefits of Weak Robust Equivalence Class Testing

1. **Enhanced Error Handling Validation:** By including invalid inputs, WRECT helps verify that the system gracefully handles errors, which is crucial for maintaining stability and user satisfaction.
2. **Increased Test Coverage:** Covers a wider range of input scenarios by incorporating tests for invalid data, thereby reducing the risk of unhandled exceptions or failures in production.

Challenges of Weak Robust Equivalence Class Testing

1. **Increased Testing Complexity:** Managing and designing tests that incorporate both valid and invalid inputs can complicate the testing process and analysis of results.
2. **Resource Intensive:** Requires more comprehensive test planning and execution, potentially leading to longer testing phases and increased costs.

Example 1 Weak Robust Equivalence Class Testing (SNECT)

Scenario: Online Payment Gateway Transaction Amount Validation

The system is designed to accept transaction amounts within certain specified limits to be processed. For this example, we will define the transaction amount limits and categorize them into different equivalence classes:

Transaction Amount:

1. Minimum transaction amount allowed: ₹50
2. Maximum transaction amount allowed: ₹500,000

Defining Equivalence Classes: We will define two main classes of valid inputs and two classes of invalid inputs based on the transaction limits:

1. **Class A (Valid - Low Range):** Range: ₹50 to ₹10,000
2. **Class B (Valid - High Range):** Range: ₹10,001 to ₹500,000
3. **Class C (Invalid - Below Minimum):** Range: Less than ₹50
4. **Class D (Invalid - Above Maximum):** Range: More than ₹500,000

Select Representative Samples:

1. **Class A (Valid - Low Range):** ₹50, ₹1000, ₹5000
2. **Class B (Valid - High Range):** ₹10,001, ₹100,000, ₹200,000
3. **Class C (Invalid - Below Minimum):** ₹0, ₹30
4. **Class D (Invalid - Above Maximum):** ₹600,000, ₹900,000,

Test Cases Based on Equivalence Classes: By selecting representative samples from each of these classes, we can efficiently test how the system handles different transaction amounts: In weak robust, we select only one sample from each class.

1. **Test Case 1: Class A (Valid - Low Range) :** Transaction Amount: ₹1,000
   Expected Result: Transaction is successfully processed.
2. **Test Case 2: Class B (Valid - High Range) :** Transaction Amount: ₹100,000
   Expected Result: Transaction is successfully processed, potentially after additional validations due to the high amount.

3. **Test Case 3: Class C (Invalid - Below Minimum) :** Transaction Amount: ₹30
   Expected Result: Transaction is rejected due to being below the minimum limit.
4. **Test Case 4: Class D (Invalid - Above Maximum) :** Transaction Amount: ₹600,000
   Expected Result: Transaction is rejected due to exceeding the maximum limit.

Analysis: Implementing equivalence class testing ensures comprehensive coverage of all input scenarios using a minimal number of test cases, organized by:

- Confirming that the system properly processes valid transaction amounts within both low and high ranges.
- Ensuring the system appropriately rejects transactions outside the allowable range, thus securing the payment process.

This method efficiently streamlines the testing process by focusing on distinct categories of inputs that represent different behaviors or responses from the system.

Example 2 Weak Robust Equivalence Class Testing (SNECT)

Scenario: Online Event Registration Platform

The platform hosts events such as concerts, seminars, and workshops, some of which have age restrictions (above 18 and below 60). Users must enter their age and select their gender (male or female) during registration to verify eligibility for certain events. The system should correctly process eligible registrations and reject ineligible ones based on age, while always correctly handling gender input.

Defined Equivalence Classes:

1. **Class A - Valid Age: 18 to 60 years**
   25 years (mid-range, representing a typical adult)
   60 years (upper limit of valid age range)
2. **Class B - Invalid Age: <18 and >60**
   17 years (just below the valid age range)
   65 years (just above the valid age range)
3. **Class C - Valid Gender:** Male, Female

Representative Samples:

1. **Class A - Valid Age:**
   25 years (mid-range, representing a typical adult)
   60 years (upper limit of valid age range)
2. **Class B - Invalid Age: <18 and >60**
   17 years (just below the valid age range)
   65 years (just above the valid age range)
3. **Class C - Valid Gender:** Male, Female

Test Cases Based on Equivalence Classes: The aim here is to include both valid and invalid ages independently in test scenarios combined with valid gender entries to check the robustness of the system's user validation process. We select only one sample from each class in a test case.

1. **Test Case 1:**
   Age: 25 years (Class A - Valid Age)
   Gender: Male (Class C - Valid Gender)
   Expected Result: Successful profile creation or update, as the age is valid and gender is correctly specified.
2. **Test Case 2:**
   Age: 60 years (Class A - Valid Age)
   Gender: Female (Class C - Valid Gender)
   Expected Result: Successful profile creation or update, as the age is at the valid upper boundary and gender is correctly specified.
3. **Test Case 3:**
   Age: 17 years (Class B - Invalid Age)
   Gender: Male (Class C - Valid Gender)
   Expected Result: Rejection due to invalid age, even though the gender is valid.
4. **Test Case 4:**
   Age: 65 years (Class B - Invalid Age)
   Gender: Female (Class C - Valid Gender)
   Expected Result: Rejection due to invalid age, even though the gender is valid.

Analysis:

- For Test Cases 1 and 2: Verify that the system processes these inputs correctly without any errors. It should recognize both ages as valid and match them with the genders provided.
- For Test Cases 3 and 4: Observe how the system handles these invalid ages. It should reject these entries and provide clear and helpful error messages stating the age limitations, ensuring the user understands why the input was rejected.

3.3.4 Strong Robust Equivalence Class Testing

Strong Robust Equivalence Class Testing (SRECT) extends the concepts of both strong normal equivalence class testing and weak robust equivalence class testing by combining rigorous testing of multiple variable combinations with the inclusion of invalid inputs. This method aims to test the system's response to both valid and invalid inputs across a full range of input combinations. It focuses on identifying complex interactions that might lead to system vulnerabilities, particularly when invalid data is involved.

The robust part comes from consideration of invalid values and the strong part refers to the multiple fault assumption. We obtain test cases from each element of the Cartesian product of all the equivalence classes both valid and invalid. Unlike Weak Robust Equivalence Class Testing (WRECT), which focuses on a single fault assumption, SRECT delves deeper into the system's behavior under various fault conditions and complex interactions between inputs.

? What is Strong Robust Equivalence Class Testing (SRECT) ?

Strong Robust Equivalence Class Testing (SRECT) is a testing methodology that emphasizes evaluating a system's response to both valid and invalid inputs, particularly focusing on scenarios involving unexpected or erroneous inputs. This approach aims to identify vulnerabilities related to error handling, boundary conditions, and the system's overall robustness against a wide range of input variations. SRECT operates under the premise that multiple faults or complex interactions between valid and invalid inputs can potentially expose critical system weaknesses.

Key Characteristics of Strong Robust Equivalence Class Testing

1. **Integration of Multiple Variables:** Like strong normal testing, SRECT involves creating test cases that combine representative values from multiple equivalence classes across different variables, but it includes both valid and invalid classes.
2. **Consideration of Invalid Inputs:** SRECT explicitly includes invalid inputs within the test cases to check how the system handles error conditions and to validate error handling mechanisms robustly.
3. **No Single Fault Assumption:** This testing methodology assumes that multiple faults can occur due to interactions between several variables, including those arising from invalid inputs. It tests the system's ability to handle complex scenarios where multiple inputs might interact in unforeseen ways.

Implementation Steps in SRECT

1. **Identify Equivalence Classes:** Define all relevant equivalence classes for each input variable. This should include classes for valid ranges as well as specifically defined classes for known invalid inputs.
2. **Select Representative Samples:** For each equivalence class, select representative samples that are expected to adequately exhibit the behaviors or responses of that class. These should include typical values, boundary values, and exceptional cases (for invalid inputs).
3. **Generate Combinations Using Cartesian Product:** Apply the Cartesian product to the sets of selected samples from each class. This means every combination of selected samples from each class will be paired with every other selected sample from every other class to form test cases.
4. **Construct Test Cases:** Each result of the Cartesian product is a combination that becomes a test case. For example, if there are three classes A, B, and C with two samples each (A1, A2; B1, B2; C1, C2), the Cartesian product will result in combinations like (A1, B1, C1), (A1, B1, C2), ... (A2, B2, C2), totaling 2 x 2 x 2 = 8 combinations.
5. **Execute and Analyze Tests: :** Execute the test cases as per the constructed scenarios. Analyze the results to identify and address potential defects or vulnerabilities caused by interactions between the multiple input types.

Benefits of Strong Robust Equivalence Class Testing

1. **Enhanced Error Handling and System Robustness:** By thoroughly testing how the system responds to both normal and abnormal input combinations, SRECT helps ensure that the system is robust against a wide range of input scenarios.
2. **Comprehensive Fault Detection:** The methodology increases the likelihood of detecting hidden or unknown bugs that may not be apparent when testing inputs in isolation or only within normal operational ranges.

Challenges of Strong Robust Equivalence Class Testing

1. **Increased Testing Complexity:** The need to consider numerous combinations of both valid and invalid inputs significantly increases the complexity of test planning and execution.
2. **Higher Resource Requirements:** This exhaustive approach requires more time and computational resources, potentially increasing the cost and duration of the testing phase.

Example **Strong Robust Equivalence Class Testing (SNECT)**

Scenario: Online Event Registration Platform

The platform hosts events such as concerts, seminars, and workshops, some of which have age restrictions (above 18 and below 60). Users must enter their age and select their gender (male or female) during registration to verify eligibility for certain events. The system should correctly process eligible registrations and reject ineligible ones based on age, while always correctly handling gender input.

Defined Equivalence Classes:

1. **Class A - Valid Age: 18 to 60 years**
   - 25 years (mid-range, representing a typical adult)
   - 60 years (upper limit of valid age range)
2. **Class B - Invalid Age: <18 and >60**
   - 17 years (just below the valid age range)
   - 65 years (just above the valid age range)
3. **Class C - Valid Gender:** Male, Female

Representative Samples:

1. **Class A - Valid Age:**
   - 25 years (mid-range, representing a typical adult)
   - 60 years (upper limit of valid age range)
2. **Class B - Invalid Age: <18 and >60**
   - 17 years (just below the valid age range)
   - 65 years (just above the valid age range)
3. **Class C - Valid Gender:** Male, Female

Test Cases Based on Equivalence Classes: Test Cases Based on Equivalence Classes: To generate test cases, we take one sample from each class and combine them to see how the system handles multiple inputs at once. We will construct these combinations by pairing each sample from Class A with every sample from Classes B and C, to cover all possible scenarios.

With each class containing two samples and using the Cartesian product approach for Strong Robust Equivalence Class Testing (SRECT), we generate 2*2*2* = 8 test cases. This ensures that every possible combination of inputs from the four defined classes (A, B, C) is tested.

1. **Test Case 1: 25 years, Male**
   Expected Outcome: Successful registration, as the age is valid and gender is correctly specified.
2. **Test Case 2: 25 years, Female**
   Expected Outcome: Successful registration, as the age is valid and gender is correctly specified.
3. **Test Case 3: 60 years, Male**
   Expected Outcome: Successful registration, as the age is at the upper valid boundary and gender is correctly specified.
4. **Test Case 4: 60 years, Female**
   Expected Outcome: Successful registration, as the age is at the upper valid boundary and gender is correctly specified.
5. **Test Case 5: 17 years, Male**
   Expected Outcome: Rejection due to underage, with an appropriate error message detailing the age requirement.
6. **Test Case 6: 17 years, Female**
   Expected Outcome: Rejection due to underage, with an appropriate error message detailing the age requirement.
7. **Test Case 7: 65 years, Male**
   Expected Outcome: Rejection due to being overage, even though the gender is correctly specified.
8. **Test Case 8: 65 years, Female**
   Expected Outcome: Rejection due to being overage, even though the gender is correctly specified.

Analysis:

- For valid combinations, the system processes registrations without errors and appropriately handles valid age boundaries. For invalid age inputs, verify that the system rejects these registrations and provides clear, informative feedback to the user.
- This systematic testing approach using Strong Robust Equivalence Class Testing ensures that the event registration system is capable of handling a range of scenarios, improving overall reliability and user satisfaction by adequately managing different user inputs.

3.3.5 Weak Normal Vs Strong Normal Equivalence Class Testing

The below table highlights the differences between Weak Normal Equivalence Class Testing and Strong Normal Equivalence Class Testing.

Aspect	Weak Normal Equivalence Class Testing	Strong Normal Equivalence Class Testing
Definition	Tests each equivalence class independently by selecting a single representative value from one class at a time. This approach simplifies identifying which class is causing an issue if a test fails.	Simultaneously tests all valid combinations of representative values from multiple equivalence classes to examine how variables interact and impact the system together.

Fault Assumption	Assumes that any failure in the system can be traced back to a fault in a single input variable. This method tests each input independently to isolate issues.	Assumes that faults may occur due to interactions among multiple variables. This method tests combinations of variables to capture these interactions.
Purpose	Aims to verify that each input, when considered separately, is handled correctly by the system. It is effective for identifying and isolating errors related to individual inputs.	Aims to ensure the system behaves as expected under a variety of conditions that arise from multiple input variables being tested together.
Input Selection	One valid input value from each equivalence class.	Multiple valid input values from each equivalence class.
Coverage	Provides basic coverage of input combinations, focusing on one sample per class.	Offers more comprehensive coverage by considering multiple valid samples per class.
Complexity	Simple and straightforward approach, suitable for basic testing scenarios.	More detailed and thorough approach, suitable for complex systems or critical functionalities.
Test Case Generation	Generates fewer test cases due to selecting only one valid sample from each class.	Generates more test cases as multiple valid combinations are considered for each class.
Resource Requirements	Requires fewer resources in terms of time and effort for test case generation.	Demands more resources for test case generation and execution due to increased valid combinations.
Suitability	Suitable for initial testing phases or simple systems with limited input variations.	Suitable for comprehensive testing, especially for critical systems or functionalities with diverse valid input scenarios.
Execution Efficiency	More efficient with fewer test cases since it tests one equivalence class at a time.	Less efficient as it requires testing many combinations, increasing the number of test cases significantly.
Risk Coverage	May miss errors caused by input interactions, as it does not test input combinations.	Provides extensive risk coverage by examining how different input combinations affect system stability and functionality.

Example **Equivalence Classes**

Age less than 18 (Child)
Age between 18 and 65 (Adult)
Age greater than or equal to 65 (Senior)

Weak Normal Equivalence Class Testing: Selects one valid age value from each class (e.g., 10, 30, 70) for testing.

Strong Normal Equivalence Class Testing: Considers multiple valid age values from each class (e.g., 10, 15, 20 for Child; 30, 40, 50 for Adult; 70, 80, 90 for Senior) for testing.

3.3.6 Weak Robust Vs Strong Robust Equivalence Class Testing

The below table highlights the differences between Weak Robust Equivalence Class Testing and Strong Robust Equivalence Class Testing.

Aspect	Weak Robust Equivalence Class Testing	Strong Robust Equivalence Class Testing
Definition	Tests both valid and invalid equivalence classes, but only considers one variable or class at a time. This method aims to identify how the system handles unexpected or erroneous inputs individually.	Tests combinations of both valid and invalid inputs from multiple equivalence classes simultaneously, analyzing how errors and valid data interact and impact the system.
Fault Assumption	Operates under the single fault assumption where issues are expected to arise from individual inputs either valid or invalid, but not from their interaction.	Rejects the single fault assumption and anticipates that system vulnerabilities can result from complex interactions between multiple erroneous and correct inputs.
Purpose	To assess the system's response to individual invalid inputs along with valid inputs to ensure robust error handling and validate proper system behavior under typical use conditions.	To thoroughly evaluate the system's ability to handle and recover from multiple simultaneous input errors, ensuring resilience and stability under adverse conditions.
Input Selection	Includes both invalid and valid inputs but tests them independently to isolate the effect of each type of input.	Integrates multiple invalid and valid inputs in complex scenarios to observe potential compound effects and system responses.
Coverage	Provides a detailed analysis of the system's ability to handle specific types of errors individually but does not cover interactions between errors.	Offers comprehensive coverage that includes both the individual and combined effects of erroneous inputs, providing a deeper insight into potential system weaknesses.
Complexity	Relatively less complex as it involves testing one type of input error at a time.	More complex due to the need to manage and interpret the effects of multiple input errors occurring simultaneously.
Test Case Generation	Generates a moderate number of test cases, focusing on the effect of individual erroneous inputs combined with standard operations.	Generates a large number of test cases due to the extensive combinations of both erroneous and correct inputs being tested together.
Resource Requirements	Less resource-intensive compared to strong robust testing, as fewer combinations are tested.	More resource-intensive, requiring significant time and computational power to execute and analyze all possible combinations.
Suitability	Ideal for initial phases of testing to quickly identify and fix straightforward input-related vulnerabilities.	Best suited for final testing phases or in high-risk environments where system failure can have serious consequences, necessitating exhaustive testing.

Execution Efficiency	More efficient with quicker test execution due to simpler test scenarios.	Less efficient, with more extensive and time-consuming test execution required.
Risk Coverage	Provides good insight into the system's behavior with specific errors but may miss issues caused by error interactions.	Ensures extensive risk coverage, detecting even subtle issues that arise only under complex error conditions.

Example

Equivalence Classes:

- **Age Classifications:**

  Valid: Adults (18-65)

  Invalid: Below minimum (<18), Above maximum (>65)

Weak Robust Testing:

- **Test Cases:**

  Age 17 (Invalid, Below Minimum) and Gender Male (Valid)

  Age 70 (Invalid, Above Maximum) and Gender Female (Valid)

Strong Robust Testing:

- **Test Cases:**

  Combinations of Age 17, Age 70 (Invalids) with Ages 25, 30 (Valids), and Genders Male, Female in multiple configurations to check all interactions.

3.4 Equivalence Class Test Cases for the Triangle Problem

The Triangle Problem involves categorizing triangles based on their side lengths. Given three integers a, b, and c, which represent the sides of a triangle, the task is to determine whether they form an Equilateral, Isosceles, Scalene, or not a triangle at all.

Definitions of Triangle Types:

- **Equilateral:** All three sides are equal.
- **Isosceles:** Exactly two sides are equal.
- **Scalene:** All sides are different.
- Not a Triangle: The sum of the lengths of any two sides must be greater than the length of the third side.

Equivalence Classes:

We can use the above definitions to identify output (range) equivalence classes as follows:

- R1 = {<a, b, c>: the triangle with sides a, b, and c is equilateral}
- R2 = {<a, b, c>: the triangle with sides a, b, and c is isosceles}
- R3 = {<a, b, c>: the triangle with sides a, b, and c is scalene}
- R4 = {<a, b, c>: sides a, b, and c do not form a triangle}

Test Case Generation:

1. **Weak Normal Equivalence Class Testing (WNECT):**

   This focuses on testing each valid equivalence class independently.Four weak normal equivalence class test cases, chosen arbitrarily from each class are as follows:

Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a triangle

2. **Strong Normal Equivalence Class Testing (SNECT):**

   SNECT typically involves creating combinations of inputs from multiple equivalence classes. However, in the case of the Triangle Problem, each set of side lengths can only belong to one type of triangle due to the distinct and non-overlapping mathematical conditions that define each class. The testing combinations of different equivalence classes as typically done in SNECT does not apply here because the conditions for one class inherently exclude the conditions for the others. This means that a test case designed for one class (like Equilateral) cannot simultaneously be a test case for another class (like Scalene). Therefore, the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

3. **Weak Robust Equivalence Class Testing (WRECT):** It Includes invalid equivalence classes but tests them individually with valid classes. Considering the invalid values for a, b, and c yields the following additional weak robust equivalence class test cases. (The invalid values could be zero, any negative number, or any number greater than 200.)

Test Case	a	b	c	Expected Output
WR1	-1	5	5	Value of a is not in the range of permitted values
WR2	5	-1	5	Value of b is not in the range of permitted values
WR3	5	5	-1	Value of c is not in the range of permitted values

4. **Strong Robust Equivalence Class Testing (SRCT):** Tests combinations of valid and invalid classes together. The test cases SR1 to SR7 combine values from both valid and invalid classes to generate a thorough set of potential scenarios testing for robustness.

Test Case	a	b	c	Expected Output
SR1	-1	5	5	Value of a is not in the range of permitted values
SR2	5	-1	5	Value of b is not in the range of permitted values
SR3	5	5	-1	Value of c is not in the range of permitted values
SR4	-1	-1	5	Value of a, b are not in the range of permitted values
SR5	5	-1	-1	Value of b, c are not in the range of permitted values
SR6	-1	5	-1	Value of a, c is not in the range of permitted values
SR7	-1	-1	-1	Values of a, b, c are not in the range of permitted values

Analysis:

Equivalence Class Testing in the context of the Triangle Problem helps ensure thorough coverage by targeting specific types of triangles and their boundary conditions. By structuring test cases around the concept of equivalence classes, we ensure that each possible triangle configuration is tested, along with robust handling of invalid inputs. This structured approach aids in confirming that the triangle categorization logic is correctly implemented and resilient to incorrect or extreme input values.

3.5 Equivalence Class Test Cases for the NextDate Function

The NextDate function calculates the next day's date given a current date composed of day, month, and year inputs. This problem is ideal for demonstrating the application of equivalence class testing due to the variety of rules governing dates, such as varying days per month and adjustments for leap years.

Equivalence Classes for the NextDate Function

1. **Valid Equivalence Classes**

   M1 = {month: month has 30 days}

   M2 = {month: month has 31 days}

   M3 = {month: month is February}

   D1 = {day: $1 \leq$ day ≤ 28}

   D2 = {day: day = 29}

   D3 = {day: day = 30}

   D4 = {day: day = 31}

   Y1 = {year: year = 2000}

   Y2 = {year: year is a non-century leap year}

   Y3 = {year: year is a common year}

2. **Invalid Equivalence Classes:**

   M4 = {month: month < 1}

   M5 = {month: month > 12}

   D5 = {day: day < 1}

   D6 = {day: day > 31}

   Y4 = {year: year < 1812}

   Y5 = {year: year > 2012}

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting.

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

Test Case Generation:

1. **Weak Normal Equivalence Class Testing (WNECT):**

   (WNECT) might involve testing valid single dates across a variety of typical scenarios such as the end of the month, leap years, and year transitions.

Case ID	Month	Day	Year	Expected Output
WN1	6	15	1912	6/16/1912

This test case is for the Weak Normal Equivalence Class Testing where the input date is June 15, 1912. The expected result should be the next day, which is June 16, 1912. Here, the function moves the day forward by one without changing the month or year, which is the basic operation for most days in a month under this function.

2. **Strong Normal Equivalence Class Testing (SNECT):**

   SNECT is designed to test interactions between different classes, in this specific case of date processing, it is similar to WNECT because the basic function of moving from one day to the next doesn't combine different class attributes in the input. Each test inherently processes the transition between days correctly under the given rules.

Case ID	Month	Day	Year	Expected Output
SN1	6	15	1912	6/16/1912

The date provided is a regular day in the middle of a month, so it tests the basic increment function of the NextDate logic without crossing the boundary conditions of month-end or year-end, and without the additional complexity of leap year calculations. This ensures that the fundamental date increment logic is functioning correctly.

3. **Weak Robust Equivalence Class Testing (WRECT):** It includes invalid inputs alongside valid ones, but typically one invalid class at a time alongside valid ones to see if the system handles exceptions (e.g., invalid dates) correctly.

Case ID	Month	Day	Year	Expected Output
WR1	6	15	1912	6/6/1912
WR2	-1	15	1912	Value of month not in the range 1 ... 12
WR3	13	15	1912	Value of month not in the range 1 ... 12
WR4	6	-1	1912	Value of day not in the range 1 ... 31
WR5	6	32	1912	Value of day not in the range 1 ... 31
WR6	6	15	1811	Value of year not in the range 1812 ... 2012
WR7	6	15	2013	Value of year not in the range 1812 ... 2012

4. **Strong Robust Equivalence Class Testing (SRCT):** Tests combinations of invalid and valid inputs to simulate errors occurring in multiple inputs simultaneously, checking if multiple faults lead to proper error handling.

Case ID	Month	Day	Year	Expected Output
SR1	-1	15	1912	Value of month not in the range 1 ... 12
SR2	6	-1	1912	Value of day not in the range 1 ... 31
SR3	6	15	1811	Value of year not in the range 1812 ... 2012
SR4	-1	-1	1912	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31
SR5	6	-1	1811	Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012
SR6	-1	15	1811	Value of month not in the range 1 ... 12 Value of year not in the range 1812 ... 2012
SR7	-1	-1	1811	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31 Vvalue of year not in the range 1812 ... 2012

Analysis

Each type of equivalence class testing brings a different level of rigor to the testing process:

- Weak Normal and Strong Normal are often similar for functions like NextDate where a transition from one valid state to another inherently tests the logic of crossing boundaries (e.g., from month to month).
- Weak Robust and Strong Robust testing are crucial for applications like date calculations, where input validation is critical, and handling of invalid inputs must be robust to prevent data corruption or crashes.

By setting up these classes and designing tests based on them, testers can ensure comprehensive coverage of both typical use cases and edge cases, improving the software's reliability and user satisfaction.

Detailed Test Cases for Weak Normal and Strong Normal Equivalence Class Testing

Case ID	Month	Day	Year	Expected Output
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	Invalid input date
WN4	6	31	2000	Invalid input date

Case ID	Month	Day	Year	Expected Output
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996

SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	Invalid input date
SN8	6	30	1996	Invalid input date
SN9	6	30	2002	Invalid input date
SN10	6	31	2000	Invalid input date
SN11	6	31	1996	Invalid input date
SN12	6	31	2002	Invalid input date
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/31/2002
SN19	7	30	2000	7/31/2000
SN20	7	30	1996	7/31/1996
SN21	7	30	2002	8/1/2000
SN22	7	31	2000	8/1/2000
SN23	7	31	1996	8/1/1996
SN24	7	31	2002	8/1/2002
SN25	2	14	2000	2/15/2000
SN26	2	14	1996	2/15/1996
SN27	2	14	2002	2/15/002
SN28	2	29	2000	3/1/2000
SN29	2	29	1996	3/1/1996
SN30	2	29	2002	Invalid input date
SN31	2	30	2000	Invalid input date
SN32	2	30	1996	Invalid input date
SN33	2	30	2002	Invalid input date
SN34	2	31	2000	Invalid input date
SN35	2	31	1996	Invalid input date
SN36	2	31	2002	Invalid input date

When transitioning from weak to strong normal testing, as well as from weak to strong robust testing, the issue of redundancy often arises, similar to what is observed in boundary value testing. The move

from weak to strong testing assumes independence, leading to a cross-product of equivalence classes. This results in a larger number of test cases to cover all possible combinations of inputs.

1. **Strong Normal Equivalence Class Test Cases (36 Test Cases):**
   3 month classes x 4 day classes x 3 year classes = 36 test cases
2. **Strong Robust Equivalence Class Test Cases (150 Test Cases):**
   Including 2 invalid classes for each variable results in 150 test cases (too many to show here)

3.6 Equivalence Class Test Cases for the Commission Problem

The Commission Problem involves calculating the sales commission for a salesperson based on the number of locks, stocks, and barrels sold. The inputs to this problem have natural partitions based on the quantity ranges of the items sold and special sentinel values to control input iterations. The problem complexity arises from combining these quantities into a commission calculation that varies based on predefined sales thresholds.

Equivalence Classes

1. **Valid Input Classes:**
   L1= {locks: 1 ≤ locks ≤ 70}
   L2= {locks = −1} (occurs if locks = −1 is used to control input iteration)
   S1 = {stocks: 1 ≤ stocks ≤ 80}
   B1= {barrels: 1 ≤ barrels ≤ 90}
2. **Invalid Input Classes:**
   L3= {locks: locks = 0 OR locks < −1}
   L4= {locks: locks > 70}
   S2 = {stocks: stocks < 1}
   S3 = {stocks: stocks > 80}
   B2= {barrels: barrels < 1}
   B3= {barrels: barrels > 90}
3. Output Range Classes Based on Sales Calculation:
   S1: Sales ≤ $1000.
   S2: $1000 < Sales ≤ $1800.
   S3: Sales > $1800.

Tes Case Generation

1. **Weak Normal Equivalence Class Testing (WNECT) :** Focuses on testing each class independently with a typical or boundary value.

Case ID	Locks	Stocks	Barrels	Expected Output
WN1	10	10	10	$100

2. **Strong Normal Equivalence Class Testing (SNECT):** Tests all combinations of valid classes; however, due to the nature of this function, the test cases may be similar to WN as it doesn't inherently combine variables differently.

Case ID	Locks	Stocks	Barrels	Expected Output
SN1	10	10	10	$100

3. **Weak Robust Equivalence Class Testing (WRECT) :** Includes both valid and one type of invalid input at a time to test system robustness.The variable "locks" is also used as a sentinel to indicate no more telegrams. When a value of −1 is given for locks, the values of totalLocks, totalStocks, and totalBarrels are used to compute sales, and then commission.

Case ID	Locks	Stocks	Barrels	Expected Output
WR1	10	10	10	$100
WR2	-1	40	45	Program terminates
WR3	-2	40	45	Value of locks not in the range 1 ... 70
WR4	71	40	45	Value of locks not in the range 1 ... 70
WR5	35	-1	45	Value of locks not in the range 1 ... 80
WR6	35	81	45	Value of locks not in the range 1 ... 80
WR7	35	40	-1	Value of locks not in the range 1 ... 90
WR8	35	40	91	Value of locks not in the range 1 ... 90

4. **Strong Robust Equivalence Class Testing (SRCT) :** Tests combinations of valid and invalid inputs to simulate potential real-world errors.

Case ID	Locks	Stocks	Barrels	Expected Output
SR1	-2	40	45	Value of locks not in the range 1 ... 70
SR2	35	-1	45	Value of locks not in the range 1 ... 80
SR4	35	40	-2	Value of locks not in the range 1 ... 90
SR5	-2	-1	45	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80
SR6	35	-1	-1	Value of locks not in the range 1 ... 70 Value of barrels not in the range 1 ... 90
SR7	-2	-1	-1	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90

Analysis

- Weak and Strong Normal Testing primarily ensures that all valid data combinations correctly compute the commission based on sales rules without encountering invalid data.
- Robust Testing Variants (Weak and Strong) assess the system's response to invalid inputs, essential for ensuring stability and error management in real-world scenarios.

These tests collectively provide a thorough check of the commission problem by not only validating correct computations but also ensuring the system gracefully handles invalid or unusual inputs. This comprehensive approach helps ensure that all potential edge cases and data errors are managed correctly, critical for maintaining system reliability and user trust.

3.7 Guidelines and Observations About Equivalence Class Testing

The guidelines and observations about equivalence class testing are:

1. **Comprehensive Testing Levels:** Weak equivalence class testing (normal or robust) may not cover all scenarios as effectively as strong equivalence class testing. For example, strong testing ensures more thorough coverage of input combinations.
2. **Strongly Typed Languages:** In strongly typed languages where invalid values lead to runtime errors, using robust forms of equivalence class testing may not be necessary. For instance, if the language automatically detects invalid inputs, robust testing may not provide additional benefits.
3. **Prioritizing Error Conditions:** When error conditions are critical, robust forms of equivalence class testing are suitable. For instance, if detecting and handling errors is a top priority, robust testing can help identify and address such scenarios.
4. **Input Data Characteristics:** Equivalence class testing is ideal for scenarios where input data is defined by intervals or discrete values. This is particularly relevant when system malfunctions can occur due to out-of-range input values. For example, testing a system that crashes when receiving negative values.
5. **Combining Approaches:** Strengthen equivalence class testing by combining it with boundary value testing. By integrating boundary values into equivalence classes, testing coverage can be enhanced. For instance, testing a function that calculates discounts based on different price ranges.
6. **Complex Functions:** Equivalence class testing is recommended for complex program functions. The complexity of a function can help identify relevant equivalence classes. For example, testing a function like NextDate that involves multiple conditions based on input dates.
7. **Independence of Variables:** Strong equivalence class testing assumes variable independence, potentially leading to redundant test cases. Dependencies between variables can result in error scenarios. For instance, testing a function that calculates loan interest rates based on both principal amount and duration.
8. **Discovering Equivalence Relations:** It may take multiple attempts to identify the correct equivalence relation for testing. Sometimes, the relation is obvious, while in other cases, it requires careful consideration of implementation details. For example, testing a function that sorts numbers based on different criteria.
9. **Testing Levels and Progression:** Understanding the difference between strong and weak equivalence class testing helps distinguish between progression (moving forward) and regression (ensuring previous functionality still works) testing. For instance, testing new features with strong equivalence class testing while ensuring existing features work with weak equivalence class testing.

3.8 Advantages and Disadvantages of Equivalence Class Testing

Advantages of Equivalence Class Testing

1. **Efficiency:** ECT reduces the number of test cases needed to cover various scenarios, optimizing testing efforts and resources.
2. **Coverage:** By focusing on representative values within equivalence classes, ECT ensures adequate coverage of different input conditions.
3. **Error Detection:** ECT helps identify errors, especially at boundaries and with invalid inputs, improving the overall quality of the software.
4. **Simplicity:** ECT simplifies the testing process by categorizing inputs into manageable equivalence classes, making it easier to design test cases.
5. **Time-Saving:** ECT saves time by prioritizing testing on critical input ranges and values, leading to quicker identification of defects.

Disadvantages of Equivalence Class Testing

1. **Dependency Assumption:** Strong ECT assumes independence between variables, which may not always hold true in complex systems, leading to potential oversight of interdependencies.
2. **Boundary Issues:** ECT may overlook specific boundary conditions that fall outside defined equivalence classes, potentially missing critical test scenarios.
3. **Limited Scope:** ECT may not cover all possible combinations of inputs, especially in systems with intricate interactions between variables.
4. **Subjectivity:** Defining equivalence classes can be subjective, and different testers may categorize inputs differently, leading to variations in test coverage.
5. **Overlooking Edge Cases:** ECT may not always capture extreme or outlier values that could trigger unique system behaviors, potentially leaving vulnerabilities untested.

3.9 Review Questions

Section A

Each Question Carries Two Marks

1. What do you mean by Equivalence Classes ?
2. What is Equivalence Class Testing (ECT)? Give an example.
3. Mention the Forms of Equivalence Class Testing.
4. What is Weak Normal Equivalence Class Testing (WNECT) ?
5. Mention any two Benefits of Weak Normal Equivalence Class Testing (WNECT).
6. What is Strong Normal Equivalence Class Testing (SNECT) ?
7. Mention any two Benefits of Strong Normal Equivalence Class Testing (WNECT).
8. What is Weak Robust Equivalence Class Testing (WRECT) ?
9. Mention any two Benefits of Weak Robust Equivalence Class Testing (WRECT).

10. What is Strong Robust Equivalence Class Testing (SRECT) ?
11. Mention any two Benefits of Strong Robust Equivalence Class Testing (SRECT).

Section - B

Each Question Carries Five Marks

1. What is Equivalence Class Testing (ECT)? Explain its Importance.
2. What is Weak Normal Equivalence Class Testing (WNECT) ? Write the Key Characteristics of WNECT.
3. Write the Benefits and Limitations of Weak Normal Equivalence Class Testing (WNECT).
4. What is Strong Normal Equivalence Class Testing (SNECT) ? Write the Key Characteristics of SNECT.
5. Write the Benefits and Limitations of Strong Normal Equivalence Class Testing (SNECT).
6. What is Weak Robust Equivalence Class Testing (WRECT) ? Write the Key Characteristics of WRECT.
7. Write the Benefits and Limitations of Weak Robust Equivalence Class Testing (WRECT).
8. What is Strong Robust Equivalence Class Testing (SRECT) ? Write the Key Characteristics of SRECT.
9. Differentiate Between Weak Normal Vs Strong Normal Equivalence Class Testing.
10. Differentiate Between Weak Robust Vs Strong Robust Equivalence Class Testing.
11. Explain Equivalence Class Test Cases for the Triangle Problem.
12. Explain Equivalence Class Test Cases for the NextDate Function.
13. Explain Equivalence Class Test Cases for the Commission Problem.
14. Write the Advantages and Disadvantages of Equivalence Class Testing.

Section - C

Each Question Carries Eight Marks

1. Explain the Forms or Variations of Equivalence Class Testing with examples.
2. Explain Weak Normal Equivalence Class Testing (WNECT). Write its Characteristics, Benefits and Limitations.
3. Explain Strong Normal Equivalence Class Testing (SNECT). Write its Characteristics, Benefits and Limitations.
4. Explain Weak Robust Equivalence Class Testing (WRECT). Write its Characteristics, Benefits and Limitations.
5. Explain Strong Robust Equivalence Class Testing (SRECT). Write its Characteristics, Benefits and Limitations.
6. Explain the Guidelines and Observations About Equivalence Class Testing.

❖❖❖❖❖

UNIT - II

CHAPTER 4

DECISION TABLE- BASED TESTING

Contents

4.1 Introduction

Functional testing methods based on decision tables are considered highly rigorous due to their robust logical foundation. Decision tables provide a structured way to represent complex logic and conditions in a clear and organized manner. By mapping inputs to outputs based on different combinations of conditions, decision tables help testers systematically design test cases that cover various scenarios. This method ensures thorough test coverage and helps identify potential defects or inconsistencies in the software application.

4.2 Decision Tables

Decision tables are a structured way to represent complex logical relationships and decision-making processes. They are particularly useful in scenarios where various actions are based on different combinations of conditions. They have been utilized since the early 1960s and are particularly useful for describing scenarios where multiple actions are taken based on different sets of conditions.

By using decision tables, testers and developers can effectively analyze complex decision-making processes, design test cases based on different scenarios, and verify the behavior of a system under various conditions. The tabular format of decision tables simplifies the understanding of logic and facilitates the creation of test cases that cover all possible paths through the system based on different input conditions.

? What is Decision Table?

A decision table is a systematic and structured method used in software testing and system analysis to represent complex logical relationships between conditions and actions. It provides a visual and tabular representation of different combinations of input conditions and corresponding outcomes or actions. Decision tables help in designing test cases, analyzing decision-making logic, and ensuring comprehensive coverage of various scenarios within a software system.

? What is Decision Table Based Testing?

Decision table based testing is a systematic testing technique that uses decision tables to design and execute test cases for software systems. This approach involves creating decision tables that capture various combinations of input conditions and corresponding actions or outcomes. Testers analyze these decision tables to derive test cases that cover different scenarios based on the defined conditions.

4.2.1 Characteristics of Decision Tables

Decision tables are a powerful tool for organizing and managing complex decision-making processes in software development and other fields that require logical analysis. Some key characteristics of decision tables are:

1. **Conditions:** Decision tables include conditions that represent the input variables or factors that need to be evaluated. These conditions define the criteria that influence the decision-making process within the system.

2. **Actions :** Actions in decision tables specify the outcomes or responses that result from the evaluation of specific conditions. Each combination of conditions leads to a corresponding action or set of actions.
3. **Rules :** Rules in decision tables represent individual rows that capture a unique combination of conditions and the associated actions. Each rule defines a specific scenario or decision logic within the system.
4. **Structured Format :** Decision tables are organized into distinct sections, such as the stub portion, entry portion, condition portion, and action portion. This structured format provides a clear framework for organizing and analyzing the decision-making logic.
5. **Comprehensive Coverage :** Decision tables aim to ensure comprehensive coverage of various scenarios by considering all possible combinations of conditions and actions. This helps in identifying potential gaps in the decision logic and designing test cases that cover all possible paths through the system.
6. **Visual Representation:** Decision tables offer a visual representation of complex decision logic, making it easier for testers and developers to understand the relationships between conditions and actions. The tabular format simplifies the analysis of decision-making processes.
7. **Scalability :** Decision tables can scale to accommodate a large number of conditions and actions, making them suitable for analyzing complex systems with multiple decision points. They can handle increasing complexity while maintaining clarity and organization.
8. **Flexibility :** Decision tables provide flexibility in modifying and updating the decision logic by adjusting conditions, actions, or rules as needed. This adaptability allows for easy maintenance and refinement of the decision-making process over time.

4.2.2 Importance (or) Benefits of Decision Tables

Decision tables play a crucial role in software testing by enhancing clarity, improving test coverage, identifying dependencies, validating logic, mitigating risks, facilitating communication, and ensuring scalability and flexibility in decision-making processes. Their importance lies in their ability to streamline testing efforts, enhance understanding, and promote effective decision-making within software systems.

1. **Clarity and Understanding:** Decision tables provide a clear and structured representation of complex decision logic, making it easier for stakeholders, including testers, developers, and business analysts, to understand the relationships between conditions and actions within a system.
2. **Comprehensive Coverage:** By systematically capturing all possible combinations of conditions and corresponding actions, decision tables help ensure comprehensive test coverage. This reduces the risk of overlooking critical scenarios and improves the overall quality of testing.
3. **Effective Test Case Design:** Decision tables serve as a valuable tool for designing test cases that cover various scenarios based on different combinations of conditions. Testers can use decision tables to create targeted test cases that validate the behavior of the system under specific conditions.

4. **Identification of Dependencies:** Decision tables highlight dependencies between input conditions and output actions, enabling testers to identify complex relationships and potential dependencies that may impact the system's behavior. This insight is crucial for thorough testing.
5. **Logic Validation:** Decision tables facilitate the validation of decision-making logic within a system. By analyzing the rules and conditions in the table, testers can verify that the system behaves as expected under different conditions and that the logic is correctly implemented.
6. **Risk Mitigation:** Using decision tables helps mitigate the risk of overlooking critical decision paths or scenarios during testing. By systematically documenting all possible conditions and actions, testers can address potential risks and ensure that the system functions correctly in diverse scenarios.
7. **Efficient Communication:** Decision tables provide a standardized format for communicating decision logic across teams and stakeholders. They enable effective collaboration by presenting complex information in a structured and easily understandable manner.
8. **Scalability and Flexibility:** Decision tables can scale to accommodate increasing complexity in decision-making processes. They offer flexibility in modifying conditions, actions, and rules, allowing for easy adaptation to changing requirements and evolving system behavior.

4.2.3 Decision Table Design Format (or) Components of Decision Table

Decision tables follow a structured format that includes specific components to represent the conditions, actions, and rules governing the decision-making process within a system. The design format typically consists of the following sections:

1. **Condition Stub:**
   - **Purpose :** Lists all the conditions relevant to the decision process. Conditions are typically questions or statements that can be answered with Yes (Y), No (N), or Don't Care (-).
   - **Location:** Left side of the table.
2. **Condition Entries:**
   - **Purpose :** Contains entries (Y, N, -) corresponding to each condition for every rule defined in the decision table.
   - **'Don't Care' Entries:** Sometimes a condition in a particular rule might not affect the outcome, indicated by a '-' entry. This means that the action does not depend on this condition's state.
   - **Location:** Right of the condition stub, divided by a vertical line.
3. **Action Stub:**
   - **Purpose :** Lists the possible actions that can be taken based on the evaluation of the conditions.
   - **Location:** Below the condition stub.
4. **Action Entries:**
   - **Purpose:** Indicates which actions are to be taken for each rule by marking with an X or leaving blank based on the condition entries above.
   - **Location:** Below the condition entries, divided by a horizontal line.

5. **Rules:**
   - **Purpose:** Represents a unique combination of condition entries and specifies the resulting actions. Each rule is a feasible scenario that might occur, and the table explores the outcomes.
   - **Location:** Each column in the entry portion of the table.

Example **Decistion Table Design Format**

Let us understand the condition stub, the condition entries, the action stub, and the action entries from the below table. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. In the decision table in Table 4.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true and c3 is false, then actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a "don't care" entry. The don't care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the "n/a" symbol for this latter interpretation.

Stub	Rule 1	Rule 2	Rules 3, 4	Rule 5	Rule 6	Rules 7, 8
c1	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	-	T	F	-
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

When we have binary conditions (true/false, yes/no, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing.

Decision tables in which all the conditions are binary are called **Limited Entry Decision Tables** (LETDs). If conditions are allowed to have several values, the resulting tables are called **Extended Entry Decision Tables** (EEDTs).

Example 1 **Decistion Table**

Imagine a software system that manages subscriptions for an online platform. The system must decide whether to activate a subscription based on user age verification and payment status.

Sample Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3	Rule 4
User is over 18	Y	Y	N	N
Payment received	Y	N	Y	N
Action Stub				
Activate Subscription	X			
Send Reminder		X		
Reject Application			X	X

Detailed Explanation:

1. **Rule 1 (Y, Y):**
   - **Conditions:** User is over 18 (Y) and payment received (Y).
   - **Action:** Activate Subscription.
   - **Scenario:** The user meets all criteria for subscription activation.
2. **Rule 2 (Y, N):**
   - **Conditions:** User is over 18 (Y) but payment not received (N).
   - **Action:** Send Reminder.
   - **Scenario:** The user is eligible but needs to complete payment; hence a reminder is sent.
3. **Rule 3 (N, Y):**
   - **Conditions:** User is not over 18 (N) but payment received (Y).
   - **Action:** Reject Application.
   - **Scenario:** Even though the payment was received, the user does not meet the age requirement, leading to rejection.
4. **Rule 4 (N, N):**
   - **Conditions:** User is not over 18 (N) and payment not received (N).
   - **Action:** Reject Application.
   - **Scenario:** Neither condition is met; the application is outright rejected.

Example 2 **Decistion Table**

Consider an user registration based on valid age and valid email:

Sample Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3	Rule 4
Age >= 18	Y	Y	N	N
Email Valid	Y	N	Y	N
Action Stub				
Register User	X			
Send Error Message		X	X	X

Explanation :

1. **Condition Stub:** Lists conditions like 'Age >= 18' and 'Email Valid'. These are the factors that affect decision-making in the software logic. Each condition must be evaluated to determine which actions to take.
2. **Action Stub:** Lists possible actions such as 'Register User' or 'Send Error Message'. Actions are dependent on the outcomes of the evaluated conditions.
3. **Condition Entries:** Situated to the right of the Condition Stub, this section specifies the state (Yes or No - Y/N) of each condition for each rule.
   - **Rule 1:** Both age and email conditions are met (Y, Y).
   - **Rule 2:** Age is adequate but email is invalid (Y, N).

   - **Rule 3:** Age is inadequate regardless of email validity (N, Y).
   - **Rule 4:** Bothe age and email are not met (N, N).
4. **Action Entries:** Located directly below the Action Stub, showing actions triggered based on the conditions listed directly above in the condition entries.
   - **Rule 1** results in registering the user since both conditions are satisfactory.
   - **Rule 2, Rule 3 and Rule 3** result in an error message being sent due to one or both conditions failing.
5. **Rules:**
   - Each column under the entries represents a rule, a specific scenario defined by a unique combination of condition states.
   - A rule dictates which actions occur for a given set of conditions.

4.3 Decision Table Techniques

Decision tables are a powerful tool for designing test cases especially when handling complex logical conditions and their associated actions.Let us discuss the main techniques used in decision tables.

1. Completeness and Redundancy :

Completeness in decision tables means that the table must account for every possible combination of input conditions. Redundancy refers to the presence of duplicate rules within the decision table that lead to the same actions based on the same or similar conditions.

Benefits of Ensuring Completeness and Minimizing Redundancy :

1. **Ensures Comprehensive Coverage:** Completeness in decision tables ensures that every possible input combination is considered, which significantly reduces the risk of defects slipping through undetected. This thorough coverage is essential for critical systems where failures can have severe implications.
2. **Identifies All Possible Behaviors:** By accounting for all potential conditions, completeness helps testers understand every possible behavior of the system. This understanding is crucial for verifying that the system operates correctly across all scenarios, which is especially important in complex systems with many interdependent components.
3. **Reduces the Risk of Unexpected Issues:** Complete decision tables help in detecting and addressing edge cases and rare scenarios that might not be immediately apparent. This proactive identification helps prevent unexpected issues post-deployment, thereby enhancing the reliability and stability of the system.
4. **Streamlines Testing and Maintenance:** Minimizing redundancy in decision tables makes the testing process more streamlined and resource-efficient. It eliminates unnecessary testing efforts, saving time and reducing the potential for confusion or error that can arise from handling duplicate test cases.

Example

Let's consider a decision table for an online booking system where users can book rooms depending on their membership status and room availability.

1. **Conditions:**
   - Membership Status: Valid or Invalid
   - Room Availability: Available or Not Available
2. **Actions:**
   - Book Room: Proceed with the booking.
   - Show Error: Display an error message regarding availability or membership status.

Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3	Rule 4
Membership Status (Valid)	Y	Y	N	N
Room Availability (Available)	Y	N	Y	N
Action Stub				
Book Room	X			
Show Error		X	X	X

Explanation of the Table:

- **Rule 1:** When a user has a valid membership and the room is available, the action is to book the room. This rule is necessary and unique.
- **Rule 2 :** For a valid member when the room is not available, the system shows an error about room availability.
- **Rule 3 :** If the membership is invalid but the room is available, it displays an error about membership status.
- **Rule 4 :** Shows an error when both membership is invalid and the room is not available. This covers the scenario where multiple issues prevent booking.

By ensuring every combination of conditions is addressed (completeness) and avoiding duplication of the same outputs for the same inputs (redundancy), the decision table helps in thorough testing of the booking system. This method ensures that the software's reaction to every possible user interaction is tested and verified, which enhances reliability and user satisfaction.

2. Don't Care Entries :

Don't care entries in decision tables are used when the state of a particular condition does not affect the outcome for specific rules. This approach simplifies the table by focusing only on the conditions that influence the decision, reducing the complexity and size of the table.

Advantages of Using Don't Care Entries:

1. **Simplification:** Reduces the number of rules in the decision table by not considering irrelevant conditions, making the table easier to read and understand.
2. **Focus on Relevant Conditions:** Allows stakeholders to focus only on the conditions that impact outcomes, which can speed up decision-making processes.
3. **Efficiency:** Minimizes the effort needed to test scenarios that are unaffected by certain conditions, enhancing the efficiency of the testing process.

Example

Consider a monitoring system that checks for errors and system load. The system needs to decide whether to send an alert or continue monitoring based on these factors.

1. **Conditions:**
   - **System Load is High:** This condition can impact how the system prioritizes resources but might not directly influence whether an alert should be sent or not.
   - **Error Reported:** This condition directly determines whether an alert needs to be sent.
2. **Actions:**
   - **Send Alert:** Triggered when an error is reported, regardless of the system load.
   - **Continue Monitoring:** The default action when no errors are reported.

Decision Table:

Condition Stub	Rule 1	Rule 2
System Load High	Don't Care	Don't Care
Error Reported	Yes	N0
Action Stub		
Send Alert	X	
Send Alert		X

Explanation of the Table:

- **Rule 1:** The action to send an alert is triggered if an error is reported, irrespective of whether the system load is high or not. Here, the 'Don't care' entry for the system load indicates that the presence or absence of high system load does not impact the decision to send an alert.
- **Rule 2:** Continue monitoring is the action when no error is reported. Again, the system load does not influence this action, hence the 'Don't care' designation.

In this example, using 'Don't care' entries effectively ignores the system load when deciding to send alerts or continue monitoring, as the action depends solely on whether an error is reported. This technique is particularly useful in systems where some inputs are significantly more critical than others in determining the output.

3. Impossible Rules :

Impossible rules in decision tables happen when some conditions just don't make sense together, so the actions linked to them wouldn't apply. By spotting and marking these impossible situations, we prevent confusion and make sure the decision table shows the system's rules correctly.

Advantages of Identifying Impossible Rules:

1. **Prevent Logical Errors:** Helps prevent situations where the system might attempt to execute actions under conditions that logically cannot happen.
2. **Clarity in Test Case Design:** Clear identification of impossible scenarios helps in designing more effective test cases and avoids wasting resources on testing unrealistic conditions.
3. **Enhanced Understanding:** Provides all stakeholders with a better understanding of the system's constraints and logical flows, facilitating better system design and troubleshooting.

Example

A payment processing system needs to decide whether to complete or decline a transaction based on the validity of a credit card and whether the payment has been processed.

1. **Conditions:**
   - **Credit Card Valid:** Determines if the credit card being used is valid.
   - **Payment Processed:** Indicates whether the payment transaction has successfully been processed.
2. **Actions:**
   - **Complete Transaction:** Execute this action if the transaction can be successfully completed.
   - **Decline Transaction:** Execute this action if the transaction must be declined due to various issues.

Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3
Credit Card Valid	Yes	No	Yes
Payment Processed	Yes	Yes	No
Action Stub			
Complete Transaction	X		
Decline Transaction		Impossible (X)	X

Explanation of the Table:

- **Rule 1:** This rule represents a scenario where both the credit card is valid and the payment has been processed. The logical action is to complete the transaction, so the "Complete Transaction" action is marked.
- **Rule 2:** Here, the credit card is not valid, but the condition states that the payment has been processed. This is an impossible scenario because a payment cannot be processed using an invalid credit card. Thus, this rule is marked as impossible, and the intended action (decline the transaction) is indicated but should not be executed under this rule because the scenario cannot occur.
- **Rule 3:** The credit card is valid, but the payment has not been processed successfully. The logical action is to decline the transaction.

4. Rule Count Adjustment :

Rule Count Adjustment is a technique used in decision tables to ensure that the rule counts accurately reflect the number of unique and meaningful test scenarios, especially when "don't care" entries are used. "Don't care" entries indicate that the outcome does not depend on the state of a particular condition, allowing for multiple possible states without affecting the rule's actions.

Advantages of Rule Count Adjustment:

1. **Accuracy in Testing:** Adjusting the rule count ensures that the number of test cases reflects all possible scenarios that might need to be tested, enhancing the thoroughness and reliability of testing.
2. **Effective Resource Allocation:** By understanding the actual number of scenarios each rule represents, resources can be better allocated to test all relevant cases effectively.
3. **Clarity in Specification:** This adjustment clarifies how "don't care" entries impact the overall decision-making process, helping to prevent misunderstandings and ensuring that all potential scenarios are considered.

Example

A system controls access based on whether a login is required and whether a form is filled out correctly.

1. **Conditions:**
   - **Login Required:** Specifies whether the user needs to log in.
   - **Form Filled:** Indicates whether the user has filled out a form correctly.
2. **Actions:**
   - **Allow Access:** The user is allowed access to the system.
   - **Deny Access:** The user is denied access.

Decision Table:

Condition Stub	Rule 1	Rule 2
Login Required	Don't care	Yes
Form Filled	Yes	No
Action Stub		
Allow Access	X	
Deny Access		X

Explanation of the Table:

- **Rule 1:** The condition for "Login Required" is marked as "don't care", which means the action "Allow Access" can occur regardless of whether login is required or not, as long as the form is filled correctly ("Form Filled" = Yes).
- **Rule 2:** Access is denied ("Deny Access") when login is required and the form is not filled correctly.

Rule Count Adjustment:

- **Without Adjustment:** Normally, each rule in a decision table corresponds to a specific scenario. Here, because "Login Required" in Rule 1 is a "don't care", this single rule actually represents two scenarios: one where login is required and the form is filled, and another where login is not required and the form is filled.
- **With Adjustment:** For Rule 1, each "don't care" state effectively doubles the count of scenarios it represents. Thus, Rule 1 actually covers two scenarios:
  - Login Required = Yes, Form Filled = Yes
  - Login Required = No, Form Filled = Yes

The Rule Count Adjustment technique is crucial for ensuring that decision tables not only represent but also accurately count all unique conditions covered by rules, particularly when "don't care" entries expand the implications of a rule beyond a single scenario.

5. Using Equivalence Classes :

Using Equivalence Classes is a technique in decision table design that involves defining conditions based on distinct groups or classes of inputs that are expected to behave similarly. This method helps simplify the testing process by reducing the number of individual cases that need to be tested, assuming that testing one sample from each class is representative of the entire class.

Benefits of Using Equivalence Classes:

1. **Simplification of Test Cases:** By treating all instances within an equivalence class the same way, fewer tests need to be designed and executed, focusing resources on varied scenarios.
2. **Comprehensive Coverage:** Ensures that each unique combination of input classes is considered, providing thorough coverage without the redundancy of testing each possible input value.
3. **Efficiency in Test Execution:** Reduces the number of tests required while maintaining an effective assessment of system behavior across different user categories and ticket types.

Example

A ticketing system issues tickets based on the age group of the customer and the type of ticket they choose (Standard or Premium).

1. **Conditions:**
   - **Age Group:** Defines whether the customer is an Adult, Child, or Senior.
   - **Ticket Type:** Specifies whether the ticket is Standard or Premium.
2. **Actions:**
   - **Issue Ticket:** A ticket is issued to the customer.
   - **Offer Discount:** A discount is offered, typically associated with specific conditions like age group or ticket type.

Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3	Rule 4
Age Group	Adult	Child	Senior	Adult
Ticket Type	Standard	Standard	Premium	Premium
Action Stub				
Issue Ticket	X	X	X	X
Offer Discount			X	

Explanation of the Table:

- **Rule 1:** Adults buying standard tickets are simply issued a ticket.
- **Rule 2:** Children with standard tickets receive the same action as adults with standard tickets due to the simplicity of the transaction.
- **Rule 3:** Seniors purchasing premium tickets get both an issued ticket and a discount, acknowledging the combination of a higher-priced ticket and a potentially discounted group.
- **Rule 4:** Adults purchasing premium tickets are issued a ticket but do not receive a discount, distinguishing this scenario from seniors due to the absence of age-related incentives.

This technique is particularly useful in systems where the input space is large and can be logically divided into categories or classes that are expected to elicit the same response from the system. By applying this approach, decision tables remain manageable while still offering a structured and systematic way to capture and test complex business rules and their outcomes.

6. Mutually Exclusive Conditions :

Design decision tables need to clearly separate conditions that cannot occur simultaneously to ensure that each scenario is distinct and unambiguous. Mutually exclusive conditions in decision tables help simplify the decision-making process by ensuring that conditions in a given rule cannot overlap. This clarity prevents the creation of impossible or contradictory rules and helps maintain the integrity of logical evaluations.

Benefits of Mutually Exclusive Conditions

1. **Simplification of Decision Logic:** Mutually exclusive conditions ensure that each condition or scenario is distinct and does not overlap with others. This simplification helps to reduce complexity in understanding and analyzing the decision logic, making it more straightforward to implement and test.
2. **Prevention of Rule Overlap:** Since mutually exclusive conditions do not overlap, they prevent multiple rules from being triggered simultaneously for a single set of inputs. This clear delineation helps avoid conflicts or contradictions in decision outcomes, ensuring consistent and predictable actions.
3. **Efficient Testing:** Testing becomes more manageable and less time-consuming because each condition set leads to a unique path or outcome. This efficiency aids in targeted testing strategies and reduces the potential for errors during test case execution, as the conditions define clear boundaries for each test scenario.
4. **Enhanced Clarity and Communication:** Decision tables with mutually exclusive conditions are easier to read and understand. They provide a clear visual representation of how different scenarios are handled, which can be beneficial for communication among team members and stakeholders, ensuring everyone understands the logic and expected behaviors.

Example

Let us consider a simple banking transactions.

1. **Conditions:**
   - **Account Type:** Checking, Savings (Cannot be both)
   - **Transaction Type:** Deposit, Withdrawal (Cannot be both)
2. **Actions:**
   - **Process Transaction**
   - **Reject Transaction**

Decision Table:

Condition Stub	Rule 1	Rule 2	Rule 3	Rule 4
Account Type - Checking	Yes	No	Yes	No
Account Type - Savings	No	Yes	No	Yes
Transaction Type - Deposit	Yes	Yes	No	No
Transaction Type - Withdrawal	No	No	Yes	Yes

Action Stub				
Process Transaction	X	X	X	X
Reject Transaction				

Explanation of the Table:

- **Rule 1:** Transaction is a deposit in a checking account. Process the transaction because conditions are valid and mutually exclusive.
- **Rule 2:** Transaction is a deposit in a savings account. Process the transaction as it meets the mutually exclusive conditions.
- **Rule 3:** Transaction is a withdrawal from a checking account. Again, process it due to valid and exclusive conditions.
- **Rule 4:** Transaction is a withdrawal from a savings account. Process the transaction as all conditions align without conflict.

This table ensures that each combination of account type and transaction type is uniquely handled, avoiding overlaps such as attempting to process a deposit and withdrawal simultaneously. This clarity enhances understanding and reduces errors in the system's operational logic.

4.4 Test Cases for the Triangle Problem

The decision table for the triangle problem can be constructed by breaking down the conditions into more specific tests that consider all inequalities and equalities among the triangle's sides. The decision table method ensures that all logical scenarios are covered.

Conditions:

- c1: a < b + c? Checks if side a is less than the sum of sides b and c, which is a requirement for forming a triangle.
- c2: b < a + c? Similar check for side b.
- c3: c < a + b? Similar check for side c.
- c4: a = b? Checks if sides a and b are equal.
- c5: a = c? Checks if sides a and c are equal.
- c6: b = c? Checks if sides b and c are equal.

Actions:

- a1: Not a triangle - Action if the sides do not satisfy the triangle inequality theorem.
- a2: Scalene - Action if no sides are equal and the sides form a triangle.
- a3: Isosceles - Action if exactly two sides are equal and the sides form a triangle.
- a4: Equilateral - Action if all three sides are equal and the sides form a triangle.
- a5: Impossible - Marks configurations that are logically contradictory or impossible.

Decision Table:

	1	2	3	4	5	6	7	8	9	10	11
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c>	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	T	T	T	T	T	T	T
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isoceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Analysis:

- **Columns (Rules):** Each column after the conditions represents a rule which is a specific scenario that could occur given the set of conditions. The way the table is set up guarantees that every possible logical scenario is considered.
- **Don't Care Entries ("-"):** These are used where the outcome of a condition does not affect the outcome of the test because other conditions already determine the result. For example, if c1 is false (F), it doesn't matter what the values of c2, c3, c4, c5, or c6 are, the triangle cannot be formed.
- **Rule 1 (F, -, -, -, -, -):** Indicates that if c1 is false, the sides cannot form a triangle, triggering action a1: Not a triangle.
- **Rule 10 (T, T, T, F, F, T):** This rule represents a scenario where all inequalities are satisfied (T for c1, c2, c3), but a = b is false (F for c4), a = c is false (F for c5), and b = c is true (T for c6). This means the triangle is isosceles (a3).
- **Rule 12 (T, T, T, T, T, T):** All conditions are true, indicating an equilateral triangle (a4).
- **Rules with "Impossible" (a5):** These rules indicate scenarios where the set conditions logically contradict the properties of a triangle, such as having one side equal to the sum of the others, which would not form a triangle.

Test Cases Derived from Decision Table :

Case ID	a	b	c	Expected Output
Rule 1: DT1	4	1	2	Nor a triangle
Rule 2: DT2	1	4	2	Nor a triangle
Rule 3: DT3	1	2	4	Nor a triangle
Rule 4: DT4	5	5	5	Equilateral

Rule 5: DT5	?	?	?	Impossible
Rule 6: DT6	?	?	?	Impossible
Rule 7: DT7	2	2	3	Isosceles
Rule 8: DT8	?	?	?	Impossible
Rule 9: DT9	2	3	2	Isosceles
Rule 10: DT10	3	2	2	Isosceles
Rule 11: DT11	3	4	5	Scalene

4.5 Test Cases for the Next Date Function

The NextDate function is designed to calculate the date of the following day given a specific input date consisting of day, month, and year. This function addresses various complexities associated with the Gregorian calendar, including different month lengths, leap years, and transitions from one month or year to the next.

Conditions

- **Month:** Specifies the current month. This is crucial as months have varying numbers of days.
- **Day:** Specifies the current day within the month.
- **Year:** Determines if the year is a leap year or a non-leap year based on the rules of the Gregorian calendar.

Actions

- **Next Day:** The day number for the following day.
- **Next Month:** The month number for the following day.
- **Next Year:** The year for the following day.

Decision Table for Valid Scenarios:

Condition \ Rule	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Month	Jan-Dec (not last day)	Jan-Nov (last day)	Dec (last day)	Feb (28, non-leap)	Feb (28, leap)	Feb (29, leap)	30-Day Months (Apr, Jun, Sep, Nov)	31-Day Months (Jan, Mar, May, July, Aug, Oct)
Day	Not Last Day	Last Day	Last Day	28	28	29	30	31
Year	Any	Any	Any	Non-Leap Yea	Leap Yea	Leap Year	Any	Any
Actions								
Next Day	+1 to current day	01	01	01	29	01	01	01

Next Month	Same	+1 to current month	Jan	Mar	Same	Mar	+1 to current month	+1 to current month
Next Year	Same	Same	+1 to current year (next year)	Same	Same	Same	Same	Same

Analysis:

- **Rule 1:** For any regular day that is not the last of the month, simply increment the day (e.g., March 14th to March 15th).
- **Rule 2:** For the last day of any month except December, the date changes to the first day of the next month (e.g., January 31st to February 1st).
- **Rule 3:** For December 31st, the date changes to January 1st of the next year (e.g., December 31st, 2021 to January 1st, 2022).
- **Rule 4:** For February 28th in a non-leap year, the next date is March 1st (e.g., February 28th, 2021 to March 1st, 2021).
- **Rule 5:** For February 28th in a leap year, the next date is February 29th (e.g., February 28th, 2020 to February 29th, 2020).
- **Rule 6:** For February 29th in a leap year, the next date is March 1st (e.g., February 29th, 2020 to March 1st, 2020).
- **Rule 7:** For the last day of months with 30 days, the date changes to the first day of the next month (e.g., April 30th to May 1st).
- **Rule 8:** For the last day of months with 31 days, the date changes to the first of the next month except for December (e.g., July 31st to August 1st).

This decision table effectively maps out the transitions between different dates, ensuring that each scenario is specifically addressed, thereby facilitating comprehensive testing and validation of the NextDate function.

Test Cases Derived from Decision Table :

Case ID	Day	Month	Year	Expected Output
Rule 1: DT1	15	6	2022	16/6/2022
Rule 2: DT2	30	4	2022	1/5/2022
Rule 3: DT3	31	12	2022	1/1/2023
Rule 4: DT4	28	2	2023	1/3/2023
Rule 5: DT5	28	2	2024	29/2/2024
Rule 6: DT6	29	2	2024	1/3/2024
Rule 7: DT7	30	4	2024	1/5/2024
Rule 8: DT8	31	5	2024	1/6/2024

4.6 Test Cases for the Commission Problem

The Commission problem revolves around calculating the commission for a salesperson based on the quantity of locks, stocks, and barrels sold within a month. Each product has a fixed selling price and a commission rate that varies based on the total sales amount.

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.

Conditions for Decision Table:

- **Locks Sold:** Number of locks sold within their allowed range.
- **Stocks Sold:** Number of stocks sold within their allowed range.
- **Barrels Sold:** Number of barrels sold within their allowed range.
- **Total Sales Bracket:** Defined brackets based on the total sales amount for calculating commissions.

Actions Defined for Decision Table:

- **Calculate Commission:** The specific formula used based on total sales.
- **Validate Sales:** Indicates whether the sales are within the permissible range or not.

Decision Table:

Condition \ Rule	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
Locks Sold	1-70	1-70	1-70	>70	Any	Any
Stocks Sold	1-80	1-80	1-80	Any	>80	Any
Barrels Sold	1-90	1-90	1-90	Any	Any	>90
Total Sales	≤ $1000	$1001-$1800	> $1800	N/A	N/A	N/A
Actions						
Calculate Comm.	10% of sales	$100 + 15% over $1000	$220 + 20% over $1800	0	0	0
Validate Sales	Valid	Valid	Valid	Invalid	Invalid	Invalid

Analysis:

- **Rule 1:** Applies when all items are within limits and total sales are ≤ $1000. Commission is 10% of sales.
- **Rule 2:** Applies when all items are within limits and total sales are between $1001 and $1800. Commission includes a base of $100 plus 15% of the amount over $1000.
- **Rule 3:** Applies when all items are within limits and total sales exceed $1800. Commission includes a base of $220 plus 20% of the amount over $1800.
- **Rule 4, 5,6:** Handle invalid sales scenarios where the number of locks, stocks, or barrels exceeds their respective maximum limits. In these cases, sales are marked invalid and no commission is awarded.

Test Cases Derived from Decision Table :

Case ID	Locks	Stocks	Barrels	Total Sales	Commission
Rule 1: DT1	5	5	5	$500	$50
Rule 2: DT2	15	15	15	$1500	$175
Rule 3: DT3	20	25	30	$2400	$340
Rule 4: DT4	80	40	40	Invalid	$0
Rule 5: DT5	30	85	30	Invalid	$0
Rule 6: DT6	30	40	95	Invalid	$0

1. **Rule C1 (DT1) - Valid under $1000 range:**
   Locks = 5, Stocks = 5, Barrels = 5
   Total Sales = (5×45)+(5×30)+(5×25)=225+150+125=$500
   Commission = $500 * 0.10 = $50
2. **Rule C2 (DT2) - Valid within the $1001 to $1800 range:**
   Locks = 15, Stocks = 15, Barrels = 15
   Total Sales = (15×45)+(15×30)+(15×25)=675+450+375=$1500
   Commission = $100 + ($1500 - $1000) * 0.15 = $100 + $75 = $175
3. **Rule C3 (DT3) - Valid above $1800:**
   Locks = 20, Stocks = 25, Barrels = 30
   Total Sales = (20×45)+(25×30)+(30×25)=900+750+750=$2400
   Commission = $220 + ($2400 - $1800) * 0.20 = $220 + $120 = $340
4. **Rule C4 (DT4) - Invalid due to exceeding lock limits:**
   Locks = 80, Stocks = 40, Barrels = 40
   Sales Invalid, Commission = $0
5. **Rule C5 (DT5) - Invalid due to exceeding stock limits:**
   Locks = 30, Stocks = 85, Barrels = 30
   Sales Invalid, Commission = $0
6. **Rule C6 (DT6) - Invalid due to exceeding barrel limits:**
   Locks = 30, Stocks = 40, Barrels = 95
   Sales Invalid, Commission = $0

4.7 Guidelines and Observations of Decision Table Testing

Decision table testing is particularly effective in scenarios where decision-making is complex, involving numerous logical conditions. This testing method is ideal for applications with significant conditional logic, like if-then-else structures, or where inputs and outputs are closely linked through logical operations. Some guidelines and observations of using decision table testing:

1. **When to Use Decision Tables:**
   - **Complex Decision Logic:** Best for scenarios with if-then-else conditions.
   - **Interdependent Inputs:** Useful when inputs are logically connected and affect each other's outcomes.
   - **Calculative Operations:** Ideal for functions that perform calculations on subsets of input data.
   - **Input-Output Correlation:** Effective in environments where inputs directly dictate outputs.

   - **High Complexity:** Suitable for systems with complex pathways indicating many potential paths through the code.
2. **Challenges with Scalability:**
   - Decision tables can become large as the number of conditions increases, with a binary decision table for n conditions resulting in 2^n rules. To manage this complexity:
     a) **Extended Entry Tables:** Use tables that allow multiple values per condition to reduce the number of rules.
     b) **Table Simplification:** Algebraically simplify the rules to make the table more manageable.
     c) **Table Factoring:** Break down large tables into smaller, more manageable pieces.
     d) **Pattern Recognition:** Identify and leverage repeating patterns to streamline the decision-making process.
3. **Iterative Refinement:**
   - The initial set of conditions and actions may not perfectly capture the necessary logic or may be inefficient. Iteratively refining these elements, based on testing results and deeper understanding of the application, helps in developing more effective and streamlined decision tables.

These guidelines help in making decision table testing a practical approach for thoroughly examining complex decision-making processes in software systems to ensure comprehensive coverage and understanding of all potential outcomes.

4.8 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Decision Table?
2. What is Decision Table Based Testing?
3. Write any two benefits of Decision Table Based Testing.
4. What are Don't Care Entries in Decision Table?
5. What are Mutually Exclusive Conditions in Decision Table?
6. What is Rule Count Adjustment?

Section - B

Each Question Carries Five Marks

1. What is Decision Table? Explain the Characteristics of Decision Tables.
2. Explain the Importance (or) Benefits of Decision Tables in Software Testing.
3. Explain Decistion Table Design Format with an example.

5. Explain Rule Count Adjustment in Decision Table Based Testing.
6. Explain Decision Table Based Testing Using Equivalence Classes.
7. Explain the Guidelines and Observations of Decision Table Testing.

Section - C

Each Question Carries Eight Marks

1. What is Decision Table Based Testing? Explain Decision Table Techniques.
2. Explain the Test Cases for the Triangle Problem in Decision Table Based Testing.
3. Explain the Test Cases for the Next Date Function in Decision Table Based Testing.
4. Explain the Test Cases Test Cases for the Commission Problem in Decision Table Based Testing.

❖❖❖❖❖

UNIT - II

CHAPTER 5

DATA FLOW TESTING

Contents

5.1 Data Flow Testing

Data flow testing is a specialized method of structural testing that emphasizes tracking how variables within a program are defined and used. Unlike what the name might suggest, it has no relation to data flow diagrams used in design. Instead, it focuses on understanding and verifying the flow of data through code via the variables' lifecycles.

Overview of Data Flow Testing:

1. **Key Focus:** Data flow testing examines the points in the code where variables are assigned values (defined) and where these values are utilized (used or referenced). It aims to identify issues in how data is handled, like incorrect initialization or using a variable before it's assigned a value.
2. **Purpose:** The main goal is to ensure that the interactions and dependencies involving data within the program are correct and efficient, thus helping to identify potential anomalies in the handling of data.

? What is Data Flow Testing ? or Define Data Flow Testing.

Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced).

Data flow testing is a software testing technique that focuses on examining how data moves through a program. It's a white-box testing method, meaning it relies on the internal structure of the code.

5.1.1 Characteristics of Data Flow Testing

Data flow testing is a detailed method of structural testing aimed at examining how data is handled within software applications. It looks specifically at the lifecycle of variables from their initialization to their final use in computations. Some key characteristics of data flow testing are:

1. **Definition and Use of Variables:** Data flow testing focuses on the points in the code where variables are defined (given a value) and where these values are subsequently used. This can include checking variables in conditions, calculations, or as arguments in function calls.
2. **Detection of Anomalies:** The primary aim is to detect data flow anomalies, which can indicate faults in the program. These include situations where a variable is defined but never used, used before it is defined, or redefined without any subsequent use before another definition.
3. **Program Graphs:** It utilizes program graphs to visually represent the flow of data through the program. These graphs help in tracing the sequence of events that affect data, making it easier to spot potential issues.
4. **Static Analysis:** Data flow testing often involves static analysis, meaning it analyzes the code without actually executing the program. This allows for detecting certain types of errors and inefficiencies in code handling of data statically.
5. **Complexity in Manual Execution:** Due to the detailed nature of tracking each variable's flow through software, data flow testing can be complex and time-consuming, especially without the help of sophisticated tools.

6. **Suitability for Object-Oriented Code:** This type of testing is particularly effective for object-oriented programming, where the interactions between methods and objects involve numerous variable definitions and uses.
7. **Complementary to Other Testing Methods:** While it can be used as a standalone testing approach, data flow testing is often most effective when used in conjunction with other testing strategies like path testing. It provides an additional layer of assurance by focusing on aspects of the code's logic specifically related to data handling.
8. **Coverage Metrics:** Data flow testing includes various coverage metrics to assess the extent of testing. These metrics evaluate how thoroughly the data-related aspects of the program code are tested, ensuring that all potential data interactions are examined.
9. **Tool Dependency:** Effective data flow testing can depend significantly on the availability of tools due to its complexity, especially for larger codebases. The lack of commercial tools may limit its use in some environments.
10. **Enhanced Debugging and Maintenance:** By identifying how data moves and changes within a program, data flow testing helps pinpoint where errors occur, making debugging easier and helping maintain the code more effectively.

5.1.2 Benefits of Data Flow Testing

1. It identifies define/use issues such as unused variables, uninitialized variables, and redundant definitions.
2. It improves code quality by ensuring that all parts of the program contribute to its functionality
3. It makes debugging easier by pinpointing the exact locations of data-related errors in the code.
4. It helps in understanding the flow of data through the program, which can be especially useful in complex systems.
5. It ensures comprehensive testing coverage by focusing on the interaction between variable definitions and uses.
6. It allows for static analysis to find faults without executing the program, which can save time and resources.
7. It is useful in maintenance phases to check that changes in the code do not introduce new data flow anomalies.
8. It can lead to performance optimizations by highlighting unnecessary data processing steps.

5.1.3 Challenges or Limitations of Data Flow Testing

1. Setting up data flow tests can be complex especially in large applications with extensive data interactions.
2. It consumes significant computational resources particularly in large code bases.
3. Manually conducting data flow testing is labor-intensive and prone to errors especially in identifying and tracing all relevant data paths.

4. There are limited commercial tools available that fully support data flow testing, which can hinder its adoption and effective implementation.
5. Requires a deep understanding of the program's architecture and data handling, demanding high expertise from testers.
6. Integrating data flow testing into existing testing frameworks may be challenging and can require significant adjustments to workflows.
7. The process of tracing and analyzing all potential data paths in the code is time-consuming, which can extend the testing phase.

5.1.4 Types of Data Flow Testing

Data flow testing focuses on the various points in a program where variables are defined, used, and potentially modified. It's a form of structural testing that helps in identifying how data values are manipulated across a program's execution. There are two primary types of data flow testing:

1. **Definition-Use Testing (Define-Use Testing) :** Definition-Use Testing revolves around tracking the points in the code where variables are defined (assigned a value) and subsequently used (where the value is accessed or modified). This type of testing helps uncover anomalies or bugs that may occur due to incorrect or unintended use of variable values within the application. It is particularly useful for ensuring that data integrity is maintained throughout the execution process.
2. **Slice-Based Testing :** Slice-Based Testing involves dividing the program into "slices," each focusing on a specific computation or functionality based on the program's data flow. Each slice is a subset of the program that captures the behavior with respect to a certain set of variables at a specific point of computation. This method can simplify understanding and debugging by isolating relevant parts of the code that affect the output related to the selected variables.

5.2 Define-Use Testing

Define/use testing is a technique in software testing that focuses on analyzing how variables are defined and used within a program to ensure correct data flow. By examining how variables are defined (assigned values) and where these values are used throughout the program, define/use testing helps identify potential issues related to data flow, such as variables being used before being defined, defined but never used, or defined multiple times before being used.

? What is Define-Use Testing?

Define-Use Testing is a type of structural testing focused on ensuring the correct use of variables within a program. It examines the relationships between where variables are defined (assigned values) and where those values are used in the program. This form of testing is highly detailed and aids in identifying potential issues such as the use of uninitialized variables or the improper use of variable values, which might lead to bugs or unexpected behaviors in the software.

5.2.1 Key Concepts and Definitions

Key concepts and definitions in define/use testing include:

1. **Defining Node (DEF(v, n))**
   - **Definition:** A node n in the program graph where the variable v is assigned or redefined.
   - **Meaning:** At a defining node, the program modifies the value of variable v. This change affects the state of v used in subsequent computations or decisions.
   - **Example:** In the statement x = 5;, x is assigned the value 5. This line is the defining node for x because it establishes x's value.
2. **Usage Node (USE(v, n))**
   - **Definition:** A node n in the program graph where the value of the variable v is accessed to influence computations or decisions but not modified.
   - **Meaning:** Usage nodes involve reading the variable v to execute calculations or control structures. They are critical for verifying that the value of v is being utilized correctly.
   - **Example :** In the statement y = x + 1;, x is accessed to compute y. This line is a usage node for x because x influences the calculation but its value remains unchanged.
3. **Predicate Use (P-use)**
   - **Definition:** A specific type of usage node where the variable v is part of a condition that determines the control flow of the program.
   - **Meaning:** Predicate uses are crucial in decision-making structures, where the program's path can change based on the variable's value.
   - **Example :** In if (x > 0) { // actions }, x is evaluated in a conditional expression. If x is greater than 0, the program executes the code within the block. This usage of x is a predicate use because it influences the flow of execution based on its value.
4. **Computation Use (C-use)**
   - **Definition:** A type of usage node where v is used in calculations or operations that contribute to the program's computational results but do not alter its control flow.
   - **Meaning:** Computation uses highlight how data is manipulated to produce new values or results within the program.
   - **Example:** In the statement total = price * quantity;, both price and quantity are involved in calculating total. This usage of price and quantity is considered computation use because they determine the value of total without affecting the decision paths of the program.
5. **Definition/Use Path (du-path)**
   - **Definition:** Paths within the program graph that originate at a definition of v and terminate at a usage of v.
   - **Meaning:** Du-paths help in understanding how data flows from its point of definition to where it is utilized, highlighting the lifecycle of variable usage within the program.
   - **Example :** Assuming x is defined as x = 5 at the start of a function and later used in if (x > 0), the path from x = 5 to if (x > 0) forms a du-path, tracing x's impact from definition to a critical decision point.

6. **Definition-Clear Path (dc-path)**
   - **Definition:** A du-path where v does not undergo any intermediate redefinitions between its initial definition and its subsequent use.
   - **Meaning:** Dc-paths ensure that the value of v remains unchanged from its definition to its usage, crucial for validating that operations are performed on intended data states.
   - **Example:** If x = 5 is directly followed by y = x + 1 without x being reassigned in between, the path from x = 5 to y = x + 1 is a dc-path. This ensures the value of x used in calculating y is exactly as initially defined, promoting reliability in data handling.

Example 1 Def-Use

Let's create a table that represents the data flow aspects of the function **calculateScore**. We will analyze the DEF, USE, P-use, and C-use points for each line of code:

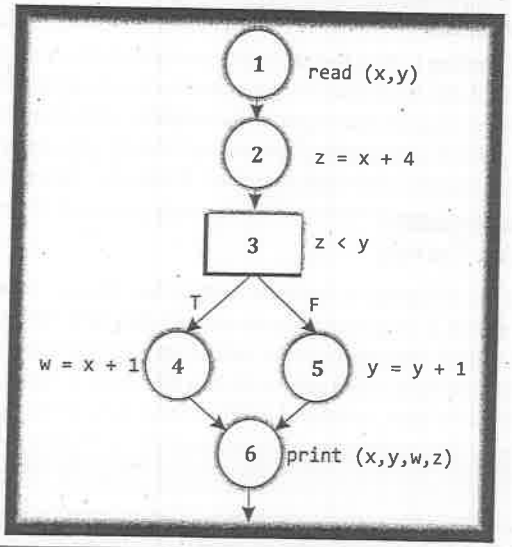
Line	Code	DEF	USE	C-use	P-use
Line 1	`int calculateScore(int grade, int bonus) {`				
Line 2	`int score;`	score			
Line 3	`score = grade;`	score	grade	grade	
Line 4	`if (grade > 90) {`		grade		grade
Line 5	`score += bonus;` `}`	score	score, bonus	score, bonus	
Line 6	`return score;`		score	score	

1. **DEF (Definition):** This column identifies where variables are defined or assigned values within the function.
   - score is first declared in line 2 and then assigned a value from grade in line 3.
   - score is updated again in line 5 where it's modified based on the bonus.
2. **USE (Usage):** This column shows where variables are read or used in computations.
   - grade is used in line 3 to assign a value to score.
   - grade is also used in line 4 to check the condition.
   - score and bonus are both used in line 5 to update score.
   - score is used again in line 6 when it's returned by the function.
3. **P-use (Predicate Use):** Indicates uses of variables in decisions that affect the control flow (predicate statements).
   - grade is used as a predicate in line 4 to decide whether to add the bonus to the score.
4. **C-use (Computation Use):** Shows where variables contribute directly to computation values but do not control the flow.
   - grade directly contributes to setting score in line 3.
   - score and bonus are used in a computation to update score in line 5.
   - score is used in the final computation that produces the output in line 6.
5. **du-path (Definition/Use Path):**
   - From score's definition in line 2 to its use in line 3.
   - From score's definition in line 3 to its use in line 5 and finally in line 6.

   - From grade's parameter definition to its use in line 3 and the predicate in line 4.
   - From bonus's definition as a parameter to its use in line 5.
6. **dc-path (Definition-Clear Path):**
   - From score's definition in line 3 directly to its first use in line 5 without intermediate redefinition of score (ignoring the sequential update within line 5).

Example 2 Def-Use

Line Number	Code	Definition	c-Use	p-Use
Line 1	read (x, y)	x, y		
Line 2	z = x + 2	z	x	
Line 3	if (z< y)			z, y
Line 4	w = x + 1 else	w	x	
Line 5	y = y +1	y	y	
Line 6	print (x, y, w, z)		x, y, w, z	



5.2.2 Define/Use Test Coverage Metrics

The hierarchy shown in the diagram illustrates the various levels of test coverage metrics from the least comprehensive to the most comprehensive in terms of data flow coverage in software testing.

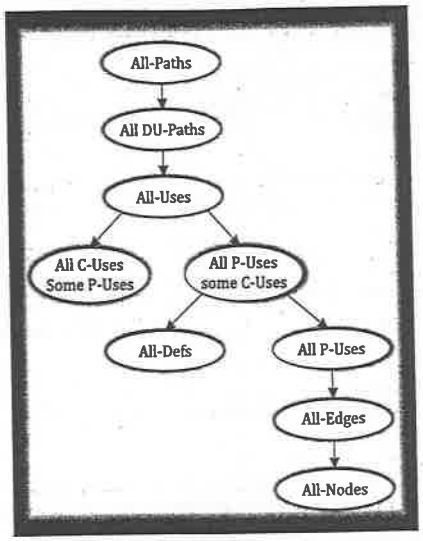


Figure 5.1 : Hierarchy of Data Flow Coverage Metrics

This hierarchical arrangement shows that as we move up the hierarchy, the coverage becomes more detailed and includes more scenarios and paths. Each level builds upon the one below it, adding additional complexity and breadth to the testing requirement.

1. **All-Nodes (AN)**
   - **Coverage:** Ensures that every node (or statement) in the program graph is executed at least once.
   - **Example:** If there is a function with five different statements, each statement must be executed during the test, regardless of the paths taken to reach them.
2. **All-Edges (AE)**
   - **Coverage:** Requires every edge in the program graph to be traversed at least once.
   - **Example:** In a function with conditional statements leading to different branches, each branch (edge) must be traversed.
3. **All P-Uses (APU):**
   - **Coverage:** All-P-Uses coverage is a testing criterion that requires exercising all predicate uses (P-uses) of variables in the program. It ensures that every decision point influenced by variable values is tested to validate the program's decision-making logic.
   - **Example:** All conditions in which a variable influences the control flow of the program must be tested. Predicate uses (P-uses) are scenarios where a variable's value determines the execution path taken in decision-making statements such as if-conditions, while-loops, for-loops, and case statements.

4. **All-Defs (AD)**
   - **Coverage:** Requires that for every variable definition, there must be at least one path that covers the definition to some use of the variable.
   - **Example:** If x = 5, there must be a test case that follows the path from this definition to its first use.
5. **All-C-Uses/Some P-Uses (ACU+P)**
   - **Coverage:** Requires all computation uses (C-uses) of all variables to be tested, and at least some of the predicate uses (P-uses).
   - **Example:** Every computation involving variables must be tested, and some decision paths that use these variables in their conditions must also be tested.
6. **All-P-Uses/Some C-Uses (APU+C)**
   - **Coverage:** All predicate uses (P-uses) of variables are tested, and some computation uses (C-uses).
   - **Example:** All decision branches influenced by variables are tested, along with some of their calculations.
7. **All-Uses (AU)**
   - **Coverage:** Ensuring every possible use of every variable from every one of its definitions is tested. Combines and extends the coverage of both All-C-Uses/Some P-Uses and All-P-Uses/Some C-Uses
   - **Example:** If variable x is defined at point A and used at point B, there should be a test case that covers the path from the definition at A to the use at B.

7. **All-DU-Paths (ADUP)**
   - **Coverage:** Requires that every definition-use path (DU-path) for every variable is executed, ensuring comprehensive coverage of variable flows.
   - Example: Every possible path from the point a variable is defined until it is used must be tested, covering all possible flows of that variable.
9. **All-Paths (AP)**
   - **Coverage:** The most comprehensive, requiring every possible path through the program to be executed.
   - **Example:** Every possible route through the software from start to finish is tested, covering all conceivable scenarios and edge cases.

5.2.3 How Def-Use Testing Works?

Def-Use Testing, or Definition-Use Testing, is a type of software testing that focuses on the interaction and relationship between the points in a program where variables are defined (assigned values) and the points where those values are used (referenced). It aims to ensure that the paths between these points are free from defects that could affect the program's execution and results.

How Def-Use Testing Works ?

1. **Identify Variables:** The first step is to identify all the variables used within the codebase that are relevant to the test.
2. **Construct a Program Graph:** Create a program graph where nodes represent statements or statement fragments, and edges represent the flow of control. This graph helps visualize how data flows through the program.
3. **Determine Definition Points (DEF):** Locate all the points in the code where each variable is defined. These are the nodes in the program graph where variables receive their values. Definitions could be through initializations, assignments, or through input read operations.
4. **Identify Usage Points (USE):** Identify all points where these variables are used. Uses can be in calculations (computation uses or C-use), or as part of conditions that influence the flow of execution (predicate uses or P-use).
5. **Trace Du-Paths:** For each variable, trace all paths from each definition point to each usage point in the program graph. These paths are known as definition-use paths (du-paths). Each path represents a potential route through the program that the execution might take, depending on inputs and conditions.
6. **Identify Definition-Clear Paths (dc-paths):** Among du-paths, identify those that do not have any intermediate redefinitions of the variable before it is used. These dc-paths ensure that the value used at the usage point is exactly the value assigned at the definition point without any modification.
7. **Test Each Path:** Design test cases to execute each du-path and dc-path identified. This involves crafting inputs and conditions that cause the program to follow these paths during execution. Each test should verify that the program behaves correctly along these paths, with particular attention to ensuring the correct values are transferred from definitions to uses.
8. **Analyze Results:** Assess the outcomes of the tests to confirm that all variable interactions are correct and that the software handles all defined and used values appropriately across different paths.

Practical Implementation

Def-Use Testing can be quite complex, especially in large programs with many variables and huge control flows. It requires thorough analysis and often automated tools to help identify all relevant paths and manage the extensive testing process. This method is particularly useful in unit testing and can be crucial for ensuring the integrity of critical sections of code, such as financial calculations, decision-making logic, and data handling operations.

Def-Use Testing not only helps find bugs related to variable misuse or mismanagement but also enhances understanding of the code's behavior, making it an invaluable tool for both testing and development phases.

5.2.4 Advantages and Disadvantages of Def-Use Testing

Advantages of Def-Use Testing

1. **Enhanced Coverage:**
   Def-Use Testing goes beyond simple code coverage by ensuring that all paths involving the definition and use of variables are tested. This can reveal data-specific bugs that might not be uncovered by less detailed testing approaches.
2. **Early Bug Detection:**
   By focusing on the flow of data within the application, Def-Use Testing can identify issues related to variable initialization, scope, and sequence of operations early in the development cycle.
3. **Improved Code Quality:**
   This testing approach encourages developers to pay close attention to how data is manipulated and transferred across the application, leading to more robust and error-free code.
4. **Supports Debugging:**
   Since Def-Use Testing maps out how variables are used throughout a program, it can be an excellent tool for debugging, helping developers understand where things might go wrong.
5. **Facilitates Regression Testing:**
   Once a Def-Use path is established and tested, any future changes that affect these paths can be quickly identified and retested, making it easier to manage regression testing.

Disadvantages of Def-Use Testing

1. **Complexity:**
   Creating and managing the program graph for larger applications can be highly complex and time-consuming. Identifying all relevant Def-Use paths in a large codebase often requires sophisticated tool support.
2. **High Overhead:**
   The level of detail involved in tracing all possible paths from definitions to uses can lead to significant testing overhead in terms of time and resources, making it less suitable for projects with tight deadlines.
3. **Requires Deep Understanding of the Code:**
   To effectively implement Def-Use Testing, testers and developers must have a deep understanding of the code's structure and logic. This high knowledge requirement can be a barrier for new team members or those unfamiliar with the code.
4. **Tool Dependency:**
   Effective Def-Use Testing often relies on specialized tools that can analyze and visualize complex program graphs. These tools can be expensive and may have a steep learning curve.
5. **Limited to Available Code:**
   As a form of white-box testing, Def-Use Testing can only be performed when the source code is available. It's not applicable to black-box testing scenarios where internal code structures are not accessible.
6. **Does Not Cover Non-Functional Testing:**
   This method primarily assesses the correctness of program execution concerning data flow and does not address non-functional aspects such as performance, usability, or scalability.

5.2.5 Example- The Commission Problem using Define-Use Testing

We will use the commission problem and its program graph to illustrate these definitions. The numbered pseudocode and its corresponding program graph are shown in Figure 4.1. This program computes the commission on the sales of the total numbers of locks, stocks, and barrels sold. The while loop is a classic sentinel controlled loop in which a value of –1 for locks signifies the end of the sales data. The totals are accumulated as the data values are read in the while loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

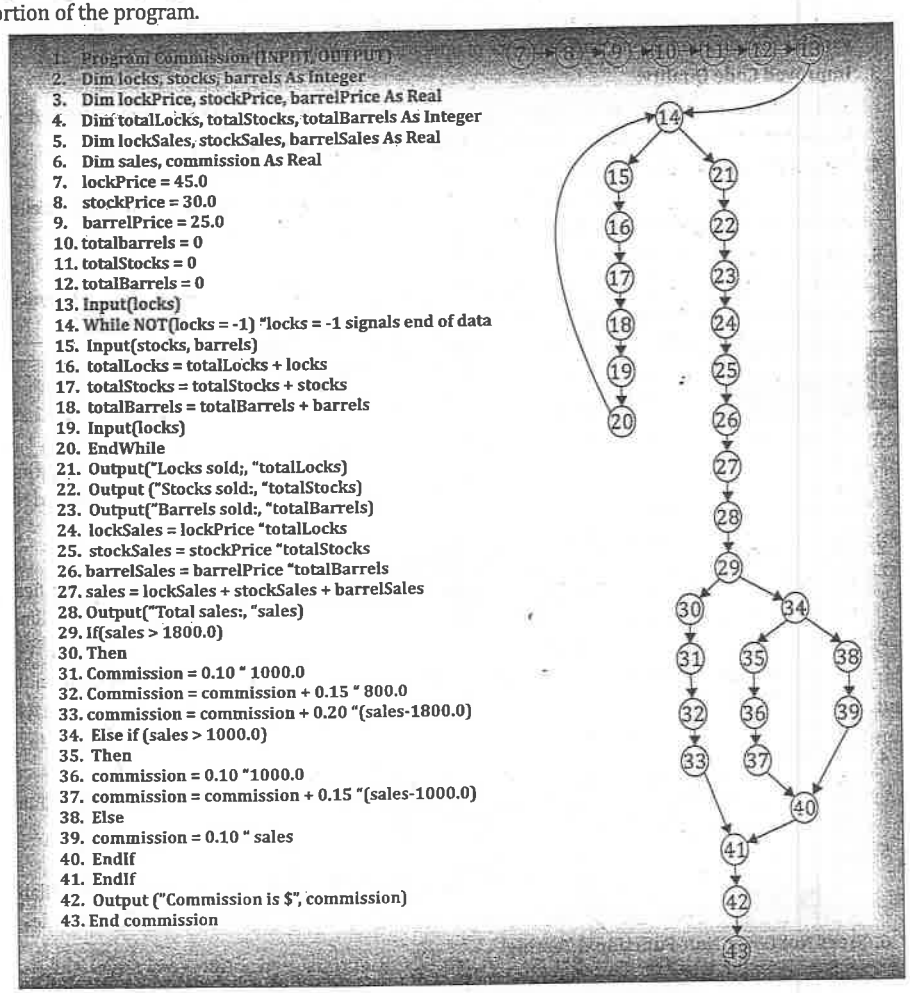


Figure 5.2 : Commission Problem and its Program Graph

Variable Definitions and Usages

Table 4.1 lists the define and usage nodes for the variables in the commission problem. We use this information in conjunction with the program graph in Figure 4.1 to identify various definition-use and definition-clear paths. It is a judgment call whether non-executable statements such as constant and variable declaration statements should be considered as defining nodes.

Variable	Defined at Node	Used at Node
lockPrice	7	24
slockPrice	8	25
banelPrice	9	26
lotallocks	10,16	16, 21, 24
totaiStockts	11, 17	17, 22, 25
totalearrels	12,18	18, 23, 26
locks	13, 19	14, 16
Mocks	15	17
Barrels	15	18
lockSales	24	27
stockSales	25	27
banelSales	26	27
Sales	27	28, 29, 33, 34, 37, 38
Commission	31, 32, 33, 36, 37, 38	32, 33, 37, 42

Table 4.1 Define/Use Nodes for Variables in Commission Problem

DU Paths (Definition-Use Paths)

DU paths trace the flow of data from points where variables are defined (assigned values) to points where these values are used (either in computations or conditions).

Example DU Paths from the Provided Graph:

1. **DU Path for totalLocks:**
   - Defined at Node 16 (totalLocks = totalLocks + locks)
   - Used at Node 21 (Output("Locks sold", totalLocks))
   - Used at Node 24 (lockSales = lockPrice * totalLocks)
   - Path : 16, 21, 24
2. **DU Path for commission:**
   - Path1 : Defined at Node 31,Used at 32, 33 and Finally at 42.
   - Path 2: Defined at Node 36,Used at 37 and Finally at 42.
   - Path 3: Defined at Node 39,Used at at 42.

DC Paths (Definition-Clear Paths)

DC paths are special types of DU paths where the variable, once defined, does not undergo any redefinition before its use. These paths ensure that the value used is exactly the one initially defined, with no intermediate modifications that could alter the behavior or outcome.

Example DC Paths from the Provided Graph:

1. **DC Path for sales:**
   - Defined at Node 27 (sales = lockSales + stockSales + barrelSales)
   - Used at Node 28, 29 without redefinition in between.

   This ensures that the calculations of commission based on sales at subsequent nodes (32, 33, 37, 39) are based on the same sales value computed at Node 27.

Comprehensive Analysis of All Paths

To identify all DU and DC paths effectively, each variable's definition and use points need to be traced through the program's flow. For each variable:

- Trace from every definition point to all possible use points without crossing another definition of the same variable (for DC paths).
- Trace from definition to use, even if the variable is redefined along the path (for DU paths).

For a thorough testing strategy, every such path should be verified to ensure it behaves as expected under various conditions, including edge cases and potential error conditions. This approach not only helps in validating the logic but also aids in uncovering hidden bugs related to data handling.

Applying Def-Use Testing

By tracing these variable flows from their points of definition to their points of use, testers can ensure that each path is correct and all potential interactions are accounted for. For example, testing the path for Sales, starting from its computation at node 27 through its multiple uses in commission calculations, involves validating that the sales values are accurately calculated and correctly influence the commission outcomes under different sales conditions. This would require creating test cases that:

- Validate correct sales calculations from nodes 24, 25, 26 to node 27.
- Ensure the commission is calculated correctly based on the sales values through various branches (node 32 for sales ≤ 1000, node 33 for sales between 1000 and 1800, and node 37 for sales > 1800).

Each path would be tested to verify that no intermediate redefinitions incorrectly alter the expected outcomes, thus ensuring the integrity of the computation across different parts of the program. This structured approach highlights how data flow testing provides a comprehensive examination of the software's logical flow, enhancing reliability and correctness.

5.3 Slice-Based Testing

Slice-based testing is a testing technique that focuses on verifying specific parts of a program by analyzing "slices" of code. A program slice consists of all parts of a program that affect the values computed at some point of interest, known as a slicing criterion. This criterion typically includes a variable and a program point. The primary goal of slice-based testing is to isolate and test parts of a program that contribute to the outcome at specific points. It reduces the complexity involved in understanding and testing the entire program.

Key Concepts:

1. **Program Slice:**
   - A subset of a program that potentially affects the values computed at certain points.
   - It helps in identifying the relevant portions of the code that contribute to the outputs at specific locations.
2. **Slicing Criterion:**
   - It consists of a variable of interest and a specific statement or line number in the program.
   - It is used to determine which parts of the code should be included in the slice.

? What is Slice Based Testing?

Slice-Based Testing is a software testing technique that involves isolating and testing specific portions or "slices" of a program's code that impact the values computed at particular points of interest known as slicing criteria. These slices typically include all code segments that contribute to the outcome based on variables and program locations specified in the slicing criterion. The primary goal of slice-based testing is to focus testing efforts on critical parts of the program that influence specific results, thereby reducing the complexity of testing the entire program and improving test effectiveness.

In simple words, Slice-based testing also known as program slicing is a software testing technique that focuses on specific portions of the code relevant to a particular variable or output value.

Example: Slice Based Testing

Consider a simple program that calculates the total cost of items purchased, including tax, based on various inputs:

```
1. int calculateTotalCost(int price, int quantity, float taxRate) {
2.     int subtotal = price * quantity;           // Calculates the subtotal
3.     float taxAmount = subtotal * taxRate;      // Calculates the tax amount
4.     int totalCost = subtotal + (int)taxAmount; // Calculates the total cost
5.     print("Subtotal: ", subtotal);
6.     print("Tax: ", taxAmount);
7.     print("Total Cost: ", totalCost);
8.     return totalCost;
9. }
10. int main() {
11.     int finalCost = calculateTotalCost(100, 2, 0.05);
12.     print("Final Purchase Cost: ", finalCost);
13.     return 0;
14. }
```

Steps in Slice-Based Testing:

1. **Identify the Slicing Criterion:**
   - The variable of interest is totalCost.
   - Point of Interest: Line 4, where totalCost is calculated.

2. **Determine the Program Slice:**
   - The relevant code that affects totalCost includes:
     a) Calculation of subtotal on line 2.
     b Calculation of taxAmount on line 3.
     c) Addition of subtotal and taxAmount to form totalCost on line 4.
3. **Construct the Slice:**
   - Extract the lines of code that directly contribute to the slicing criterion

```
2. int subtotal = price * quantity;
3. float taxAmount = subtotal * taxRate;
4. int totalCost = subtotal + (int)taxAmount;
```

4. **Create Test Cases:**

- **Test Case 1:**
  Input: price = 100, quantity = 2, taxRate = 0.05
  Expected Output: totalCost = 210
  Rationale: Subtotal is 200, tax is 10, so total cost should be 210.
- **Test Case 2:**
  Input: price = 50, quantity = 4, taxRate = 0.10
  Expected Output: totalCost = 220
  Rationale: Subtotal is 200, tax is 20, so total cost should be 220.

These steps ensure that the critical functionality involving the computation of totalCost is rigorously tested, isolating the specific computations for targeted verification. This approach simplifies testing, making it more focused and efficient, particularly for verifying the correctness of calculations within the program.

5.3.1 Characteristics or Features of Slice Based Testing

Slice-based testing is a specialized approach within software testing focused on analyzing specific "slices" of code related to certain variables or conditions. The key characteristics or features of slice-based testing are:

1. **Slicing Criterion:**
   Testing revolves around a specific variable or set of variables that influence the program's behavior at a certain point or over a section of the program. The slicing criterion typically includes the variable of interest and the specific location in the code.
2. **Program Slices:**
   A slice is a subset of a program that includes all the statements that could affect the values of the variables in the slicing criterion at specific points. It isolates the parts of code that are directly relevant to the criterion.
3. **Data Flow Analysis:**
   Slice-based testing relies heavily on data flow analysis to determine how data moves through the program and which parts of the program are affected by and affect the slicing criterion.
4. **Static and Dynamic Slicing:**
   - **Static Slicing:** Analyzes the program's source code without executing it, providing slices based on potential data flow.

   - **Dynamic Slicing:** Generates slices based on actual execution paths and runtime data, which are specific to a particular execution instance.
5. **Reduction of Complexity:**
   By focusing on slices, testers can reduce the complexity of the test environment and concentrate on verifying specific functionalities without the overhead of the entire program's context.
6. **Error Localization:**
   Facilitates precise error localization within the slice under test, making it easier to detect where exactly defects are occurring within the subset of the program being analyzed.
7. **Efficiency in Regression Testing:**
   Especially useful in regression testing where changes to the code base are verified to ensure that new updates have not introduced new bugs into previously tested slices of the program.
8. **Integration with Other Testing Techniques:**
   Often used in conjunction with other testing techniques to ensure comprehensive coverage. While slice-based testing targets specific functionalities, other tests can cover areas outside the scope of slices.
9. **Tool Dependent:**
   The generation and analysis of slices typically require specialized tools that can perform static or dynamic analysis and manage the complexities involved in isolating slices effectively.
10. **Focused on Functional Dependencies:**
    Emphasizes testing the functional dependencies within the code, particularly how specific variables or operations influence the behavior of the system.

These characteristics make slice-based testing a powerful tool for dealing with complex software systems, particularly in modular or object-oriented environments where understanding interactions and dependencies can be critical for ensuring correct behavior.

5.3.2 Importance or Benefits of Slice-Based Testing

Slice-Based Testing offers several important benefits in software testing:

1. **Focused Testing:** By isolating and testing specific slices of code, this technique allows testers to concentrate their efforts on critical parts of the program that directly impact the desired outcomes. This focused approach enhances the effectiveness of testing by targeting key areas.
2. **Reduced Complexity:** Testing the entire program can be complex and time-consuming. Slice-Based Testing simplifies the testing process by breaking down the program into smaller, manageable slices, making it easier to understand and test individual components.
3. **Improved Debugging:** By testing slices of code that contribute to specific results, identifying and debugging errors becomes more efficient. Testers can pinpoint issues in the relevant parts of the program, leading to quicker resolution of defects.
4. **Enhanced Test Coverage:** Since slice-based testing targets specific parts of the program, it helps ensure that critical areas are thoroughly tested. This approach can improve test coverage by focusing on the most important aspects of the software.

5. **Cost-Effective Testing:** By prioritizing testing efforts on essential program components, slice-based testing can optimize resource allocation and reduce unnecessary testing of less critical areas. This can result in cost savings for the testing process.

5.3.3 Limitations or Disadvantages of Slice-Based Testing

While Slice-Based Testing offers several benefits, it also has some limitations and disadvantages:

1. **Limited Coverage:** Since slice-based testing focuses on specific parts of the program, there is a risk of overlooking interactions and dependencies between different slices. This may result in incomplete or limited test coverage, leaving certain paths untested.
2. **Difficulty in Slice Identification:** Identifying the appropriate slicing criteria and determining the relevant slices can be challenging, especially in complex programs. Incorrect slicing criteria selection may lead to ineffective testing and missed defects.
3. **Maintenance Overhead:** Maintaining and updating slice-based tests as the program evolves can be difficult job. Changes in one slice may require adjustments in related slices, increasing maintenance overhead and effort.
4. **Limited Scope:** Slice-based testing may not be suitable for all types of software projects or testing scenarios. It may be less effective for systems with huge interdependencies or where end-to-end testing is crucial.
5. **Risk of False Positives/Negatives:** Testing individual slices in isolation may result in false positives (passing tests despite defects) or false negatives (failing tests due to external factors). This can lead to inaccurate assessment of the software quality.
6. **Complexity in Integration Testing:** Integrating individual slices back into the complete program for end-to-end testing can be complex. Ensuring that all slices work together seamlessly and do not introduce new issues during integration can be challenging.
7. **Tool Dependency:** Effective slice-based testing often relies on advanced tools to identify and manage slices, which may not be readily available or it may come with high cost..

5.4 Guidelines and Observations for Data Flow Testing

Data flow testing, including both define/use testing and slice-based testing, offers a detailed approach to uncovering software bugs that might not be easily caught through traditional testing techniques. Here are some guidelines and observations related to these techniques:

Guidelines for Data Flow Testing:

1. **Understand Program Structure:**

   Gain a thorough understanding of the program's structure. Familiarity with the control flow graph of the application is crucial as it helps in identifying all potential def-use pairs and relevant slices.

2. **Select Appropriate Slicing Criteria:**

   Define clear and meaningful slicing criteria based on the variables and conditions critical to the application's functionality. This will determine the effectiveness of the slice in isolating relevant portions of the code.

3. **Use Appropriate Tools:**

   Utilize specialized tools designed for data flow analysis. These tools can automate the process of identifying def-use chains and generating program slices, making the testing process more efficient and less error-prone.

4. **Integrate with Other Testing Methods:**

   Combine data flow testing with other testing techniques, such as unit testing, integration testing, and system testing, to enhance overall test coverage and effectiveness.

5. **Prioritize Test Cases:**

   Prioritize test cases based on the complexity and criticality of the def-use paths or slices. Focus on paths that have a higher risk of failure or are crucial for the application's performance.

6. **Regularly Update Test Cases:**

   Update test cases as the software evolves. Changes in the codebase can affect existing def-use relationships and necessitate new slices or revised testing strategies.

7. **Document Test Cases and Results:**

   Maintain detailed documentation of test cases, the rationale for their selection, and the outcomes. This documentation is invaluable for future testing cycles and for understanding the impact of changes in the code.

Observations on Data Flow Testing

1. **Complexity and Cost:**

   Data flow testing can be complex and costly due to the need for detailed analysis of the codebase and the potential for a large number of test cases, especially in large applications with complex logic.

2. **High Effectiveness for Certain Bugs:**

   Particularly effective at finding bugs related to improper use of data, such as the use of uninitialized variables, incorrect updates, and violations of sequential dependencies.

3. **Tool Dependency:**

   The effectiveness of data flow testing is often dependent on the quality of the tools used, as manual identification and testing of data flow paths can be error-prone and impractical in large applications.

4. **Limited by Feasibility of Paths:**

   Not all identified paths may be feasible to execute due to runtime conditions or constraints in the code that were not apparent at compile-time.

5. **Learning Curve:**

   There is a significant learning curve associated with understanding and implementing data flow testing effectively, particularly when dealing with complex systems or languages with intricate data flow characteristics.

6. **Integration with Development Processes:**

   Data flow testing is most effective when integrated into the development process, allowing for immediate testing and bug fixing, which aligns with agile methodologies.

These guidelines and observations can help software development and testing teams effectively implement and benefit from data flow testing techniques, thereby enhancing the reliability and robustness of their software products.

5.5 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Data Flow Testing ?
2. Define Data Flow Testing.
3. Mention the Types of Data Flow Testing.
4. What is Define-Use Testing?
5. Define du-path and dc-path.
6. What is Slice Based Testing?

Section - B

Each Question Carries Five Marks

1. What is Data Flow Testing? Explain the Characteristics of Data Flow Testing.
2. Explain the Benefits of Data Flow Testing.
3. Write the Challenges or Limitations of Data Flow Testing.
4. What is Define-Use Testing? Explain Define/Use Test Coverage Metrics.
5. How Def-Use Testing Works?
6. Write the Advantages and Disadvantages of Def-Use Testing.
7. Discuss the Characteristics or Features of Slice Based Testing.
8. Explain the Importance or Benefits of Slice-Based Testing.
9. Explain the Limitations or Disadvantages of Slice-Based Testing.
10. Discuss the Guidelines and Observations for Data Flow Testing.

Section - C

Each Question Carries Eight Marks

1. Elaborate Define-Use Testing with detailed example. Mention its Characteristics.
2. Discuss the Commission Problem using Define-Use Testing.
3. Elaborate Slice Based Testing with detailed example. Mention its Characteristics.

❖❖❖❖❖

UNIT - III

CHAPTER 6

INTEGRATION TESTING

Contents

6.1 Levels of Testing

In Chapter 1, we learned about the basic types of testing - unit, integration, and system testing - in the traditional software development process, especially in the waterfall model. Each type of testing corresponds to different stages of development, from detailed design to combining parts and ultimately checking the whole system. In modern software development methods, these testing stages are adjusted and used in new ways to fit the latest approaches and technologies.

6.1.1 Levels of Testing in Different Life Cycle Models

In the realm of software development, various life cycle models dictate how products are developed, tested, and maintained. Each model has its own approach to testing, tailored to fit the specific needs and stages of the development process. Here, we delve into how testing levels are integrated into different life cycle models.

1. **Waterfall Model :** The traditional Waterfall model is linear and sequential, generally divided into distinct phases: requirements, design, implementation, testing, and maintenance. It's predicated on the idea that each phase must be completed and its deliverables approved before the next phase begins.

   Testing in Waterfall:

   - **Unit Testing:** Conducted after the implementation or coding phase. Each component is tested in isolation to ensure it works as expected.
   - **Integration Testing:** Follows unit testing; it involves testing combined parts of the application to ensure they work together correctly.
   - **System Testing:** This final testing stage assesses the complete and integrated software product to verify that it meets the specified requirements.

2. **Incremental Development :** Incremental development breaks the software product into smaller, manageable increments. Each increment is fully developed and tested, which provides functionality gradually throughout the life cycle.

   Testing in Incremental Development:

   - **Unit Testing:** Performed on each increment as it is developed.
   - **Integration Testing:** As new increments are added, integration testing is performed to ensure new and old increments work harmoniously.
   - **System Testing:** Conducted on each complete increment to ensure it meets the broader system requirements.

3. **Evolutionary Development :** Suitable for projects with unclear requirements, this iterative approach allows the software to evolve through repeated cycles of development and testing, based on continuous user feedback and evolving requirements.

   Testing in Evolutionary Development:

   - **Iterative Testing:** Each iteration includes unit, integration, and system testing to ensure the evolving parts of the application function correctly together.

   - **Regression Testing:** Frequently conducted to ensure that new changes do not disrupt existing functionality.

4. **Spiral Model :** The Spiral model combines elements of both iterative development and systematic risk management, involving iterative refinement through spiraling cycles, each consisting of planning, risk analysis, engineering, and evaluation.

   Testing in Spiral Model:

   - **Progressive Testing:** At each cycle's end, progressive testing (including unit, integration, and system testing) is conducted, tailored to the new features developed in that cycle.
   - **Risk Analysis:** Special emphasis on testing to mitigate identified risks at each cycle.

5. **Rapid Prototyping :** This model focuses on quickly developing prototypes to refine requirements through early user feedback, helping to better define what the final system should look like.

   Testing in Rapid Prototyping:

   - **Prototype Testing:** Each prototype is tested to evaluate its functionality and to gather user feedback, which is crucial for refining subsequent prototypes.

6. **Executable Specifications :** This approach extends the concept of rapid prototyping by developing executable specifications that serve as a functional prototype of the system.

   Testing in Executable Specifications:

   - **Continuous Testing:** As the executable specification is developed and refined, continuous testing is conducted to ensure that the specification meets the desired functionality and behavior.

6.2 The SATM System

As we discussed in Chapter 1. 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.

6.2.1 Overview of the SATM System

The SATM system is designed as a teaching tool or simplified version of a commercial ATM system, containing only the essential features required to perform basic ATM functions such as transactions and user interactions. This reduced complexity allows for focused development and testing, making it an ideal candidate for educational purposes.

Screens and Terminal Layout

- **Screens for the SATM system :** The screens illustrates the user interface flow, showcasing how users interact with the system through various screens. For example, it includes screens for entering a PIN, selecting transaction types, and receiving notifications about the transaction status. Each screen is designed to guide the user smoothly through their banking transactions.

- **The SATM Terminal :** It details the physical components of the ATM, such as the keypad for input, a screen for display, a card reader, receipt printer, and cash dispensing units. This layout is critical for understanding the user interaction points and for planning the physical security measures and usability testing.

Context Diagram of the SATM (Simple ATM) System

The context diagram of the SATM (Simple ATM) system visually depicts how the ATM interfaces with external entities and processes inputs and outputs.

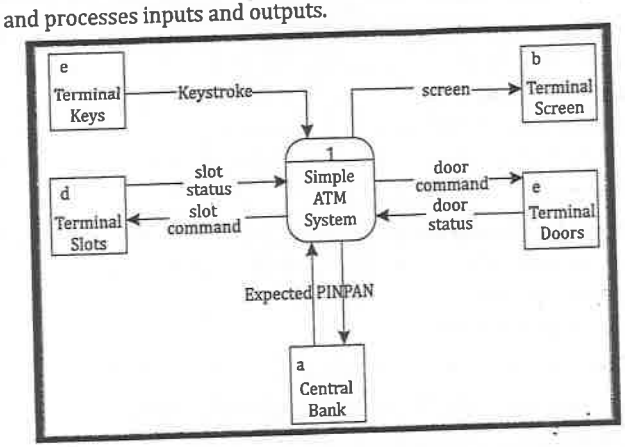


Fig 6.1 : Context Diagram of the SATM (Simple ATM) System

1. **Simple ATM System:** Central to the diagram, it processes all user interactions and manages communication with external entities like the bank and hardware components of the ATM.
2. **External Entities and Interactions:**
   - **Terminal Keys:** Users input data such as PINs and transaction choices here, which are then processed by the ATM system.
   - **Terminal Screen:** Displays transaction information, instructions, and prompts to the user, controlled by the ATM system based on the transaction status and inputs.
   - **Terminal Slots and Doors:** Handle physical aspects of transactions like card input, cash dispensing, and receipt printing. The ATM system sends commands to these components based on user requests and transaction requirements.
3. **Data Flows:**
   - **Keystrokes to System:** Data from user inputs goes to the ATM system for processing.
   - **Information to Screen:** The ATM system sends data to be displayed.
   - **Commands to Slots/Doors:** Controls mechanical actions like opening deposit doors or activating cash dispensers.
4. **Integration with Central Bank:** The ATM system communicates with the central bank to authenticate transactions, check balances, and perform account updates, ensuring secure and accurate banking operations.

This diagram provides a clear visual of the SATM system's scope, highlighting its interactions with users and the central bank, which is essential for understanding system behavior and dependencies.

Entity/Relationship Model of the SATM System:

The entity/relationship (E/R) model of the SATM system depicted above provides a structured representation of how different entities within an ATM system interact and relate to each other. It presents how different entities such as customers, accounts, and transactions relate within the system. It's vital for ensuring that the database design can support all necessary operations and for validating the integrity of transaction processing and data storage.

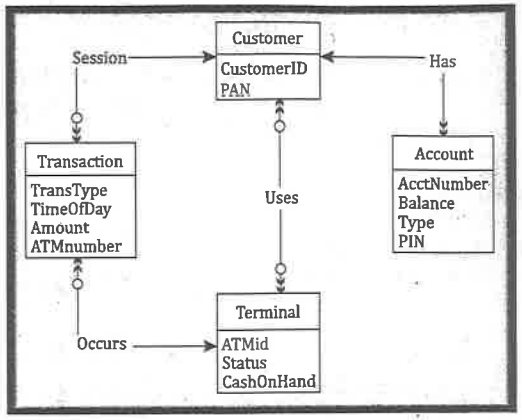


Fig 6.2 : Entity/Relationship Model of the SATM System

1. **Entities and Their Attributes:**
   - **Customer:** Identified by CustomerID and PAN (Personal Account Number).
   - **Account:** Associated with each customer, identified by AcctNumber, and includes attributes such as Balance, Type of account, and PIN (Personal Identification Number).
   - **Session:** Represents an interaction session between the customer and the ATM, not detailed in terms of attributes in this diagram.
   - **Transaction:** Occurs during a session and includes attributes like TransType (Transaction Type), TimeOfDay, Amount, and ATMNumber.
   - **Terminal:** Represents the ATM terminal where transactions occur, characterized by ATMid, Status, and CashOnHand.
2. **Relationships:**
   - **Has:** Links Customer to Account, indicating ownership or association of an account by a customer.
   - **Uses:** Connects Customer to Transaction, implying that customers use their accounts to perform transactions.
   - **Occurs:** Transaction takes place at a Terminal, denoting the physical location and device where transactions are executed.

3. **Interactions:**
   - The flow from Customer to Transaction through the Account entity indicates that transactions are not just linked directly to the customer but are performed through an account that the customer holds.
   - The terminal is integral to the execution and management of transactions, indicating the interaction of physical ATM components with the transactional data.
4. **System Functions and Operations:**
   - The model suggests that any transaction is recorded with details of type, time, amount, and the specific ATM used, which can be critical for audit trails, security checks, and operational logs.
   - The status of the terminal and the cash on hand are crucial for managing the availability and functionality of ATMs, ensuring that transactions can be successfully processed at any given terminal.

Finite State Machines

The below diagram is a finite state machine (FSM) for the PIN entry process in an ATM system, specifically detailing how the ATM software handles the PIN entry verification.

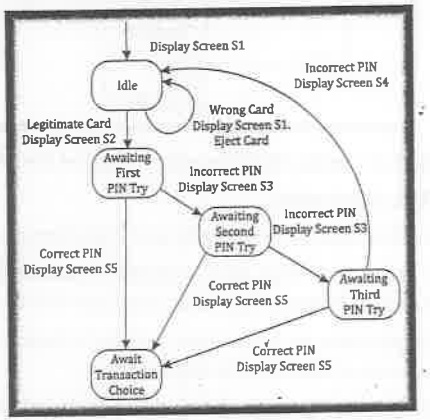


Fig 6.3 : PIN Entry Finite State Machines

1. **States:**
   - **Idle:** This is the default state where the machine waits for an action to initiate a session, typically by inserting an ATM card.
   - **Legitimate Card Display Screen S2:** The system transitions to this state when a valid card is recognized, prompting the user to enter their PIN.
   - **Awaiting First PIN Try:** After the PIN entry prompt, the system waits for the user to input their PIN.
   - **Awaiting Second PIN Try and Awaiting Third PIN Try:** If the first attempt is incorrect, the system allows up to two additional attempts to enter the correct PIN.

   - **Correct PIN Display Screen S5:** This state is reached when a correct PIN is entered at any try, leading to the transaction choice screen.
   - **Await Transaction Choice:** After successful PIN verification, the user is prompted to choose the transaction they wish to perform.
2. **Transitions:**
   - **From Idle to Legitimate Card:** Triggered by the insertion of a card that passes initial validation (not expired, not on a blocklist, etc.).
   - **Incorrect PIN:** If the PIN entered is incorrect, the machine moves to the next attempt state or, after three failed attempts, may lock the card or ask for it to be ejected, not shown in this diagram.
   - **Correct PIN:** If the correct PIN is entered during any of the three tries, the machine moves to the Correct PIN state and subsequently to the transaction choice.
3. **Error Handling:**
   - **Wrong Card Display Screen S1, Eject Card:** If an invalid card is inserted (possibly a card that is damaged, unreadable, or not supported), the system will display an error message and eject the card.
4. **Important Considerations:**

   This FSM ensures security by limiting the number of PIN attempts to three.

   It enhances user experience by providing immediate feedback on the status of the PIN entry (correct, incorrect).

   The transition to "Await Transaction Choice" after a correct PIN entry seamlessly connects the user authentication phase to transaction operations.
5. **Use in Testing:**

   Each transition and state can be explicitly tested to ensure the ATM's software handles every possible scenario correctly, ensuring robust handling of user inputs and errors.

   This state machine diagram is crucial for developers and testers as it outlines the logical flow of user interactions needed for PIN verification, a critical security feature in ATM systems.

6.2.2 Testing Strategy

Testing strategies for a Simple Automatic Teller Machine (SATM) system should include a variety of testing methods to ensure that both the functional requirements and the user interactions are handled correctly. Using the detailed design provided by the entity/relationship models and finite state machines, the SATM system is meticulously tested through a structured integration approach:

Structured Approach to Testing the SATM System:

1. **Unit Testing:**

   Objective: Validate the correctness of individual units or components in isolation.

   Test Components: Each screen (e.g., Welcome, Enter PIN, Balance Display), handling of input fields, and response actions (e.g., button presses).

Methods: Use mock objects for hardware interactions like card reading and cash dispensing to simulate real-world usage.

2. **Integration Testing:**

   Objective: Ensure that the integrated subsystems work together as expected.

   Test Scenarios: Sequences involving multiple screens and actions, such as the process from card insertion to PIN verification to transaction selection.

   Methods: Use integration harnesses to combine two or more units, testing their interfaces and interactions.

3. **System Testing:**

   Objective: Validate the complete and integrated SATM software system for compliance with its requirements.

   Test Scenarios: Complete user workflows from start to finish, including exceptional and error handling scenarios such as incorrect PIN entries, failed transactions, network errors, etc.

   Methods: Conduct black-box testing, simulating user interactions on the actual hardware where possible or within a simulated environment.

4. **Interface Testing:**

   Objective: Ensure that the ATM's interfaces with external systems (e.g., bank databases, networking systems) work as intended.

   Test Scenarios: Transactions that require real-time data exchange like balance checks, deposits, withdrawals, and interconnectivity with central banking systems.

   Methods: Use stubs and drivers to simulate external systems where direct testing is not feasible.

5. **Usability Testing:**

   Objective: Verify that the ATM machine is user-friendly and all instructions and error messages are clear and helpful.

   Test Scenarios: User interaction with the ATM under various scenarios, focusing on the clarity of the instructions and the ease of navigating through screens.

   Methods: Conduct tests with real users or use usability experts to evaluate the system.

6. **Performance Testing:**

   Objective: Ensure the system performs well under expected and peak load conditions.

   Test Scenarios: Simultaneous access by multiple users, rapid sequence of transactions, handling of peak load times.

   Methods: Use load testing tools to simulate multiple users and stress testing tools to test the limits of system capacity.

7. **Security Testing:**

   Objective: Confirm that the system securely handles user data and protects against potential breaches.

   Test Scenarios: Attempts to breach security measures, testing of encryption methods, session management, and PIN handling.

   Methods: Conduct vulnerability scans, penetration testing, and employ ethical hackers to try to exploit any weaknesses in the system.

8. **Regression Testing:**

   Objective: Ensure that new changes do not adversely affect existing functionality.

   Test Scenarios: Re-run previous tests to verify that new code changes have not introduced new bugs.

   Methods: Automated testing can be particularly useful here to rerun a large number of tests efficiently every time the software is changed.

9. **Compliance Testing:**

   Objective: Verify that the ATM complies with all relevant regulations and standards.

   Test Scenarios: Compliance with financial regulations, data protection laws, and hardware safety standards.

   Methods: Review compliance guidelines and use standard checklists to ensure all criteria are met.

Each of these testing strategies provides a systematic approach to ensuring that the SATM operates reliably, securely, and efficiently, providing a high-quality service to its users.

6.2.3 Structural and Behavioural Insights

Structural Insights:

The structural insights of the Simple ATM System (SATM) highlight how integration and system testing differ and where they intersect within the development cycle.

1. **Integration Testing Context:**
   - **Integration Testing:** This focuses on the interfaces and interaction between integrated units, testing if they function together as expected. In the context of the SATM, this means ensuring that interactions between components like the terminal, transaction processing, and customer account handling are seamless and error-free.
   - **Role in Waterfall Model:** Traditionally, in the waterfall model, integration testing occurs after "unit testing" and before "system testing," acting as a crucial step to ensure that individually tested units work together.
2. **System Testing Context:**
   - **System Testing:** This level tests the complete and integrated software to verify that it meets the specified requirements. For SATM, system testing would ensure that all functionalities, including card reading, PIN validation, transaction processing, etc., meet the customer and bank operational standards.
   - **Testing against Requirements:** System testing is usually aligned with the requirements specifications, ensuring that the software behaves as intended in a real-world scenario.

3. **Identifying Extremes:**

   Identifying the "extremes" in system testing, which involves testing the system's behavior under extreme conditions. Example scenarios provided include various combinations of card insertions, PIN entries, and transaction processes to cover possible real-world use cases.

4. **Integration vs. System Testing:**

   The integration testing focuses more on the preliminary design and the interaction between units, system testing is more about aligning with the final requirements specification, ensuring the end product functions as a cohesive unit.

Behavioral Insights:

Behavioral insights revolve around understanding how the system behaves in response to various inputs and conditions, typically viewed from the system's "port boundary," which involves inputs and outputs that the system interacts with in its operational environment.

1. **Port Boundary Concept:**
   - **Definition:** The port boundary concept refers to the interaction points where the system communicates with the external world (users, other systems). For SATM, these include user inputs through the terminal, outputs displayed on the screen, and physical interactions like card insertion or money dispensation.
   - **Testing Implications:** Testing should ensure that all interactions at these boundaries occur as expected. Inputs and outputs should be tested to verify that the system reacts correctly to user actions and displays the correct information.
2. **System Test Cases:**
   - **System-Level Understanding:** System test cases are derived from how well the system adheres to its specified functionalities at the port boundary. This includes ensuring that each user action triggers the appropriate response in the system, which is crucial for user satisfaction and system reliability.
   - **Interleaved Sequence Testing:** This approach to testing suggests using interleaved sequences of port events as test cases, simulating real-world interactions to verify the system's behavior comprehensively.
3. **Analytical View of Testing:**
   - **Unit-Level Threads:** Analysis begins at the unit level, where individual functionalities are tested.
   - **Integration-Level Threads:** These are more complex, involving sequences of unit-level threads, and are essential for understanding how well the units integrate to perform a complete function within the system.

By focusing on both structural and behavioral insights, the testing strategy for the SATM system ensures a comprehensive evaluation of both individual and integrated functionalities, emphasizing real-world usage and interaction with the system's external environment. This holistic approach to testing helps in identifying potential issues that might not be evident at the unit or integration level alone.

6.3 Introduction to Integration Testing

The failure of the Mars Climate Orbiter mission in September 1999 underscores the critical role of integration testing in software development. This incident resulted from a significant integration error: Lockheed Martin Astronautics used acceleration data in English units (pounds), while the Jet Propulsion Laboratory utilized metric units (newtons) for calculations. This discrepancy ultimately led to the loss of the spacecraft as it approached Mars. This issue highlight the importance of comprehensive integration testing to identify and rectify such issues.

Among the three key levels of software testing—**unit testing, integration testing,** and **system testing**—integration testing is often perceived as the least understood and inadequately executed phase in practice. This chapter emphasizes the importance of integration testing and explores various integration testing strategies.

6.3.1 What is 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.

6.3.2 Features (or) Characteristics (or) Importance of Integration Testing

Integration testing is characterized by several key features that distinguish it from other types of testing in the software development lifecycle. Some of the prominent characteristics of integration testing include:

1. **Interaction Testing:** Integration testing focuses on testing the interactions between different modules or components of a software system. It ensures that these components work together seamlessly and communicate effectively with each other.
2. **Interface Validation:** One of the main objectives of integration testing is to validate the interfaces between modules. This involves checking data exchange, method calls, and communication protocols to ensure compatibility and consistency.
3. **Dependency Management:** Integration testing helps in identifying and managing dependencies between modules. It ensures that changes in one module do not adversely impact the functionality of other interconnected modules.
4. **Incremental Approach:** Integration testing is often conducted incrementally, starting with testing individual units and gradually integrating them to form larger components. This incremental approach helps in detecting integration issues early in the development process.
5. **Types of Integration:** There are different types of integration testing approaches, such as top-down integration testing, bottom-up integration testing, and sandwich (hybrid) integration testing. Each type focuses on integrating components in a specific order to achieve comprehensive test coverage.

6. **Fault Detection:** Integration testing aims to detect faults that may arise due to the integration of different modules. By simulating real-world interactions, integration testing helps in uncovering defects that may not be apparent during unit testing.
7. **Automation:** Automation tools and techniques are often used in integration testing to streamline the testing process, execute test scenarios efficiently, and generate reports on integration test results.
8. **System Behavior Validation:** Integration testing validates the overall behavior and performance of the software system by testing the integrated components as a whole. It ensures that the system functions as expected and meets the specified requirements.

6.3.3 Types of Integration Testing

Integration testing is a critical phase in the software testing process that focuses on verifying the interactions and interfaces between different components or modules of a software system. There are several types of integration testing approaches that software development teams can be used to ensure the seamless integration and functionality of the system. Some common types of integration testing are:

1. **Decomposition-Based Integration Testing:**
   - **Top-Down Integration:** In top-down integration testing, testing begins with the highest-level modules or components and gradually progresses to lower-level modules. Stub functions are used to simulate the behavior of lower-level modules that have not yet been integrated.
   - **Bottom-Up Integration:** The bottom-up integration testing starts with the lowest-level modules and moves upwards by integrating higher-level modules incrementally. Drivers are used to simulate the behavior of higher-level modules that are yet to be integrated.
   - **Sandwich Integration:** Also known as hybrid integration testing, sandwich integration combines elements of both top-down and bottom-up approaches. It involves integrating modules both from the top down and from the bottom up simultaneously to ensure comprehensive testing coverage.
2. **Call Graph-Based Integration Testing:**
   - **Pairwise Integration:** Pairwise integration testing focuses on testing interactions between pairs of modules. This approach aims to uncover defects that may arise from the interaction of specific module pairs.
   - **Neighborhood Integration:** Neighborhood integration testing expands on pairwise testing by considering the interactions within a neighborhood or group of modules. It tests the interactions between modules within a specific proximity to identify integration issues.
3. **Path-Based Integration Testing:**
   Path-based integration testing involves testing the execution paths through integrated components to ensure that the system behaves as expected under different scenarios. This approach focuses on verifying the flow of data and control between modules along specific paths to validate system functionality and behavior.

These different types of integration testing approaches offer various strategies for testing the integration of software components, each with its own strengths and considerations. By using a combination of these integration testing techniques, software development teams can effectively validate the interactions and interfaces between modules, detect integration issues early, and ensure the overall reliability and performance of the integrated system.

6.4 Decomposition-Based Integration Testing

Decomposition-Based Integration Testing is a crucial phase in the software development lifecycle that focuses on verifying the correct integration of individual modules or components to ensure that the system functions as intended when all parts are combined. This testing approach aligns with the principles of Decomposition-Based Integration, where the system is built by integrating modular components based on functional decomposition.

Key aspects of Decomposition-Based Integration Testing include:

1. **Module Testing:** Before integration, each module is tested in isolation to validate its functionality and behavior. Unit tests are conducted to ensure that individual modules perform as expected according to their specifications.
2. **Integration Strategy Selection:** Based on the system architecture and design, an appropriate integration strategy (such as top-down, bottom-up, sandwich, or big bang) is chosen to combine modules in a systematic manner.
3. **Interface Testing:** During integration, special attention is given to testing the interfaces between modules to ensure that data and control flow correctly between components. Interface testing helps identify communication issues and compatibility problems.
4. **Data Flow Testing:** Data flow within the integrated system is tested to verify that information is passed correctly between modules and that data integrity is maintained throughout the system.
5. **Dependency Management:** Testing focuses on managing dependencies between modules to ensure that changes in one component do not adversely affect other parts of the system. Dependency testing helps in identifying and resolving issues related to inter-module interactions.
6. **Error Handling:** Integration testing includes scenarios to test error handling and recovery mechanisms when modules encounter unexpected situations or faults. This ensures that the system can gracefully handle errors and maintain stability.
7. **Regression Testing:** As modules are integrated and changes are made during the testing process, regression testing is performed to verify that existing functionalities remain unaffected and that new integrations do not introduce regressions.

By conducting thorough Decomposition-Based Integration Testing, software development teams can identify and address integration issues early in the development cycle, validate the system's overall functionality, and ensure that the integrated software meets the specified requirements and quality standards. This testing approach plays a critical role in delivering a reliable and robust software product to end-users.

6.4.1 Top- Down Integration Testing

Top-Down Integration Testing is a software testing approach where testing begins with the highest-level modules or components and gradually progresses towards lower-level modules. In this strategy, the focus is on integrating and testing the main control modules or components first, followed by the integration of subordinate modules. Stubs or simulated modules are used to stand in for lower-level modules that have not yet been developed or integrated.

How Top-Down Integration Testing Works:

- **Start with Main Module:** The testing process begins with the main or top-level module of the software system.
- **Subordinate Module Integration:** Once the main module is tested, integration proceeds to the next level of modules that are directly dependent on the main module.
- **Stubs Usage:** Stubs are used to simulate the behavior of lower-level modules that are not yet integrated. Stubs provide the necessary input and mimic the output of the missing modules.
- **Incremental Integration:** Integration is done incrementally, with each level of modules being added and tested in a step-by-step manner.
- **Testing Continues Downwards:** The integration and testing process continues downwards through the hierarchy of modules until all components are integrated and tested together.

Example **Top-Down Integration Testing**

Let's consider a simple example of a banking application with the following modules:

- **Main Module:** Account Management
- **Subordinate Modules:** Transaction Processing, Customer Information

1. **Main Module Testing:**
   - The Account Management module, responsible for overall account handling, is tested independently.
   - Functionality such as account creation, deletion, and updating is verified.
2. **Subordinate Module Integration:**
   - The Transaction Processing module, which handles deposit and withdrawal transactions, is integrated with the Account Management module.
   - Stubs are used to simulate the Customer Information module's behavior.
   - Transactions are processed and validated within the integrated system.
3. **Further Integration:**
   - The Customer Information module, which stores customer details, is integrated with the Account Management and Transaction Processing modules.
   - Stubs are replaced with actual modules as they become available.
   - Customer information retrieval and validation are tested in conjunction with account and transaction functionalities.
4. **Complete System Testing:**
   - Once all modules are integrated, end-to-end testing is performed to ensure that the entire banking application functions correctly.
   - Data flow, error handling, and system performance are evaluated in a holistic manner.

Advantages and Disadvantages of Top-Down Integration Testing

Advantages

- Early identification of high-level issues.
- Critical functionalities are integrated and tested first.
- Provides a structured approach to integration testing.

Disadvantages

- Dependency on lower-level modules that may not be ready.
- Potential delays in testing lower-level functionalities.

6.4.2 Bottom-Up Integration Testing

Bottom-Up Integration Testing is a software testing approach that starts with testing individual modules or components at the lowest level and gradually progresses towards higher-level modules or the complete system. In this strategy, lower-level modules are integrated and tested first, and then the focus shifts to integrating higher-level modules that depend on the already tested lower-level components. Drivers are used to simulate the behavior of higher-level modules that are not yet developed or integrated.

How Bottom-Up Integration Testing Works:

1. **Start with Lowest-Level Modules:** Testing begins with the individual modules at the lowest level of the software system.
2. **Higher-Level Module Integration:** Once the lower-level modules are tested, integration proceeds to the next level of modules that depend on the already tested components.
3. **Drivers Usage:** Drivers are used to simulate the behavior of higher-level modules that are not yet integrated. Drivers provide the necessary input and mimic the output of the missing modules.
4. **Incremental Integration:** Integration is done incrementally, with each level of modules being added and tested in a step-by-step manner.
5. **Testing Continues Upwards:** The integration and testing process continues upwards through the hierarchy of modules until all components are integrated and tested together.

Example **Bottom-Up Integration Testing**

Consider a simple example of an e-commerce application with the following modules:

- **Lowest-Level Modules:** Payment Processing, Inventory Management
- **Higher-Level Module:** Order Fulfillment

1. **Lowest-Level Module Testing:**
   - The Payment Processing module, responsible for handling payment transactions, is tested independently.
   - Functionality such as payment validation and processing is verified.

2. **Higher-Level Module Integration:**
   - The Inventory Management module, which tracks product availability, is integrated with the Payment Processing module.
   - Drivers are used to simulate the Order Fulfillment module's behavior.
   - Payment processing and inventory updates are tested in conjunction.
3. **Further Integration:**
   - The Order Fulfillment module, which manages order processing and shipment, is integrated with the Payment Processing and Inventory Management modules.
   - Drivers are replaced with actual modules as they become available.
   - Order processing and shipment handling functionalities are tested in the integrated system.
4. **Complete System Testing:**
   - Once all modules are integrated, end-to-end testing is performed to ensure that the entire e-commerce application functions correctly.
   - Data flow, error handling, and system performance are evaluated in a holistic manner.

Advantages and Disadvantages of Bottom-Up Integration Testing

Advantages

- Early identification of issues in lower-level modules.
- Critical functionalities are tested and validated from the ground up.
- Provides a structured approach to integration testing.

Disadvantages

- Dependency on higher-level modules that may not be ready.
- Potential delays in testing higher-level functionalities.

6.4.3 Sandwich Integration Testing

Sandwich Integration Testing is a hybrid approach that combines elements of both Top-Down and Bottom-Up Integration Testing strategies. In this method, testing starts simultaneously from the top (main control modules) and the bottom (individual modules) towards the middle layers of the software system. The idea is to integrate and test modules at both ends while gradually moving towards the center of the system, where the integration of components from both directions occurs.

How Sandwich Integration Testing Works:

- **Simultaneous Integration:** Testing begins by integrating and testing the main control modules at the top level and individual modules at the bottom level concurrently.
- **Middle Layer Integration:** As testing progresses, integration moves towards the middle layers of the system, where components from both the top and bottom are integrated.
- **Validation and Verification:** The interactions between the top-level and bottom-level modules are validated and verified in the middle layers.

- **Incremental Approach:** Integration is done incrementally, with a focus on ensuring that the interactions between different layers of the system work seamlessly.
- **Comprehensive Testing:** The goal is to achieve comprehensive testing coverage by combining the strengths of both Top-Down and Bottom-Up approaches.

Example **Sandwich Integration Testing**

Consider a social media platform with the following modules:

- **Top-Level Module:** User Interface and Social Feed
- **Bottom-Level Modules:** User Authentication, Post Management
- **Middle Layer:** Notification System

In Sandwich Integration Testing:

- User Interface and Social Feed are tested at the top level.
- User Authentication and Post Management are tested at the bottom level.
- The Notification System in the middle layer integrates components from both ends to ensure seamless notifications for user interactions.

Advantages and Disadvantages of Sandwich Integration Testing

Advantages

- Ensures thorough testing of the entire system by integrating components from both ends.
- Helps in identifying integration issues early in the testing process.
- Combines the benefits of Top-Down and Bottom-Up strategies for a more balanced integration testing process.
- Facilitates efficient integration of modules from different levels of the system.

Disadvantages

- Managing integration from both ends simultaneously can be complex and challenging.
- Ensuring that dependencies between top-level and bottom-level modules are handled effectively.
- Requires coordination and resources to conduct testing from multiple directions.

6.5 Call Graph-Based Integration

Call Graph-Based Integration Testing is a software testing approach that focuses on analyzing the call relationships between different modules or components in a software system to guide the integration testing process. A call graph is a graphical representation of how modules or functions call each other during program execution. By examining the call graph, testers can identify the flow of control and data between components, helping them prioritize integration testing efforts based on these dependencies.

The call graph is developed by considering units to be nodes, and if unit A calls (or uses) unit B, there is an edge from node A to node B. The call graph for the Calendar program is shown in below Figure.

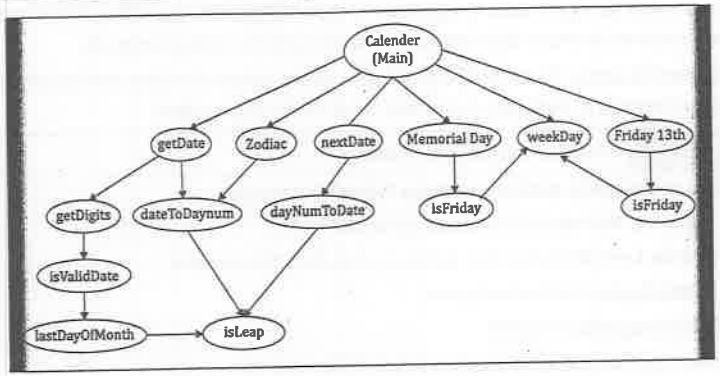


Fig 6.4 : Call Graph of Calender Program

Characteristics of Call Graph-Based Integration Testing:

- **Call Graph Generation:** The first step involves generating a call graph that illustrates the relationships between modules or functions in the software system.
- **Identifying Dependencies:** Testers analyze the call graph to identify dependencies between components, including function calls, data exchanges, and control flow.
- **Prioritizing Integration Testing:** Based on the call graph analysis, integration testing priorities are established to focus on critical paths and interactions between modules.
- **Testing Scenarios:** Test scenarios are designed to cover the interactions identified in the call graph, ensuring comprehensive integration testing.
- **Regression Testing:** Call graph-based testing helps in identifying potential regression issues when changes are made to the software system.

Advantages of Call Graph-Based Integration Testing:

- **Dependency Visualization:** Provides a clear visualization of dependencies between modules, aiding in understanding the software architecture.
- **Focused Testing:** Enables testers to focus on critical paths and interactions, improving the effectiveness of integration testing.
- **Early Issue Detection:** Helps in early detection of integration issues based on the identified dependencies.
- **Efficient Testing:** Guides testers in designing targeted test scenarios for thorough integration testing.

Challenges of Call Graph-Based Integration Testing:

- **Complexity:** Analyzing and interpreting large call graphs can be complex, especially in large software systems.
- **Dynamic Environments:** Call graphs may change dynamically based on runtime behavior, requiring continuous updates for accurate testing.

- **Tool Dependency:** Effective call graph generation and analysis may require specialized tools, adding to the testing process complexity.

Example **Call Graph-Based Integration**

Consider a web application with the following modules:

- User Authentication
- Data Processing
- Reporting Module

In Call Graph-Based Integration Testing:

- The call graph analysis reveals that the Data Processing module calls functions from User Authentication for user validation.
- Test scenarios are designed to validate the interactions between User Authentication and Data Processing based on the call graph.
- Integration testing focuses on ensuring seamless data flow and authentication processes between the modules.

6.5.1 Pair wise Integration Testing

Pairwise Integration Testing is a systematic testing technique that focuses on testing interactions between pairs of modules or components in a software system. The goal of pairwise integration testing is to identify and validate the interactions between different pairs of modules to ensure that the integrated system functions correctly. This approach helps in detecting integration issues that may arise due to the interactions between specific pairs of modules.

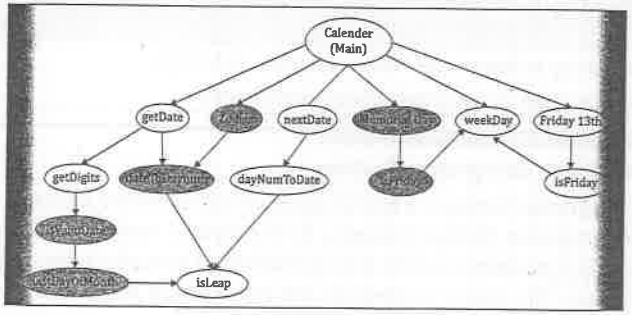


Fig 6.5 : Three Pairs for Pairwise Integration

Characteristics of Pairwise Integration Testing:

- **Pair Selection:** Modules are selected in pairs for integration testing based on their dependencies and interactions in the software system.
- **Testing Scenarios:** Test scenarios are designed to cover the interactions between each pair of modules, focusing on input-output relationships and data flow.
- **Isolation of Pairs:** Each pair of modules is tested in isolation to identify any integration issues specific to that pair.

- **Combinatorial Testing:** Pairwise testing aims to cover all possible combinations of interactions between pairs of modules to ensure comprehensive testing coverage.
- **Error Detection:** The testing process aims to detect errors related to the interactions between specific pairs of modules, helping in early issue identification.

How it works?

Pairwise Integration Testing works by systematically selecting pairs of modules or components in a software system and testing the interactions between these pairs to ensure the integrated system functions correctly.

1. **Pair Selection:**
   - Modules are selected in pairs based on their dependencies, interactions, and integration points within the software system.
   - The selection of pairs is typically guided by the understanding of the software architecture, module dependencies, and critical paths.
2. **Testing Scenarios Design:**
   - Test scenarios are designed to cover the interactions between each pair of modules selected for testing.
   - Scenarios focus on input-output relationships, data flow, error handling, and boundary conditions specific to the interactions between the selected pairs.
3. **Isolation of Pairs:**
   - Each pair of modules is tested in isolation to identify any integration issues that may arise due to the interactions between those specific modules.
   - Isolating the pairs helps in pinpointing integration issues and understanding the behavior of the integrated components.
4. **Combinatorial Testing:**
   - Pairwise testing aims to cover all possible combinations of interactions between pairs of modules while minimizing the number of test cases needed.
   - The goal is to achieve comprehensive testing coverage by testing interactions between pairs without testing every possible combination of modules.
5. **Execution and Analysis:**
   - Test cases designed for each pair of modules are executed to validate the interactions and integration points.
   - Test results are analyzed to identify integration issues, such as data inconsistencies, communication failures, or incorrect behavior resulting from the interactions between the pairs.
6. **Issue Identification and Resolution:**
   - Integration issues specific to certain pairs of modules are identified during testing.
   - Detected issues are documented, prioritized, and resolved to ensure the integrated system functions correctly.

7. **Iterative Process:**
   - Pairwise Integration Testing is often an iterative process where test scenarios are refined based on feedback and issues encountered during testing.
   - The process may involve retesting pairs of modules after issue resolution to validate the effectiveness of the fixes.

Example: Pairwise Integration Testing

Consider a banking application with the following modules:

- User Authentication
- Account Management
- Transaction Processing

In Pairwise Integration Testing:

- Test scenarios are designed to test interactions between pairs of modules, such as User Authentication and Account Management, Account Management and Transaction Processing, and User Authentication and Transaction Processing.
- Each pair of modules is tested to validate the data flow and interactions between them, focusing on input validation, data processing, and error handling.
- Integration testing efforts are concentrated on identifying and resolving issues specific to the interactions between these pairs of modules.

Pros and Cons of Pairwise Integration Testing

1. **Pros of Pairwise Integration Testing:**
   - **Efficient Coverage:** Pairwise testing provides comprehensive coverage by testing interactions between pairs of modules without the need to test all possible combinations, reducing the number of test cases required.
   - **Focused Testing:** Targets specific pairs of modules for integration testing, allowing for focused testing efforts on critical integration points and dependencies.
   - **Early Issue Identification:** Helps in early detection of integration issues specific to certain pairs of modules, enabling timely resolution and reducing the likelihood of more significant problems later in the development cycle.
   - **Reduced Testing Effort:** Achieves thorough testing coverage with fewer test cases compared to exhaustive testing approaches, saving time and resources.
   - **Optimized Testing:** Identifies integration issues related to specific pairs of modules, enabling targeted testing and efficient use of testing resources.
2. **Cons of Pairwise Integration Testing:**
   - **Dependency Identification:** Identifying the dependencies and interactions between modules accurately can be challenging, leading to potential gaps in test coverage.
   - **Test Case Design Complexity:** Designing effective test scenarios to cover all pairs of modules while avoiding redundant testing can be complex and time-consuming.

- **Dynamic Systems:** Testing interactions between pairs of modules in dynamic systems with frequent changes may require continuous updates to test cases, impacting testing efficiency.
- **Limited Scope:** Pairwise testing focuses on interactions between pairs of modules, which may not uncover issues that arise from interactions involving multiple modules simultaneously.
- **Maintenance Overhead:** As the software system evolves, maintaining pairwise test cases to reflect changes in module interactions can be challenging and may require ongoing effort.

6.5.2 Neighborhood Integration Testing

Neighborhood Integration Testing is a testing approach that focuses on testing the interactions and integration between a module under test and its neighboring modules or components within a software system. This testing technique aims to verify the communication, data exchange, and functionality between the module being tested and its immediate neighbors to ensure seamless integration and proper system behavior.

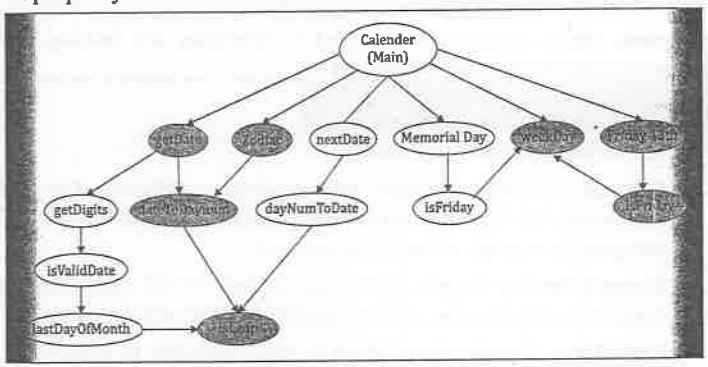


Fig 6.6 : Three neighborhoods (of radius1) for neighborhood integration

Characteristics of Neighborhood Integration Testing :

- **Proximity Testing:** Neighborhood Integration Testing targets modules that are in close proximity or have direct dependencies on each other within the software system.
- **Clustered Testing:** Modules are grouped into clusters or neighborhoods based on their interconnections and shared functionalities for focused integration testing.
- **Inter-Module Communication:** Testing scenarios are designed to verify the communication channels and data exchanges between neighboring modules.
- **Dependency Validation:** Emphasis is placed on validating dependencies and interactions between immediate neighbor modules to ensure seamless integration.
- **Functional Cohesion:** Testing focuses on the functional cohesion and collaboration between closely related modules to detect integration issues early.

How it works?

Neighborhood Integration Testing works by focusing on testing the interactions and integration between a specific module under test and its neighboring modules or components within a software system.

1. **Cluster Identification:**
   Identify clusters or neighborhoods of modules that have strong interdependencies or shared functionalities within the software system.
1. **Testing Scope Definition:**
   Define the scope of testing for each neighborhood, outlining the specific modules and interactions to be tested within the cluster.
3. **Scenario Design:**
   Design test scenarios that target the interactions between modules within each neighborhood, focusing on data flow, communication protocols, and shared functionalities.
4. **Integration Testing:**
   Conduct integration testing within each neighborhood to validate the interactions and integration points between neighboring modules.
5. **Boundary Testing:**
   Test boundary conditions and edge cases within the neighborhood to uncover potential integration issues related to data boundaries or exceptional scenarios.
6. **Error Handling:**
   Verify error handling mechanisms and exception scenarios between neighboring modules to ensure robust error recovery and system stability.
7. **Regression Testing:**
   Perform regression testing within each neighborhood after changes or updates to confirm that the integration between modules remains intact.
8. **Collaborative Issue Resolution:**
   Collaborate with developers and stakeholders to address integration issues identified within specific neighborhoods, focusing on enhancing communication and data exchange between modules.

Example **Neighborhood Integration Testing**

Let's consider the following modules for Neighborhood Integration Testing:

- **User Authentication Module:** Responsible for user login, authentication, and session management.
- **Product Catalog Module:** Manages the display of products, product information, and search functionalities.
- **Shopping Cart Module:** Handles adding/removing items to/from the cart, calculating totals, and managing the shopping cart state.

Example Scenario for Neighborhood Integration Testing:

1. **Cluster Identification:** The User Authentication Module, Product Catalog Module, and Shopping Cart Module are identified as a neighborhood cluster due to their interdependencies in the e-commerce platform.

2. **Testing Scope Definition:** Define the scope of testing to focus on the interactions between these three modules within the cluster.
3. **Scenario Design:**
   - **Test Scenario 1:** Verify that a logged-in user can add items from the Product Catalog to the Shopping Cart.
   - **Test Scenario 2:** Test the functionality to update the shopping cart total when items are added or removed.
   - **Test Scenario 3:** Validate that only authenticated users can access the shopping cart feature.
4. **Integration Testing:** Execute the test scenarios to ensure seamless communication and data flow between the User Authentication, Product Catalog, and Shopping Cart modules.
5. **Boundary Testing:** Test scenarios with edge cases such as adding a large number of items to the cart to check for performance and boundary issues.
6. **Error Handling:** Validate error handling mechanisms for scenarios like incorrect login credentials, out-of-stock products, or invalid cart operations.
7. **Regression Testing:** Perform regression testing within the neighborhood cluster after any updates or changes to confirm that the integration between modules remains intact.
8. **Collaborative Issue Resolution:** Collaborate with developers to address any integration issues identified during testing, focusing on improving the communication and data exchange between the User Authentication, Product Catalog, and Shopping Cart modules.

Pros and Cons of Neighborhood Integration Testing

1. **Pros of Neighborhood Integration Testing:**
   - **Focused Testing:** Neighborhood Integration Testing allows for focused testing on specific clusters of closely related modules, ensuring thorough testing of critical interactions within these clusters.
   - **Early Issue Detection:** By testing interactions between immediate neighbor modules early in the development cycle, integration issues can be identified and resolved promptly, reducing the likelihood of complex integration problems later on.
   - **Improved Collaboration:** Testing within neighborhood clusters encourages collaboration between developers working on interconnected modules, leading to better communication and understanding of integration requirements.
   4. **Efficient Testing:** By targeting specific clusters of modules, testing efforts can be optimized, leading to more efficient use of resources and time during the testing phase.
   5. **Enhanced Stability:** Validating interactions between closely related modules helps improve the stability and reliability of the software system by ensuring seamless integration within specific functional areas.
2. **Cons of Neighborhood Integration Testing:**
   1. **Limited Scope:** Neighborhood Integration Testing focuses on interactions within specific clusters, potentially overlooking integration issues that may arise between modules in different clusters or at a higher level of integration.

   2. **Dependency Risks:** If the dependencies between modules within a neighborhood are not accurately identified or tested, there is a risk of missing critical integration issues that may impact the overall system functionality.
   3. **Integration Gaps:** Testing within neighborhood clusters may create gaps in testing coverage between clusters, leading to potential integration issues at the boundaries of these clusters that are not adequately addressed.
   4. **Complexity Management:** Managing testing efforts for multiple neighborhood clusters can become complex, especially in large-scale software systems with numerous interconnected modules, requiring careful planning and coordination.
   5. **Maintenance Challenges:** As the software system evolves, maintaining neighborhood integration tests and ensuring they remain up-to-date with changes in module interactions can pose challenges, especially when dealing with frequent updates or modifications.

6.6 Path-Based Integration Testing

Path-Based Integration Testing is a testing approach that focuses on validating the flow of data and control between interconnected modules or components within a software system along specific paths or sequences. This testing technique aims to verify the correct execution of critical paths through the system, ensuring that data is processed accurately and control is transferred correctly between modules. Path-Based Integration Testing helps identify integration issues related to data flow, control flow, and the interaction between modules along predefined paths within the software system.

Characteristics of Path-Based Integration Testing:

1. **Path Identification:** Critical paths through the system are identified based on the expected flow of data and control between modules.
2. **Path Selection:** Specific paths are selected for testing based on their importance to system functionality, error-prone areas, or high-risk scenarios.
3. **Data Flow Validation:** Testing focuses on verifying the correct flow of data along the selected paths, ensuring that data is processed accurately and transferred between modules without loss or corruption.
4. **Control Flow Verification:** Validation of the control flow between modules to ensure that control is transferred correctly along the selected paths, following the expected sequence of operations.
5. **Boundary Condition Testing:** Testing scenarios include boundary conditions along the selected paths to uncover potential issues related to data boundaries, exceptional cases, or edge conditions.
6. **Error Handling Testing:** Validation of error handling mechanisms along the paths to ensure that exceptions are handled appropriately and the system maintains stability under error conditions.
7. **Integration Point Testing:** Testing at integration points along the paths to verify the interaction and communication between modules, including parameter passing, data exchange, and synchronization.

How Path-Based Integration Testing Works:

1. **Path Identification:**
   Identify critical paths through the system that represent key functionalities, user transactions, or data processing sequences.
2. **Path Selection:**
   Choose specific paths for testing based on factors such as complexity, risk, dependencies, and impact on system behavior.
3. **Test Scenario Design:**
   Design test scenarios that follow the selected paths, including input data, expected outcomes, and steps to be executed along the path.
4. **Path Execution:**
   Execute the test scenarios to validate the flow of data and control along the selected paths, monitoring the behavior of the system at integration points.
5. **Result Analysis:**
   Analyze test results to identify any deviations from expected behavior, including data discrepancies, control flow errors, or integration issues along the paths.
6. **Issue Resolution:**
   Collaborate with developers to address any integration issues identified during testing, focusing on improving data flow, control transfer, and interaction between modules along the critical paths.

Example **Path-Based Integration Testing**

Path-Based Integration Testing Scenario: Funds Transfer

1. **Path Identification:** The critical path for funds transfer involves the user logging in, selecting the transfer option, entering transfer details, verifying the transaction, processing the transfer, updating account balances, and sending a notification.
2. **Path Selection:** Select the funds transfer path for testing due to its importance in ensuring accurate financial transactions and data integrity.
3. **Test Scenario Design:**

   Scenario 1: Successful Funds Transfer

   User logs in with valid credentials.

   User selects the transfer option and enters transfer details.

   System verifies the transaction details.

   Funds transfer is processed successfully.

   Account balances are updated accordingly.

   Notification is sent to the user confirming the transfer.
4. **Path Execution:** Execute the test scenario to validate the flow of data and control along the funds transfer path, ensuring that each step functions correctly and data is processed accurately.
5. **Result Analysis:** Monitor the test execution to identify any deviations from the expected behavior, such as incorrect account balances, failed transactions, or missing notifications.
6. **Issue Resolution:** Collaborate with developers to address any integration issues discovered during testing, focusing on improving the interaction between modules involved in the funds transfer process

Pros and Cons of Path-Based Integration Testing

1. **Pros of Path-Based Integration Testing:**
   - **Focused Testing:** Path-Based Integration Testing allows for focused testing on critical paths through the system, ensuring thorough validation of key functionalities and user transactions.
   - **Comprehensive Coverage:** By targeting specific paths, this approach provides comprehensive coverage of important scenarios, including data flow, control flow, error handling, and integration points.
   - **Early Issue Detection:** Identifying and testing critical paths early in the development cycle helps in detecting integration issues promptly, reducing the likelihood of complex problems later on.
   - **Improved Quality:** Validating the flow of data and control along specific paths enhances the quality and reliability of the software system, ensuring that key functionalities work as intended.
   - **Efficient Testing:** Path-Based Integration Testing optimizes testing efforts by focusing on high-impact areas, leading to efficient use of resources and time during the testing phase.
2. **Cons of Path-Based Integration Testing:**
   - **Limited Scope:** This approach may overlook integration issues that occur outside the selected paths, potentially missing critical interactions between modules in different scenarios.
   - **Complexity Management:** Managing and testing multiple critical paths can become complex, especially in systems with intricate dependencies and interactions, requiring careful planning and coordination.
   - **Dependency Risks:** If dependencies between modules along the selected paths are not accurately identified or tested, there is a risk of missing integration issues that may impact overall system functionality.
   - **Maintenance Challenges:** Maintaining path-based test scenarios and ensuring they remain up-to-date with system changes can be challenging, especially in dynamic development environments with frequent updates.
   - **Risk of Oversimplification:** Focusing solely on critical paths may oversimplify testing, potentially missing edge cases, exceptional scenarios, or less common paths that could lead to integration issues.

6.7. Review Questions

Section - A

Each Question Carries Two Marks

1. What is Top- Down Integration Testing?
2. Mention the Advantages and Disadvantages of Top-Down Integration Testing.
3. What is Bottom-Up Integration Testing?

4. Mention the Advantages and Disadvantages of Bottom-Up Integration Testing.
5. What is Sandwich Integration Testing?
6. What is Call Graph-Based Integration?
7. Mention the types of Call Graph-Based Integration.
8. What is Pair wise Integration Testing?
9. What is Neighborhood Integration Testing?
10. What is Path-Based Integration Testing?

Section - B

Each Question Carries Five Marks

1. Explain Structural and Behavioural Insights of Simple ATM Problem.
2. Explain the Features and Importance of Integration Testing.
3. How Top- Down Integration Testing Works? Explain with an example.
4. How Bottom-Up Integration Testing Works? Explain with an example.
5. How Sandwich Integration Testing Works? Explain with an example.
6. What is Call Graph-Based Integration? Mention its Characteristics.
7. What is Pair wise Integration Testing? Mention its Characteristics.
8. How Pair wise Integration Testing Works?
9. Write the Pros and Cons of Pairwise Integration Testing.
10. What is Neighborhood Integration Testing? Mention its Characteristics.
11. How Neighborhood Integration Testing Works?
12. Write the Pros and Cons of Neighborhood Integration Testing.
13. What is Path-Based Integration Testing? Mention its Characteristics.
14. How Path-Based Integration Testing Works?
15. Write the Pros and Cons of Path-Based Integration Testing.

Section - C

Each Question Carries Eight Marks

1. Explain Levels of Testing in Different Life Cycle Models.
2. Explain the Testing Strategy for Simple ATM System (SATM).
3. Explain the Types of Integration Testing with examples.
4. Explain the types of Decomposition-Based Integration Testing with examples.
5. Discuss Call Graph-Based Integration and Write its characteristics, advantages and disadvantages.
6. Explain the types of Call Graph-Based Integration with examples.
7. Eloborate Path-Based Integration Testing with an example.

❖❖❖❖❖

UNIT - III

CHAPTER 7

SYSTEM TESTING

Contents

7.1 Introduction to System Testing

System testing is like trying out a used car or an online service to ensure it meets our expectations, rather than just finding faults. It focuses on meeting requirements rather than checking the code itself. However, because it's informal and often rushed due to deadlines, it may lack the formality it should have. To improve, we can think of testers as skilled craftsmen with deep knowledge.

In system testing, we can use **Atomic System Functions (ASFs)** to analyze how the system behaves. This leads to the concept of "threads," which are sequences of operations in the system. By identifying and testing these threads, we can find challenges and enhance our testing approach.

System testing is strongly connected to the requirements of the system. By utilizing system-level models and identifying system-level threads within these models, we can develop a systematic testing approach centered around these threads. This method integrates both requirements-based testing and code-based testing, leading to improved testing outcomes.

To demonstrate this approach, we can apply it to a simple automated teller machine (SATM) system, as discussed earlier. This practical example shows how a thread-based system testing method can be valuable and implemented effectively.

7.1.1 What is System Testing?

? 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.

7.1.2 Objectives of System Testing

The main objectives of system testing are:

- To identify defects in the system.
- To verify that the system meets the specified requirements.
- To ensure that the system is usable and reliable.
- To improve the system's performance.
- To validate the system's security measures.
- To assess the system's scalability and robustness.
- To confirm the system's compatibility with external interfaces and systems.
- To validate the system's compliance with regulatory standards and industry best practices.
- To assess the system's maintainability and ease of support.
- To ensure that the system can recover effectively from failures and errors.

7.1.3 Features or Characterisers of System Testing

System testing is a comprehensive evaluation of a fully integrated software product. It verifies that the system meets specified requirements and functions correctly in real-world scenarios. The key features of system testing are:

1. **End-to-End Testing:**
   System testing examines the entire application, including interactions between different components and subsystems. It validates the software's behavior from start to finish, ensuring that all parts work together seamlessly.
2. **Requirement Verification:**
   This testing phase ensures that the software meets all specified requirements. Test cases are derived from requirements documentation, making sure that every feature and function performs as expected.
3. **User Perspective:**
   System testing is conducted from the end-user's point of view. It focuses on the user's experience and validates that the system behaves as intended in real-world usage scenarios.
4. **Environment Simulation:**
   The testing is performed in an environment that closely resembles the production environment. This includes hardware, software, network configurations, and other system dependencies to ensure accurate results.
5. **Non-Functional Testing:**
   System testing includes non-functional aspects such as performance, security, usability, and reliability. It ensures that the system not only performs correctly but also meets performance criteria and security standards.
6. **Regression Testing:**
   System testing involves regression testing to ensure that new changes or additions do not negatively impact the existing functionality. It verifies that previously developed and tested software still performs correctly after modifications.
7. **Acceptance Criteria Validation:**
   The tests are designed to validate that the system meets the acceptance criteria defined by stakeholders or customers. This ensures that the software is ready for delivery and satisfies customer needs.
8. **Defect Detection:**
   While the primary goal is to validate correct behavior, system testing also aims to uncover defects that might not have been detected in earlier testing stages. These defects could include functional errors, performance issues, and integration problems.
9. **Documentation and Reporting:**
   Detailed documentation of test cases, test execution, and results is a critical aspect of system testing. This helps in tracking the testing process, identifying defects, and ensuring transparency for stakeholders.

10. **Simulating Real-World Scenarios:**
    Test cases are designed to simulate real-world scenarios, including edge cases and unusual conditions. This helps in ensuring that the system can handle unexpected situations gracefully.
11. **Automated and Manual Testing:**
    System testing often involves a combination of automated and manual testing techniques. Automated tests are used for repetitive tasks and large-scale test execution, while manual testing is applied to complex scenarios that require human judgment.

7.2 Atomic System Function (ASF)

The concept of an **Atomic System Function (ASF)** is crucial for effective system testing. It provides a granular approach to understanding and validating the individual components of a system.

? What is Atomic System Function (ASF) ?

An Atomic System Function (ASF) is the smallest unit of functionality within a system that can be independently tested. Each ASF represents a single, indivisible operation that the system can perform. These operations contribute directly to the overall behavior and functionality of the system. The ASFs are self-contained, meaning they do not need to be decomposed further for the purpose of testing.

7.2.1 Importance of Atomic System Functions (ASFs)

The concept of Atomic System Functions (ASFs) plays a vital role in the system testing process.

- **Enhanced Test Coverage:** ASFs ensure that every smallest unit of functionality is tested, which leads to comprehensive test coverage. This thorough examination helps in identifying and addressing even minor issues.
- **Fault Isolation:** Testing ASFs individually allows for precise fault isolation. When an issue arises, it can be quickly traced back to the specific ASF, making debugging and resolution more efficient.
- **Incremental Testing and Integration:** ASFs facilitate incremental testing, where new functionalities can be tested as they are developed. This ensures smooth integration and continuous validation of the system's evolving features.
- **Foundation for Automation:** The granularity and specificity of ASFs make them ideal candidates for automated testing. Automating ASF tests ensures consistent and repeatable testing processes, enhancing efficiency and reliability.
- **Improved Quality Assurance:** By validating each ASF, testers can ensure that all individual components function correctly. This leads to higher overall system quality, as each part has been rigorously tested.
- **Risk Mitigation:** Testing at the ASF level helps in early detection of defects, reducing the risk of major issues during later stages of development or after deployment. This proactive approach minimizes potential system failures.
- **Supports Agile Practices:** ASFs align well with agile methodologies, where small, incremental changes are made frequently. Testing ASFs ensures that each iteration is thoroughly validated, supporting the agile process of continuous delivery and improvement.

- **Scalability:** ASFs provide a scalable testing strategy that can be applied to systems of varying sizes and complexities. This flexibility makes the ASF approach suitable for both small projects and large, complex systems.

7.2.2 Characteristics of Atomic System Functions (ASFs)

ASFs have distinct characteristics that define their role and functionality in system testing. The the key characteristics are :

- **Granularity:** ASFs are the smallest units of functionality within a system. They represent single, indivisible operations that can be independently tested.
- **Independence:** Each ASF is designed to operate independently, without relying on other functions. This independence allows for isolated testing and precise fault identification.
- **Specificity:** ASFs focus on specific actions or functions within the system. This specificity ensures that each ASF has a well-defined scope and purpose, making it easier to test and validate.
- **Self-Contained:** ASFs are self-contained, meaning they encompass all the necessary logic and data to perform their function. This self-containment simplifies testing and reduces dependencies.
- **Reusability:** Due to their well-defined and specific nature, ASFs can often be reused across different parts of the system or in different projects. This reusability enhances development efficiency and consistency.
- **Clarity:** ASFs have clear, concise definitions and purposes. This clarity helps in creating precise test cases and simplifies the testing process.
- **Modularity:** ASFs contribute to the modularity of the system. Each ASF represents a distinct module that can be tested, maintained, and updated independently, supporting a modular architecture.
- **Testability:** ASFs are designed with testability in mind. Their granular and specific nature makes them straightforward to test, whether manually or through automated tests.
- **Documentation:** Each ASF can be documented independently, providing detailed information about its functionality, inputs, outputs, and expected behavior. This documentation aids in understanding and testing the ASF.
- **Interoperability:** While ASFs are independent, they are designed to interact seamlessly with other ASFs. This interoperability ensures that individual functions can be combined to form higher-level operations and workflows.

Example **Understanding ASF**

To illustrate the concept of an Atomic System Function (ASF) in a simple and straightforward manner, let's consider a basic operation in an ATM system: Withdrawing Cash.

Higher-Level Function: Withdraw Cash

The "Withdraw Cash" function involves several steps, each of which can be considered an ASF.

List of Interconnected ASFs:

1. Insert Card
2. Enter PIN
3. Select Withdrawal Amount
4. Verify Account Balance
5. Dispense Cash

Let's focus on one specific ASF: **"Enter PIN".**

Atomic System Function: Enter PIN

This ASF handles the user input of their Personal Identification Number (PIN) and verifies its correctness.

Characteristics

- **Granularity:** This ASF is a specific, small unit of functionality that performs a single operation—taking and verifying the user's PIN.
- **Independence:** The PIN entry process can be tested independently of other ASFs. It does not require the cash dispensing process to function correctly for testing purposes.
- **Specificity:** It has a clear, concise purpose—accepting and verifying the PIN entered by the user.
- **Self-Contained:** This function includes all the logic needed to accept user input and verify the PIN against stored data.

Test Scenarios : To test the "Enter PIN" ASF, we would design various test scenarios:

- **Correct PIN**

  Precondition: The stored PIN is 1234.

  Test Input: User enters 1234.

  Expected Outcome: PIN is verified successfully, and the system proceeds to the next step.

- **Incorrect PIN**

  Precondition: The stored PIN is 1234.

  Test Input: User enters 4321.

  Expected Outcome: PIN verification fails, and the system prompts the user to re-enter the PIN.

- **Maximum Attempts Exceeded**

  Precondition: The stored PIN is 1234, and the user has already entered the incorrect PIN twice.

  Test Input: User enters 4321.

  Expected Outcome: PIN verification fails, the system locks the card, and the user is prompted to contact their bank.

By focusing on the "Enter PIN" ASF in an ATM system, this short example illustrates how ASFs provide a structured approach to testing individual units of functionality within a system. This ensures that each component works correctly on its own before being integrated into the overall system.

7.3 Concept of Threads in System Testing

It is important to distinguish that a thread in the context of system testing is different from a thread in a programming language. In programming, a thread is a unit of execution within a process, often used for concurrent operations. However, in system testing, a thread represents a sequence of Atomic System Functions (ASFs) that together achieve higher-level functions or user goals.

The primary purpose of threads in system testing is to validate that the entire system supports and correctly executes complete user workflows. This ensures that users can achieve their goals without encountering issues to provide a smooth and reliable experience.

? What is a Thread in System Testing ?

A thread in system testing is a sequence of interconnected Atomic System Functions (ASFs) that together accomplish a specific user task or workflow in a system. It include all the steps a user takes to achieve a particular outcome to ensure that the system correctly executes the complete sequence of operations required.

Example Example of a Thread in an ATM System

Scenario : A customer wants to withdraw cash from an ATM.

User Goal : To successfully withdraw a specified amount of cash from their bank account using the ATM.

Thread Name : Withdraw Cash

List of ASFs :

1. **Insert Card :** User inserts their ATM card into the machine.
2. **Enter PIN :** User enters their Personal Identification Number (PIN) to authenticate.
3. **Select Withdrawal Amount :** User selects the amount of money they wish to withdraw from the displayed options or enters a custom amount.
4. **Verify Account Balance :** The system checks the user's account to ensure there are sufficient funds for the requested withdrawal.
5. **Dispense Cash :** The ATM dispenses the requested amount of cash to the user.
6. **Print Receipt :** The ATM prints a receipt detailing the transaction, including the amount withdrawn and the remaining balance.
7. **Eject Card :** The ATM ejects the user's card, completing the transaction.

By testing this thread, we ensure that all the steps necessary for withdrawing cash from an ATM work together seamlessly. This comprehensive approach validates the ATM's ability to support the complete withdrawal workflow from start to finish.

7.3.1 Objectives of a Thread

1. **User Workflow Validation :** Ensure that the system correctly supports and executes complete user workflows.

   Example : In an ATM system, this involves confirming that a user can successfully withdraw cash from start to finish. The goal is to ensure that all steps in the process work together without any issues, providing a smooth and seamless experience.

2. **Integration Assurance :** Verify that various Atomic System Functions (ASFs) integrate and interact correctly.

   Example: The transition from entering a PIN to selecting a withdrawal amount should be seamless to ensure that integrated components do not cause any disruptions or errors.

3. **Real-World Simulation :** Simulate real-world usage scenarios to identify potential issues that may arise during actual use.

Example : In an ATM system, simulating different withdrawal amounts and account balances helps ensure the system performs correctly under various conditions.

4. **Comprehensive Coverage :** Achieve thorough test coverage by examining not only individual ASFs but also their interactions and dependencies.

   Example : Testing both the card insertion process and how it leads to PIN entry ensures that no part of the workflow is neglected.

5. **Early Defect Detection :** Detect defects that may not be apparent when testing ASFs in isolation, particularly those that result from interactions between different parts of the system.

   Example : Issues that only occur when transitioning from balance verification to cash dispensing can be identified and resolved before the system is fully deployed.

6. **Performance Validation :** Ensure that the system performs efficiently under various conditions.

   Example : An ATM should dispense cash promptly and handle multiple transactions in a short period without slowing down or failing.

7. **Usability Verification :** Validate that the system is user-friendly and intuitive.

   Example : Confirming that the ATM's interface is straightforward and that users can understand prompts and complete transactions without confusion.

8. **Security Assurance :** Verify that the system maintains security and privacy standards throughout the workflow.

   Example : An ATM should ensure that PINs are encrypted and that users cannot access others' account information.

9. **Compliance Check :** Ensure that the system adheres to relevant standards, regulations, and requirements.

   Example : An ATM must comply with banking regulations and standards for electronic transactions.

7.3.2 Characteristics and Importance of a Thread

1. **Sequence of Operations :** It includes a specific order of operations that users follow to achieve a goal. This helps to replicate the exact way users would interact with the system in real-world scenarios. For example, in an ATM system, the steps include card insertion, PIN entry, amount selection, balance verification, cash dispensing, receipt printing, and card ejection. Testing the sequence ensures that transitions between each step are seamless and function as expected.
2. **User-Centric :** The thread is designed around the user's perspective and interactions with the system. The focus is on the user's journey through the system to ensure that their experience is smooth and intuitive. This involves considering how users interact with the system, what actions they take, and what feedback they receive at each step. The thread should cover all actions a user takes to achieve a goal, such as withdrawing cash from an ATM, ensuring each step is user-friendly and meets user expectations.

3. **Integration :** It tests the integration and interaction of different ASFs to ensure they work together seamlessly. Integration testing within a thread verifies that different parts of the system, or ASFs, interact correctly. It ensures that data flows accurately between components and that there are no interruptions or errors during transitions. For example, after entering the PIN, the system should correctly transition to balance verification and then to cash dispensing, ensuring that integrated components do not cause disruptions.
4. **End-to-End Testing :** The thread validates the complete end-to-end process from start to finish. End-to-end testing ensures that all steps in a user workflow are tested comprehensively. This type of testing covers the entire process that a user would go through to achieve a goal, from the beginning to the end. For instance, the thread for cash withdrawal must test everything from card insertion to card ejection, covering all intermediate steps to ensure the entire process functions as expected and delivers the intended outcome.
5. **Real-World Simulation :** Threads simulate actual user scenarios.This characteristic ensures that the testing environment closely mimics real-world usage, including typical user behaviors and edge cases. By simulating how users interact with the system in real life, including varying scenarios such as different withdrawal amounts or incorrect PIN entries, the thread helps ensure that the system can handle real-world conditions effectively. This enhances the reliability and robustness of the system in actual use.
6. **Fault Detection and Isolation :** Threads help detect and isolate faults. By testing sequences of operations, threads can identify where in the process a fault occurs, making it easier to pinpoint and resolve issues. For example, if cash is not dispensed, the thread can help determine if the fault lies in balance verification, the cash dispenser, or another part of the process. This facilitates quick troubleshooting and resolution, ensuring that the system can be quickly restored to proper functioning.
7. **Dependency Testing :** Threads account for dependencies between different ASFs. They ensure that dependent functions interact correctly and that data flows accurately between them. For example, verifying that the system checks the balance correctly after the user selects the withdrawal amount ensures that all dependencies are managed properly.

7.3.3 Types of Threads in System Testing

System testing threads can be categorized based on the level of complexity and scope they cover. The primary types include unit-level threads, integration-level threads, and system-level threads. Each type serves a different purpose in the software testing lifecycle.

1. **Unit-Level Threads :** Unit-level threads focus on the smallest testable parts of a system, typically individual functions or methods within a module.

Examples	Unit Level Threads
1. **Simple Arithmetic Function:** • **Function :** add(a, b) • **Thread :** Test the function with different pairs of numbers to ensure it returns the correct sum. • **Test Cases :** add(2, 3), add(-1, 5), add(0, 0)	

2. **Validation Function:**
   - **Function :** isValidEmail(email)
   - **Thread :** Check the function with different email formats to ensure proper validation.
   - **Test Cases :**
     isValidEmail("skyward.publishers@gamail.com")->true
     isValidEmail("ss@gmail") -> false

2. **Integration-Level Threads :** Integration-level threads focus on testing the interactions between different modules or components to ensure they work together correctly.

Examples **Integration Level Threads**

1. **Database Interaction**
   - **Scenario :** Test the interaction between a data access module and the database.
   - **Thread :** Verify that data is correctly retrieved, inserted, updated, and deleted.
   - **Test Cases :** Retrieve user data, insert a new record, update an existing record, delete a record.
2. **API and Service Integration:**
   - **Scenario :** Test the integration between a front-end application and a back-end API.
   - **Thread :** Ensure that the front-end correctly sends requests and handles responses from the API.
   - **Test Cases :** Login API call, data retrieval API call, data submission API call.

3. **System-Level Threads :** System-level threads involve testing the complete and integrated system to ensure it meets the specified requirements. These threads simulate real-world usage scenarios.

Example **System Level Threads**

1. **E-commerce Checkout Process**
   - **Scenario :** Test the entire checkout process in an e-commerce application.
   - **Thread :** Simulate a user selecting items, adding them to the cart, entering shipping information, making a payment, and receiving an order confirmation.
   - **Test Cases :** Add items to cart, proceed to checkout, enter shipping details, select payment method, complete payment, verify order confirmation.
2. **User Registration and Authentication**
   - **Scenario :** Test the user registration and login process in a web application.
   - **Thread :** Ensure that a user can register, verify their email, log in, and access their account dashboard.
   - **Test Cases :** Register with valid details, receive and click email verification link, log in with correct credentials, access dashboard.
3. **ATM Transaction Workflow :**
   - **Scenario :** Test the complete workflow of an ATM transaction.
   - **Thread :** Simulate a user inserting a card, entering a PIN, selecting a transaction type, performing the transaction, and retrieving the card.
   - **Test Cases :** Insert card, enter PIN, select withdrawal, check balance, dispense cash, print receipt, eject card.

Different types of threads serve distinct purposes in the software testing lifecycle. Unit-level threads focus on individual functions or methods, integration-level threads test the interactions between components, and system-level threads validate the entire system's functionality. By comprehensively testing across these levels, we can ensure that each part of the system works correctly on its own and in conjunction with other parts, ultimately leading to a robust and reliable system.

7.3.4 Thread Possibilities in the SATM System

Defining the endpoints of a system-level thread can be complex, but a graph theory–based approach can help provide a structured definition. In graph theory, a thread can be viewed as a path connecting a series of nodes (representing ASFs) with clearly defined start and end points. By mapping out these nodes and the transitions between them, we can visualize and analyze the complete user journey through the system. This backward approach from the desired outcomes ensures that all critical interactions and dependencies are captured. Using this method, we can identify four candidate threads in the Simple Automated Teller Machine (SATM) system:

1. Entry of a Digit
2. Entry of a Personal Identification Number (PIN)**
3. A Simple Transaction
4. An ATM Session Containing Multiple Transactions

Candidate Threads in SATM System

1. **Entry of a Digit :** This is the simplest and most granular thread, representing a minimal Atomic System Function (ASF).

   Sequence:

   1. **Port Input Event :** Digit keystroke.
   2. **Port Output Event :** Screen digit echo.

   It qualifies as a stimulus/response pair, making it suitable for integration testing. However, this level of granularity is also fine for system testing purposes as it does not include a complete user interaction.

2. **Entry of a Personal Identification Number (PIN) :** This thread represents a common ASF and serves as a bridge between integration testing and system testing.

   Sequence:

   1. **Screen Request :** A prompt requesting PIN digits.
   2. **User Input :** An interleaved sequence of digit keystrokes.
   3. **System Feedback :** Screen responses echoing the digits.
   4. **Cancellation Option :** The possibility for the user to cancel the operation before completing the PIN entry.
   5. **System Disposition:**
      - **Correct PIN :** After a correct PIN is entered within three attempts, the system displays a screen requesting the transaction type.

- **Incorrect PIN :** If the PIN is incorrect after three attempts, the system advises the customer that the ATM card will not be returned, and access to ATM functions is denied.

This thread exemplifies a family of stimulus/response pairs initiated by a port input event, traversing programmed logic, and terminating in several possible responses (port output events). It include a sequence of system-level inputs and outputs, marking the starting point of system testing.

3. **A Simple Transaction :** This thread represents a complete, end-to-end user transaction commonly executed by customers.

   Sequence:

   1. **Card Entry :** User inserts ATM card.
   2. **PIN Entry :** User enters their PIN.
   3. **Transaction Selection :** User selects the transaction type (e.g., deposit or withdrawal).
   4. **Account Details :** User provides account details (checking or savings, amount).
   5. **Conduct Operation :** The system processes the transaction.
   6. **Report Results :** The system reports the transaction results and returns the card.

   This is a good example of a system-level thread involving the interaction of several ASFs. It ensures the end-to-end completion of a transaction, making it an ideal candidate for system testing.

4. **An ATM Session Containing Multiple Transactions :** This thread represents a complex scenario where a user performs multiple transactions within a single session.

   Sequence:

   1. **Card Entry :** User inserts ATM card.
   2. **PIN Entry :** User enters their PIN.
   3. **First Transaction :** User performs an initial transaction (e.g., withdrawal).
   4. **Additional Transactions :** User performs one or more additional transactions (e.g., deposit, balance inquiry).
   5. **End Session :** User ends the session, and the card is returned.

   This thread is a sequence of multiple threads and is part of system testing. At this level, the focus is on the interactions among different threads. It ensures that the system can handle multiple transactions in a single session, maintaining state and functionality throughout.

Summary

In summary, each of these candidate threads represents different levels of complexity and granularity in system testing:

1. **Entry of a Digit :** Too granular for system testing, suitable for integration testing.
2. **Entry of a PIN :** Upper limit of integration testing and the starting point of system testing, involving multiple stimulus/response pairs.

3. **A Simple Transaction :** An end-to-end system-level thread, commonly executed by customers, involving multiple ASFs.
4. **An ATM Session with Multiple Transactions :** A sequence of threads, focusing on interactions among threads, representing a higher complexity level in system testing.

By understanding and testing these threads, we can ensure that the SATM system is robust, reliable, and user-friendly, capable of handling various user interactions and scenarios.

7.4 Basic Concepts for Requirements Specification

Instead of anticipating all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs: data, actions, devices, events, and threads. Every system can be modeled in terms of these five fundamental concepts (and every requirements specification model uses some combination of these). Each concept plays a crucial role in defining how a system operates and how it can be tested effectively. We examine these fundamental concepts here to see how they support the tester's process of thread identification.

1. **Data :** Data in a system refers to the information used and created by the system. It includes variables, data structures, fields, records, data stores, and files. Data models, such as Entity/Relationship (E/R) models, describe the relationships between different data entities. Data can be created, retrieved, updated, or deleted (CRUD operations).

   Example : In the SATM system, data includes account information, such as Account Number, Account Type, PINs, and account balances. For instance, each account might have a data structure with the account type, Expected PIN, and current balance. As ATM transactions occur, the system updates this data.

2. **Actions :** Actions are the operations performed by the system, often described as transforms, processes, activities, tasks, methods, or services. Actions have inputs and outputs, which can be data or events. They can be decomposed into lower-level actions, particularly in methodologies like Structured Analysis with data flow diagrams.

   Example : An action in the SATM system could be the process of verifying a PIN. This action takes the entered PIN as input, compares it with the stored PIN, and produces an output indicating whether the PIN is correct or incorrect.

3. **Devices :** Devices, or ports, are the sources and destinations of system-level inputs and outputs. Ports include physical interfaces like keyboards, screens, and card readers, which translate physical actions (like keystrokes) into logical events and vice versa.

   Example : The SATM system includes devices such as the keypad for entering PINs, the display screen for showing prompts and transaction results, the card reader for reading ATM cards, and the cash dispenser for dispensing money.

4. **Events :** Events are system-level inputs or outputs that occur at port devices. Events can be discrete (like keystrokes) or continuous (like temperature readings). Events translate real-world actions into logical actions within the system and vice versa.

Example : In the SATM system, pressing a key on the keypad generates an event. For instance, pressing the "1" key sends a digit entry event to the system, which then processes this input to display the digit on the screen or use it as part of the PIN.

5. **Threads :** Threads represent sequences of actions and events that achieve a specific user goal. They describe the interaction among data, events, and actions within the system. Threads are crucial for testing complete workflows from the user's perspective.

   Example : A thread in the SATM system could be the process of withdrawing cash:

   1. Insert ATM card (Device: Card Reader, Event: Card Inserted).
   2. Enter PIN (Device: Keypad, Event: PIN Entry).
   3. Select withdrawal transaction (Action: Transaction Selection).
   4. Enter amount to withdraw (Device: Keypad, Event: Amount Entry).
   5. Verify account balance (Action: Balance Verification).
   6. Dispense cash (Device: Cash Dispenser, Event: Cash Dispensed).
   7. Print receipt (Device: Printer, Event: Receipt Printed).
   8. Eject card (Device: Card Reader, Event: Card Ejected).

Relationships among Basis Concepts

The basis concepts are interconnected through many-to-many relationships. Data and events serve as inputs and outputs for actions. Events occur on devices, and many events can happen on a single device. Actions can belong to multiple threads, and threads are composed of several actions. This complex interplay requires thorough testing to ensure all relationships function correctly.

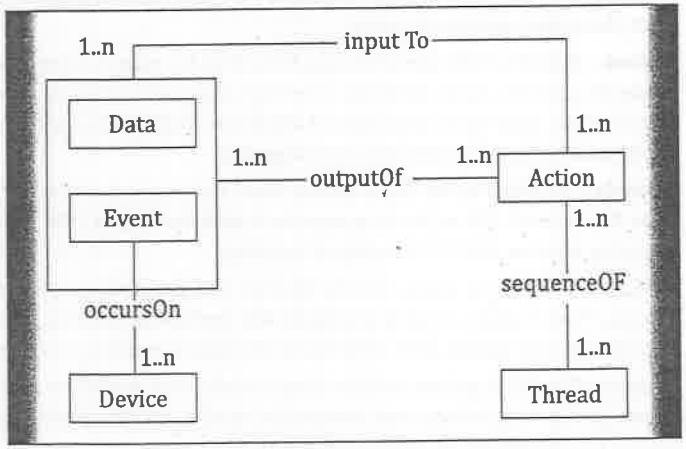


Figure 7.1 : E/R Model of Basis Concepts

Example : In the SATM system:

1. **Data :** Account details and transaction records.
2. **Events :** Card insertion, PIN entry, button presses.

3. **Actions :** Verify PIN, check balance, dispense cash.
4. **Devices :** Keypad, screen, card reader, cash dispenser.
5. **Threads :** Complete workflows like cash withdrawal and balance inquiry.

By understanding these basis concepts and their relationships, testers can effectively identify and validate threads, ensuring that all aspects of the system work together seamlessly. This comprehensive approach helps in identifying potential issues and verifying that the system meets its requirements.

7.5 Finding Threads in System Testing

In system testing, threads are sequences of actions and events that represent user interactions with the system to achieve specific goals. Defining and identifying these threads is crucial for comprehensive system testing. A graph theory-based approach can be used to define these threads by mapping out the states and transitions within the system. Finite state machines (FSMs) are a powerful tool for modeling these threads, as they capture the stages of processing and the transitions between them caused by various events.

7.5.1 Core Concepts for Finding Threads

1. **Finite State Machines (FSMs) :** FSMs consist of states and transitions:
   - **States:** Represent different stages in a process, each reflecting a specific condition or situation within the system. For example, in an ATM system, states can include "Card Entry," "PIN Entry," and "Transaction Processing."
   - **Transitions:** These are the changes from one state to another, triggered by events such as user inputs or system conditions. In the ATM example, a transition occurs when a user inserts a card, moving from the "Idle" state to the "Card Entry" state.

   FSMs help visualize the flow of operations, making it easier to identify and define threads. By mapping out these states and transitions, testers can see the entire process a user might go through and ensure that all possible paths are tested.

2. **Threads :** Threads are sequences of system actions and events that a user follows to accomplish a specific task:
   - In FSMs, threads correspond to paths through the state machine from an initial state to a final state.
   - A thread ensures that all intermediate steps and transitions are considered. It provides a comprehensive view of the user journey.

   Example : In an ATM system, a thread might involve:
   - Card Entry (initial state)
   - PIN Entry
   - Transaction Selection
   - Transaction Execution
   - Receipt Printing
   - Card Ejection (final state)

   Each step involves transitions that must be tested to ensure the system functions correctly.

3. **Port Events or Device Events :** Port events or Device events are interactions between the user and the system:
   - These include actions like keystrokes, screen displays, card insertions, and cash dispensing.
   - Port events are crucial for defining transitions in FSMs because they trigger state changes.

   Example : Pressing a key on the ATM's keypad is a port event that might transition the system from the "PIN Entry" state to the "Transaction Selection" state if the entered PIN is correct.
4. **Hierarchy of State Machines :** Hierarchy of FSMs involves nesting FSMs to manage complexity:
   - High-level FSMs represent broader states, while lower-level FSMs provide detailed breakdowns of these states.
   - This hierarchical approach helps in handling complex systems by breaking them down into manageable components.

   Example : In an ATM system:
   - The top-level FSM might include states like "Card Entry," "PIN Entry," and "Transaction Processing."
   - The "Transaction Processing" state can have a nested FSM detailing the steps for different transactions like "Withdrawal," "Deposit," and "Balance Inquiry."

   This hierarchical structure ensures that each aspect of the system is thoroughly tested at different levels of detail.

7.5.2 General Procedure for Finding Threads in System Testing

Finding threads in system testing involves the following steps:

1. **Define the Scope of the System:** Identify the main functionalities and boundaries of the system to be tested.
2. **Identify Key User Interactions:** Determine the primary tasks users will perform with the system. These tasks will help in identifying the main threads.
3. **Model the System with Finite State Machines (FSMs):** Create FSMs to represent the system states and transitions based on user interactions and system responses.
   - **States:** Different stages or conditions of the system.
   - **Transitions:** Events or actions that cause the system to change from one state to another.
4. **Identify Port Events:** Determine the input and output events at the system's ports. Port events are crucial for triggering state transitions.
   - **Port Input Events:** User inputs like keystrokes, card insertions, etc.
   - **Port Output Events:** System outputs like screen displays, printed receipts, etc.
5. **Create a Hierarchy of State Machines:** For complex systems, break down the FSM into hierarchical levels. High-level FSMs represent broad states, while lower-level FSMs provide detailed breakdowns.
6. **Trace Paths Through the FSMs:** Identify all possible paths (threads) from the initial state to the final state, considering all transitions and states. Each path represents a possible sequence of user interactions and system responses.

7. **Define Test Cases Based on Threads:** Develop test cases that cover each identified thread. Ensure comprehensive coverage of all possible scenarios, including normal and edge cases.

7.5.3 Example - Finding Threads in the SATM System

Let us understand the process of finding threads in the SATM (Simple Automated Teller Machine) system.

Step 1: Define the Scope of the SATM System

The SATM system handles user interactions for tasks such as card insertion, PIN entry, and transaction processing (e.g., withdrawals, deposits).

Step 2: Identify Key User Interactions

- Inserting the ATM card.
- Selecting a transaction type.
- Receiving the card and receipt.
- Entering the PIN.
- Performing the transaction.

Step 3: Model the System with FSMs

(a) **Top-Level SATM State Machine :** The Top-Level SATM State Machine as shown in below figure provides a high-level overview of the ATM system's main states and transitions. The primary states include "Card Entry," "PIN Entry," and "Await Transaction Choice," which represent the key stages of user interaction. Transitions between these states are triggered by events such as inserting a legitimate card, entering a correct or incorrect PIN, and selecting a transaction. This diagram 7.2 helps visualize the overall workflow of an ATM transaction, ensuring that each critical step is accounted for in the system testing process.

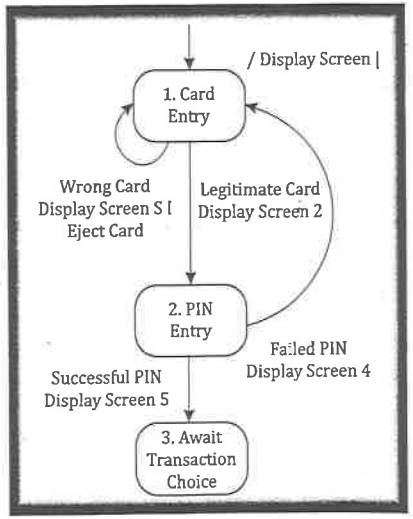


Figure 7.2: Top-Level SATM State Machine

- **States:** Card Entry, PIN Entry, Await Transaction Choice.
- **Transitions:** Legitimate Card, Wrong Card, Successful PIN, Failed PIN.

(b) PIN Entry Finite State Machine : The PIN Entry Finite State Machine as shown in below figure provides a detailed view of the PIN entry process within the SATM system. It includes states such as "First PIN Try," "Second PIN Try," and "Third PIN Try," allowing for multiple attempts to enter the correct PIN. Transitions are triggered by events like entering a correct or incorrect PIN and cancelling the operation. Successful entry of the correct PIN transitions the system to the "Await Transaction Choice" state, while incorrect PIN entries or cancellations lead to error states, guiding the system's response to various user inputs during PIN verification.

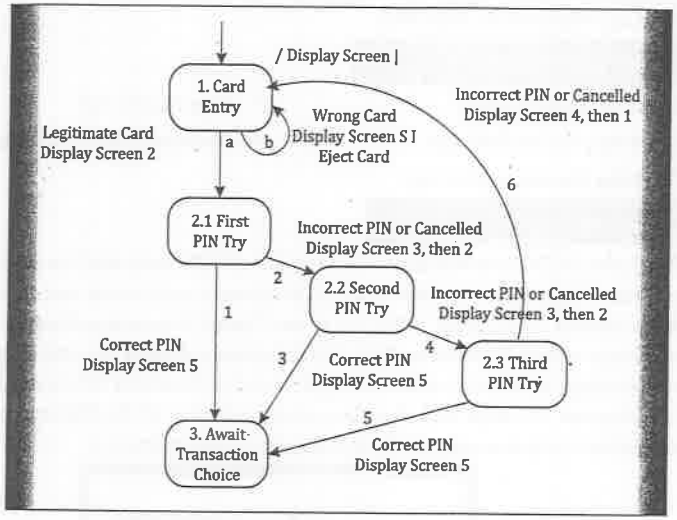


Figure 7.3: PIN Entry Finite State Machine

- **States:** Card Entry, First PIN Try, Second PIN Try, Third PIN Try, Await Transaction Choice, Incorrect PIN or Cancelled.
- **Transitions:** Correct PIN, Incorrect PIN, Cancelled.

(c) Detailed PIN Entry State Machine : The Detailed PIN Entry State Machine as shown in below figure offers a granular view of the digit-by-digit entry process for the PIN within the SATM system. This FSM starts with the state "0 Digits Received" and progresses through "1 Digit Received," "2 Digits Received," up to "4 Digits Received," reflecting the user's entry of each digit. Transitions occur with each digit entered and are shown by events like "digit/echo" and "cancel." Upon entering the fourth digit, the FSM transitions to either "Correct PIN" or "Incorrect PIN" based on the accuracy of the entered PIN. Additionally, the user can cancel the operation at any point, transitioning to the "Cancel Hit" state. This detailed state machine ensures that every step of the PIN entry is accurately modeled and tested, capturing all possible user interactions.

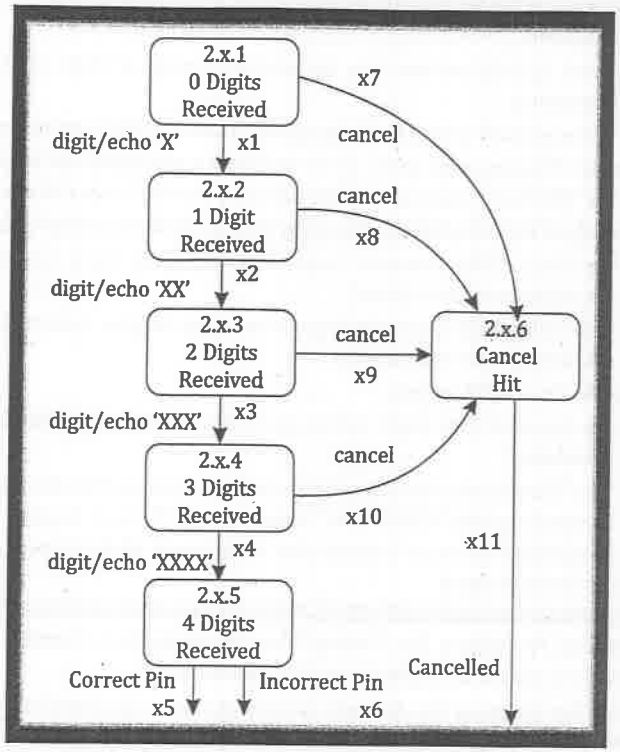


Figure 7.4 : Detailed PIN Entry State Machine

- **States:** 0 Digits Received, 1 Digit Received, 2 Digits Received, 3 Digits Received, 4 Digits Received, Cancel Hit, Correct PIN, Incorrect PIN.
- **Transitions:** Digit Entry, Cancel, Correct PIN, Incorrect PIN.

Step 4: Identify Port Events

Port Input Events :

- Digit: User entering digits on the keypad.
- Cancel: User pressing the cancel button.

Port Output Events

- Legitimate Card: Display Screen 2
- Wrong Card: Display Screen 1
- Correct PIN: Display Screen 5
- Incorrect PIN: Display Screen 4
- Cancelled: Display Screen 5

Step 5: Create a Hierarchy of State Machines

For detailed interactions, use lower-level FSMs like the Detailed PIN Entry State Machine to capture fine-grained user interactions.

Step 6: Trace Paths Through the FSMs

Example Threads in the SATM System:

1. **Thread for Successful Transaction:**
   - **Path:** Card Entry → PIN Entry (Correct PIN) → Await Transaction Choice
   - **Top-Level SATM State Machine:** Card Entry (Display Screen 1) → Legitimate Card → PIN Entry (Display Screen 2)
   - **PIN Entry Finite State Machine:** First PIN Try → Correct PIN → Await Transaction Choice
2. **Thread for Incorrect PIN Entry with Retry:**
   - **Path:** Card Entry → PIN Entry (Incorrect PIN) → Retry → Correct PIN → Await Transaction Choice
   - **Top-Level SATM State Machine:** Card Entry (Display Screen 1) → Legitimate Card → PIN Entry (Display Screen 2)
   - **PIN Entry Finite State Machine:** First PIN Try → Incorrect PIN → Second PIN Try → Correct PIN → Await Transaction Choice
3. **Thread for PIN Entry Cancellation:**
   - **Path:** Card Entry → PIN Entry (Partial Entry) → Cancel → Eject Card
   - **Top-Level SATM State Machine:** Card Entry (Display Screen 1) → Legitimate Card → PIN Entry (Display Screen 2)
   - **PIN Entry Finite State Machine:** First PIN Try → Cancelled → Eject Card

Step 7: Define Test Cases Based on Threads

Test Case 1: Successful Transaction

- **Preconditions:** Valid ATM card, correct PIN.
- **Steps:**
  a. Insert ATM card.
  b. Enter correct PIN.
  c. Select transaction type.
  d. Complete the transaction.
  e. Receive card and receipt.

Test Case 2: Incorrect PIN Entry with Retry

- **Preconditions:** Valid ATM card, incorrect PIN on first attempt, correct PIN on second attempt.
- **Steps:**

  a. Insert ATM card.
  b. Enter incorrect PIN.
  c. Re-enter correct PIN.
  d. Select transaction type.
  e. Complete the transaction.
  f. Receive card and receipt.

Test Case 3: PIN Entry Cancellation

- **Preconditions:** Valid ATM card.
- **Steps:**
  a. Insert ATM card.
  b. Start entering PIN.
  c. Press cancel.
  d. Card is ejected.

Summary

By following these steps, we can systematically identify threads in the SATM system. Using FSMs, we visualize the states and transitions, allowing us to trace paths that represent complete user interactions. Each thread is then used to create comprehensive test cases, ensuring thorough system testing. This approach helps in verifying that the SATM system handles all possible scenarios effectively, providing a robust and reliable user experience.

7.6 Structural Strategies for Thread Testing

Structural strategies for thread testing focus on the system's architecture and its underlying state machines (FSMs). These strategies ensure that all possible paths, nodes, and edges in the state machines are covered during testing. The goal is to achieve comprehensive coverage by testing all possible transitions and states to ensure that every part of the system's structure is validated.

The below given structural strategies ensure that the software is thoroughly tested by covering all possible states and transitions and validating each level of interaction from the most granular to the most complex.

Node and Edge Coverage Metrics

1. **Node Coverage :** Node coverage is a fundamental metric in FSM-based testing. It ensures that every state in the state machine is visited at least once during testing.
   - **Definition:** Node coverage ensures that every state (node) in the state machine is visited at least once during testing.
   - **Purpose:** The objective is to verify that the system can reach all possible states, confirming that no state is skipped or inaccessible. This is crucial for validating the completeness of the system's functionality.

- **Example:** In an ATM system, node coverage would require testing all states such as "Card Entry," "PIN Entry," "Await Transaction Choice," "Transaction Processing," etc. For instance, a test case would ensure that the system can successfully transition to the "PIN Entry" state after the user inserts a card.

2. **Edge Coverage :** Edge coverage extends the concept of node coverage by ensuring that the transitions between states are also tested. This metric is critical for validating the interactions and sequences within the system.
   - **Definition:** Edge coverage ensures that every transition (edge) between states is traversed at least once during testing.
   - **Purpose:** The objective is to verify that all possible transitions between states are correctly implemented and functional to ensure that the system can handle all pathways users might take. This helps identify any issues in the flow of operations within the system.
   - **Example:** In an ATM system, edge coverage would require testing transitions such as from "Card Entry" to "PIN Entry," "PIN Entry" to "Await Transaction Choice," "Await Transaction Choice" to "Transaction Processing," and any error handling transitions. For example, a test case would verify that after entering the correct PIN, the system correctly transitions from "PIN Entry" to "Await Transaction Choice."

Bottom-Up Threads

Bottom-up testing is a structured methodology that begins with the most detailed and specific components of the system, progressively integrating and testing higher levels of the system hierarchy. This approach ensures thorough validation at each level before moving to more complex interactions.

- **Methodology:** Bottom-up testing involves starting with the lower-level FSMs and progressively moving up to higher-level FSMs. This approach ensures that all fundamental interactions are validated before they are integrated into more complex workflows.
- **Process:**
  1. **Test Low-Level FSMs:** Begin by thoroughly testing the most detailed and specific FSMs that handle basic, granular functions or interactions.

     Example: In the SATM system, start by testing the detailed digit-by-digit PIN entry FSM.
  2. **Validate Interactions:** Ensure that all states and transitions within these low-level FSMs are correctly implemented and functioning.

     Example: Confirm that each digit entered transitions the system to the next state (e.g., from "1 Digit Received" to "2 Digits Received") and that the cancel operation works correctly.
  3. **Integrate and Move Up:** Once the lower-level FSMs are validated, integrate them and move to the next higher level in the hierarchy.

     Example: After validating the digit-by-digit FSM, integrate it with the higher-level PIN entry FSM to test the complete PIN entry process.

  4. **Repeat:** Continue this process until the highest level FSMs, which handle broad, overall system interactions, are tested.

     Example: Finally, test the top-level FSM that includes states like "Card Entry," "PIN Entry," and "Await Transaction Choice."

- **Focus :** The bottom-up approach focuses on detailed testing at each level, ensuring comprehensive coverage by thoroughly validating each component before integration.
- **Detailed Testing:** By starting at the bottom, testers can focus on detailed, specific interactions and ensure their correctness before integrating them into more complex interactions.

  Example: In the SATM system, focusing on the detailed digit entry ensures that each possible user action is accounted for and handled correctly before moving to higher-level interactions.
- **Comprehensive Coverage:** This approach helps ensure that no part of the system is overlooked and that interactions between various levels of the system are thoroughly tested.

  Example: In the SATM system, this ensures that transitions from "Card Entry" to "PIN Entry" and further to "Await Transaction Choice" are all tested, covering every possible interaction path.

7.7 Functional Strategies for Thread Testing

Functional strategies for thread testing focus on the system's functionality and the behavior specified by the requirements. These strategies ensure that the system performs as expected by testing specific functionalities and scenarios described in the requirements. The goal is to validate that the system meets its functional requirements and behaves correctly in various scenarios.

Functional strategies for thread testing ensure that the system's behavior aligns with its specified requirements by focusing on different aspects of functionality. Event-based testing ensures comprehensive coverage of all user interactions, port-based testing validates the handling of input and output events at each interface, and data-based testing confirms the integrity of data operations.

1. Event-Based Thread Testing

Event-based thread testing centers on the various events that can occur in a system, particularly at the points where users interact with the system. This approach ensures that all possible user actions and system responses are tested comprehensively.

1. **Approach:**
   - **Focus on Port Input Events and Their Sequences:** Identify all possible events that can be triggered by user interactions, such as keystrokes, button presses, or other inputs. Define sequences of these events to create realistic and edge-case scenarios.
   - **Testing Common and Uncommon Sequences:** Ensure that not only typical user interactions are tested but also less common or rare event sequences that might reveal hidden issues.
2. **Metrics:**
   - **Ensure Each Port Input Event Occurs:** Verify that every possible event is triggered at least once during testing.

- **Coverage of Common and Uncommon Sequences:** Test both regular usage patterns and unusual sequences of events to ensure robustness.

Example in the SATM System:

- **Common Sequence:** Insert card → Enter PIN → Select transaction → Complete transaction.
- **Uncommon Sequence:** Insert card → Enter incorrect PIN twice → Cancel transaction → Remove card.

2. Port-Based Thread Testing

Port-based thread testing focuses on the interaction between the system and its ports (input/output interfaces). This approach ensures that the system correctly handles all interactions at each port, including input and output events.

1. **Approach:**
   - **Focus on Events Occurring at Each Port:** Identify all ports (e.g., card reader, keypad, display) and the events associated with each port. Create test scenarios that involve various combinations of port interactions.
   - **Testing Interactions and Resulting Outputs:** Ensure that the system correctly processes inputs and generates appropriate outputs for each port event.
2. **Objective:**
   - Validate System's Handling of Port Events:
   - Test how the system responds to each port event and the correctness of the resulting outputs.

Example in the SATM System:

1. **Card Reader Port:**
   - **Input Event:** Insert card.
   - **Output Event:** Display prompt for PIN entry.
2. **Keypad Port:**
   - **Input Event:** Enter PIN digits.
   - **Output Event:** Echo digits on the screen and validate PIN.

3. Data-Based Thread Testing

Data-based thread testing focuses on the system's handling of data, ensuring that data is correctly created, retrieved, updated, and deleted (CRUD operations). This approach validates the integrity and correctness of data interactions within the system.

1. **Approach:**
   - **Focus on Data Interactions:** Identify all data entities and their interactions within the system. Create test cases that cover all CRUD operations for each data entity.
   - **Testing CRUD Operations:** Ensure that data can be accurately created, retrieved, updated, and deleted as per the requirements.

2. **Objective:**
   - **Validate Data Handling:** Confirm that the system correctly manages data throughout its lifecycle, maintaining data integrity and consistency.

Example in the SATM System:

1. **Account Data:**
   - **Create:** Adding new accounts.
   - **Retrieve:** Querying account balances.
   - **Update:** Modifying account balances after transactions.
   - **Delete:** Removing closed accounts.

7.8 Review Questions

Section - A

Each Question Carries Two Marks

1. What is System Testing?
2. What is Atomic System Function (ASF)?
3. What is a Thread in System Testing ?
4. Give an example of a Thread in an ATM System.
5. Mention the types of Threads in System Testing.
6. What are Unit-Level Threads? Give an example.
7. What are Integration Level Threads? Give an example.
8. What are System Level Threads? Give an example.
9. What is Event-Based Thread Testing?
10. What is Port-Based Thread Testing?
11. What is Data-Based Thread Testing?

Section - B

Each Question Carries Five Marks

1. What is System Testing? What are the Key Objectives of System Testing.
2. Explain the Features or Characterisers of System Testing.
3. What is Atomic System Function (ASF)? Explain the Importance of Atomic System Functions (ASFs).

4. Explain the Characteristics of Atomic System Functions (ASFs).
5. What is a Thread in System Testing ? Explain the Objectives of a Thread.
6. Explain the Characteristics and Importance of a Thread in System Testing.
7. Explain the Types of Threads in System Testing.
8. Discuss the Basic Concepts for Requirements Specification in System Testing.
9. Explain the General Procedure for Finding Threads in System Testing.

Section - C

Each Question Carries Eight Marks

1. What is System Testing? Explain the Features, Objectives and Importance of System Testing.
2. What is a Thread in System Testing ? Explain the Features, Objectives and Importance of a Thread.
3. Explain Thread Possibilities in the SATM System.
4. Explain the process of Finding Threads in the SATM System.
5. Explain the Structural Strategies for Thread Testing.
6. Explain the Functional Strategies for Thread Testing.

❖❖❖❖❖

UNIT - III

CHAPTER 8

INTERACTION TESTING

Contents

8.1 Introduction to Interaction Testing

Interaction testing focuses on faults and failures that occur due to interactions within a system. These interactions are often difficult to detect as they may remain hidden even after extensive testing. Interaction faults typically have a low probability of occurrence and often surface only after a significant number of threads have been executed. This chapter discusses the importance of specifying interactions as a first step in detecting and testing for these faults.

Interaction faults are difficult to detect because they occur under specific conditions that may not be easily replicated during standard testing procedures. They often involve complex dependencies and interactions between different components of a system, making them less predictable and harder to isolate.

To effectively detect and test for interaction faults, it is crucial to specify the interactions within the system as a first step. This involves identifying all possible interactions between components and understanding the conditions under which they occur. By doing so, testers can create more targeted and effective test cases.

8.1.1 What is Interaction Testing?

? What is Interaction Testing ?

Interaction testing involves examining the interactions between various components of a system to uncover faults that occur during these interactions. These faults can arise from complex dependencies and sequences of events that are not immediately noticeable. Interaction testing aims to identify and address these issues by focusing on how different parts of the system interact under various conditions.

8.1.2 Key Aspects of Interaction Faults in System Testing

1. **Complex Dependencies:** Interaction faults often occur from complex dependencies between different components. These components may work correctly in isolation but fail when combined due to unforeseen dependencies.

   Example: In a web application, an interaction fault might occur when the database, application server, and frontend all interact in a specific sequence that was not anticipated during unit testing.

2. **Sequences of Events:** The order in which events occur can significantly impact the system's behavior. An event sequence that works correctly in one scenario might fail in another due to differences in timing or context.

   Example: In an ATM system, a specific sequence of card insertion, PIN entry, and transaction selection might work correctly, but reversing the order of these actions or adding additional steps might reveal faults.

3. **Hidden Faults:** These faults are typically not apparent during initial testing phases. They remain dormant until a specific set of conditions triggers them.

   Example: A hidden fault in an e-commerce system might only surface when a user tries to apply multiple discount codes in a specific order, which was not covered in standard test cases.

4. **Low Probability of Occurrence:** Interaction faults often have a very low probability of occurring and hence it is difficult to detect without extensive testing.

   Example: A fault that only occurs when two users simultaneously attempt to book the last available slot for a service may only be discovered after numerous transactions.

5. **Comprehensive Testing:** To uncover interaction faults, comprehensive testing is required. This includes not only functional testing but also testing under different conditions, load levels, and configurations.

   Example: Stress testing an ATM system to simulate peak usage times can reveal interaction faults that do not appear under normal usage conditions.

6. **Scenario-Based Testing:** Interaction testing often involves creating detailed scenarios that mimic real-world usage. These scenarios are designed to explore different combinations and sequences of interactions.

   Example: Testing an online banking system by simulating different user scenarios, such as simultaneous fund transfers and bill payments, to uncover potential interaction faults.

8.1.3 Importance of Interaction Testing

Interaction testing is crucial for ensuring the seamless functioning of complex systems. By focusing on how various components interact, it uncovers hidden issues that might only emerge during specific sequences of events or under certain conditions. The importance of Interaction testing are listed below.

1. **Uncovers Hidden Issues:** Detects problems that arise only when components interact.
2. **Ensures System Integrity:** Verifies that all parts of the system work together correctly.
3. **Enhances Reliability:** Improves the overall stability and dependability of the system.
4. **Validates Complex Scenarios:** Tests interactions under various conditions and sequences.
5. **Prevents Failures:** Identifies potential faults before they impact users.
6. **Optimizes Performance:** Ensures that component interactions do not negatively affect system performance.
7. **Supports Maintenance:** Helps in detecting issues that could arise from future updates or changes.
8. **Improves User Experience:** Ensures a seamless and efficient experience for end-users.
9. **Reduces Risk:** Minimizes the likelihood of critical failures in production.
10. **Strengthens Robustness:** Enhances the system's ability to handle unexpected or rare conditions.

8.1.4 Features (or) Characteristics of Interaction Testing

Interaction testing is designed to thoroughly examine how various components of a system work together to ensure seamless and efficient performance. The Key features are :

1. **Component Interaction Analysis:** Examines how different parts of the system work together.
2. **Complex Dependency Testing:** Identifies and tests complex dependencies between components.
3. **Scenario-Based Testing:** Uses various scenarios to simulate real-world conditions and interactions.
4. **Sequential Event Validation:** Checks for issues that arise from specific sequences of events.
5. **Comprehensive Coverage:** Ensures all possible interactions are tested for faults.
6. **Dynamic Testing:** Adapts to changes in the system and tests new interactions as they arise.
7. **Automated Test Execution:** Utilizes automated tools to efficiently test numerous interaction scenarios.
8. **Risk-Based Prioritization:** Focuses on interactions with the highest potential impact on system performance.
9. **Performance Monitoring:** Assesses the impact of interactions on overall system performance.
10. **Fault Isolation:** Helps in pinpointing the exact location and cause of interaction faults.
11. **Regression Testing:** Ensures new changes do not introduce new interaction issues.
12. **Real-Time Feedback:** Provides immediate insights into the effects of interactions during testing.
13. **Scalability:** Can be applied to both small and large systems with complex interactions.
14. **Documentation and Reporting:** Generates detailed reports on interaction faults and testing outcomes.

8.1.5 Advantages and Disadvantages of Interaction Testing

Interaction testing offers significant benefits by uncovering hidden faults, ensuring comprehensive coverage, and improving system reliability. However, it also presents challenges such as high resource requirements, complex test design, and difficulty in isolating issues. Balancing these advantages and disadvantages is key to effective interaction testing and maintaining a robust system.

Advantages of Interaction Testing

1. **Uncovers Hidden Faults:**
   Benefit: Detects issues that arise only when components interact, ensuring a more reliable system.
   Example: Identifying faults in a web application that only appear when the database, server, and frontend interact in specific ways.
2. **Ensures Comprehensive Coverage:**
   Benefit: Tests a wide range of scenarios and interactions, providing thorough validation.
   Example: Using scenario-based testing to cover various user behaviors in an online banking system.

3. **Improves System Reliability:**
   Benefit: Enhances the overall stability and performance by addressing complex dependencies.
   Example: Validating that an ATM system handles all possible sequences of user actions correctly.
4. **Validates Real-World Scenarios:**
   Benefit: Mimics actual usage conditions, leading to more accurate and relevant test results.
   Example: Simulating peak usage times to identify potential issues in an e-commerce platform.
5. **Prevents Critical Failures:**
   Benefit: Reduces the risk of major issues occurring in production by catching them early.
   Example: Detecting a rare fault in a service booking system where simultaneous bookings can cause a crash.

Disadvantages of Interaction Testing

1. **Resource Intensive:**
   Drawback: Requires significant time, effort, and computational resources to perform extensive testing.
   Example: Running comprehensive interaction tests on a large-scale enterprise application can be costly and time-consuming.
2. **Complex Test Design:**
   Drawback: Designing tests for all possible interactions and sequences can be very complex and challenging.
   Example: Creating detailed scenarios for a complex healthcare management system to cover all interactions.
3. **Difficult to Isolate Issues:**
   Drawback: Pinpointing the exact cause of interaction faults can be challenging due to the complexity of dependencies.
   Example: Troubleshooting a fault in a financial system where multiple components and interactions are involved.
4. **Low Probability Faults:**
   Drawback: Interaction faults often have a low probability of occurring, making them difficult to detect without extensive testing.
   Example: A rare fault in a social media platform that only occurs under specific conditions with simultaneous user actions.
5. **High Maintenance:**
   Drawback: Keeping interaction tests up-to-date with system changes requires ongoing effort and can be difficult to manage.
   Example: Continuously updating tests for a rapidly evolving software application to ensure all new interactions are covered.

8.2 Context of Interaction in Interaction Testing

Understanding the context of interaction is essential for effective interaction testing. This involves examining how different entities within the system interact and the specific conditions under which these interactions occur. By thoroughly understanding the context, testers can design more accurate and effective test cases to identify potential faults. Key factors to consider include the nature of the

interactions, their timing, their physical or logical positions, and the rules governing their execution. This approach ensures that all possible scenarios and conditions are covered, leading to a more robust and reliable system.

1. **Interacts With Relationship:**
   - This relationship indicates that entities such as data, actions, events, and threads interact with each other.
   - **Example:** In an ATM system, the interaction between entering a PIN (action) and verifying it against stored data (data). Understanding these interactions helps identify where faults may occur.
2. **Location (Time and Position):**
   - **Time:** Interactions have a temporal component, occurring at specific moments or over durations.
   - **Position:** Interactions occur at specific physical or logical locations.
   - **Example:** User performs a transaction at an ATM located in a busy urban area at 3 PM. This specific time and location are critical because high usage can lead to different interactions compared to a less busy time.

     Testing under various times and locations ensures the system can handle peak usage and different environmental conditions, such as network connectivity issues in specific locations.
3. **Time Views:**
   - **Instantaneous View:** Focuses on events happening at a specific point in time.
   - **Duration View:** Considers the length of time an event or interaction lasts.
   - **Example:** The duration from when a user inserts their card to when it is ejected might take longer during peak hours due to system load. An instantaneous view would check the exact moment the card is read, while a duration view would consider the entire process time. Both views are essential for ensuring smooth and timely transactions, especially under varying load conditions.
4. **Processor Rules:**
   - **Execution Time:** Threads have a positive time duration, meaning they take time to execute.
   - **Simultaneity:** Multiple threads cannot execute simultaneously on a single processor.
   - **Event Duration:** Events have positive time durations and cannot occur simultaneously across processors.
   - **Example:** When an ATM processes a transaction, it must handle multiple operations in sequence: card reading, PIN verification, transaction processing, and receipt printing. These operations cannot happen simultaneously but must follow a strict sequence to prevent errors. Adhering to processor rules ensures that each operation completes correctly without interference, which is crucial for maintaining transaction integrity and user trust.

8.3 Taxonomy of Interactions

Taxonomy of interactions refers to a structured classification system that categorizes different types of interactions within a system based on specific criteria. This classification helps in understanding and managing the complexity of interactions, making it easier to identify, test, and address potential faults.

The main purpose of creating a taxonomy of interactions is to systematically organize and describe the various ways in which components of a system can interact. This facilitates more targeted and effective testing by providing a clear framework for identifying and understanding the different types of interactions that can occur.

Types of Interactions in System Testing: Classification Based on Time and Position

In the context of system testing, interactions can be classified based on two primary aspects: location, which includes **time** and **position**.

1. **Time:**
   - **Time-Independent (Static) Interactions:** Interactions that are not time-dependent and occur regardless of timing. For instance, interactions between two data items that occur regardless of when they are accessed or modified.

     Example : When a user approaches the ATM, the machine displays a welcome screen. This interaction involves the ATM's processor fetching the static welcome message stored in its memory and displaying it on the screen. The timing of this interaction is irrelevant; the welcome message will always be the same, regardless of when it is accessed or displayed.
   - **Time-Dependent (Dynamic) Interactions:** Interactions where the timing or sequence of operations is critical. It require specific order or timing. For example, operations that must occur in a specific order or within a certain timeframe.

     Example : When a user requests their account balance, the ATM sends a request to the bank's server to retrieve the most up-to-date account balance. This interaction is time-dependent because the account balance can change with each transaction made by the user or other users. The ATM must request the balance from the bank's server at the exact time of the user's inquiry to provide the current balance.
2. **Position:**
   - **Single Processor Interactions:** Interactions that occur within a single processing unit. These interactions do not require coordination between multiple processors and are confined to one processing environment.

     Example : After verifying the user's credentials and checking the account balance, the ATM dispenses the requested amount of cash. This involves internal operations such as triggering the cash dispenser, updating the local transaction log, and printing a receipt, all managed by the ATM's single processor. The ATM's processor controls the mechanical components that count and dispense the cash, update the transaction log stored in the local memory, and print the receipt.

- **Multiple Processors Interactions:** Interactions that span across multiple processing units. It require synchronization and coordination between multiple processors. These interactions are more complex due to the need for managing communication and data consistency between processors.

  Example : When a user requests a balance inquiry, the ATM's local processor sends a request to the central bank's server to fetch the latest account balance. This interaction involves the ATM processor communicating with the remote server, receiving the response, and displaying the balance to the user. In this case, ATM's server communicates and coordinates with central bank's server and hence it is considered as multiple processor interactions.

The detailed types of interactions in software systems based on time dependency and the number of processors involved are as follows:

1. **Static Interactions in a Single Processor:** Interactions independent of time within a single processor.

   Example: : Accessing and displaying the ATM's welcome screen. When a user approaches the ATM, the machine's processor fetches the static welcome message stored in its local memory and displays it on the screen. The timing of this interaction is irrelevant; the welcome message will always be the same, regardless of when it is accessed or displayed.

2. **Static Interactions in Multiple Processors:** Time-independent interactions across multiple processors. Synchronizing static data between different processors to maintain consistency.

   Example: Synchronizing static configuration data between the ATM and the central server. For instance, updating the list of supported languages or currencies in the ATM. The central server periodically sends updated static data to all connected ATMs to ensure consistency. This process is independent of the specific timing of individual updates as long as the data remains consistent across the network.

3. **Dynamic Interactions in a Single Processor:** Time-dependent interactions within a single processor. Executing a sequence of instructions where the order of operations is critical for correctness.

   Example: Entering and validating a user's PIN. When a user enters their PIN, the ATM's processor must validate the entered PIN against the stored hash of the correct PIN. This involves reading the PIN input, processing it, and verifying it in the correct sequence. The order and timing are crucial to ensure the security and correctness of the authentication process.

4. **Dynamic Interactions in Multiple Processors:** Time-dependent interactions spanning across multiple processors. Distributed computing tasks where processes must coordinate based on timing for proper execution

   Example: Performing a balance inquiry. When a user requests to check their account balance, the ATM's processor sends a request to the central bank's server to retrieve the latest account balance. This interaction involves the ATM processor communicating with the remote server, receiving the response, and displaying the balance to the user. The timing is critical because the account balance can change with each transaction made by the user or other users. The ATM must request and display the balance in real-time to ensure accuracy.

8.3.1 Static Interactions in a Single Processor

Static interactions in a single processor refer to data interactions that remain consistent over time and are entirely processed within one processing unit. These interactions are crucial for maintaining data integrity and preventing data corruption or inconsistency within the system. Understanding and managing these static interactions ensure that the system operates correctly and reliably.

? What are Static Interactions in a Single Processor ?

Static interactions in a single processor refer to the relationships and dependencies between components or elements within the processor that remain constant and do not change during the execution of a program. These interactions are typically defined at design time and do not involve dynamic changes based on inputs or events.

Example Static Interactions in a Single Processor

Scenario: ATM Welcome Screen Display

1. **Static Welcome Message Display:**
   - **Data:** Welcome message stored in the ATM's memory.
   - **Event:** ATM powers on.
   - **Thread:** The ATM processor retrieves and displays the welcome message.
   - **Interaction:** The welcome message is consistently displayed on the screen every time the ATM is powered on, regardless of any other operations or timing, managed by the single processor.
2. **Card Reader Activation:**
   - **Data:** Status of the card reader (ready or not ready).
   - **Event:** ATM initialization.
   - **Thread:** The processor checks the card reader status.
   - **Interaction:** The card reader is set to 'ready' status upon ATM initialization, ensuring it is prepared to accept cards, managed solely within the single processor.

Key Concepts

In the context of static interactions in a single processor, it is crucial to understand the concepts of duration and port devices because these elements play a significant role in how the system processes and maintains data integrity without being influenced by timing. This understanding helps in identifying and resolving potential conflicts or inconsistencies within the system.

By understanding the key concepts related to static interactions, such as port devices, data interactions, and propositional logic, we can effectively analyze and manage these interactions to maintain system functionality and data integrity.

1. Duration and Port Devices:

1. **Port Devices:**
   - **No Duration:** Port devices and data are unique constructs in a system because they do not have a duration. They exist in a state that does not change over time within the context of the processor.

Example: The card reader in the SATM (Secure Automated Teller Machine) is a port device that continuously exists as long as the ATM is operational. The card reader does not have a specific duration associated with its presence; it is always available to interact with customers' ATM cards whenever they are inserted.

- **Physical Interaction:** Port devices interact physically within the system, consuming space and power. They are responsible for the physical transfer of data between different parts of the system.

  Example : When a customer inserts their ATM card into the card reader, the card reader facilitates the physical transfer of data from the magnetic strip or chip on the card to the ATM's processor. This interaction consumes physical space (the slot in the ATM where the card is inserted) and power (the mechanism that reads the card's data). The card reader plays a crucial role in transferring account information from the card to the ATM system for processing transactions.

2. **Data Interactions:**
   - **Logical Interaction:** Data interactions occur logically and are essential for maintaining data integrity within a database. These interactions involve operations such as reading, writing, updating, and deleting data.: Ensuring data consistency and integrity is vital to avoid corruption, which can lead to incorrect system behavior and data loss.

     Example: When a customer checks their account balance using the SATM, the interaction involves several logical steps. The ATM reads the customer's account information from the card, communicates with the bank's central database to retrieve the current balance, and then displays the balance on the screen. This logical interaction ensures that the data is accurately retrieved and presented to the customer.

2. Propositional Logic Analysis :

Propositional logic provides a structured framework for analyzing static interactions within a single processor. By defining propositions and examining their logical relationships, we can reason about the truth values of different statements related to the system's data and operations. Understanding logical concepts such as contraries, sub-contraries, contradictories, and sub-alterns helps in evaluating the consistency and validity of data interactions within the system.

Example

Scenario: Withdrawing Cash

- **Proposition p:** WithdrawalAmount = ₹500.00
- **Proposition q:** SufficientBalance = True

In the SATM (Simple Automated Teller Machine) system, when a user attempts to withdraw cash, the following static interactions occur:

- The system checks the requested withdrawal amount.
- It verifies whether the account has a sufficient balance to cover the withdrawal.
- These checks are independent of timing and always produce the same results if the data remains unchanged.

Propositional Logic Analysis:

1. **Contraries:**
   - **Definition:** Contraries are two propositions that cannot both be true at the same time.
   - **Example:** If the system's rules state that a withdrawal of ₹500.00 cannot be processed due to insufficient balance, then the propositions "WithdrawalAmount = ₹500.00" and "SufficientBalance = True" cannot both be true simultaneously. If there is not enough balance (SufficientBalance = False), the system cannot process the withdrawal amount of ₹500.00 (WithdrawalAmount = ₹500.00 must also be False).
2. **Sub-contraries:**
   - **Definition:** Both propositions cannot be false at the same time.
   - **Example:** If the system ensures sufficient balance before allowing any withdrawal, either the proposition "WithdrawalAmount = ₹500.00" or "SufficientBalance = True" must be true. This ensures that the system functions correctly by preventing withdrawals when the balance is insufficient.
3. **Contradictories:**
   - **Definition:** If one proposition is true, the other must be false, depending on the system's rules.
   - **Example:** If the system requires sufficient balance for processing a withdrawal and the balance is insufficient (SufficientBalance = False), then the withdrawal amount of ₹500.00 (WithdrawalAmount = ₹500.00) must also be false because the system will not process the withdrawal without sufficient balance.
4. **Sub-alterns:**
   - **Definition:** If one proposition is true, the other must also be true.
   - **Example:** If the system processes a withdrawal amount of ₹500.00 (WithdrawalAmount = ₹500.00) and requires sufficient balance for processing, then the proposition "SufficientBalance = True" must also be true. This ensures that the withdrawal amount can only be processed if the account has sufficient balance.

By understanding these logical relationships, we can see how the ATM system operates based on certain conditions and rules to ensure secure and accurate transactions.

Procedure for Testing Static Interactions in a Single Processor

Testing static interactions within a single processor involves ensuring that data interactions occur consistently and correctly, independent of timing.

1. **Identify Static Data Interactions :** Determine all data interactions that do not depend on time and are confined within a single processor.

   Example : Identifying the data flow within the ATM processor that does not rely on timing, such as retrieving customer account information from the local database.

2. **Define Propositions and Logical Relationships :** Use propositional logic to define and analyze the relationships between data interactions.

   Example: Using propositional logic to define relationships, like ensuring that a customer must enter a correct PIN to access their account information.

3. **Develop Test Cases:** Create test cases that ensure data interactions are consistent and logical.

   Example : Creating a test case where a customer enters an incorrect PIN three times consecutively to validate if the system locks the account as per logic.

4. **Execute Tests and Monitor Results :** Run the test cases in a controlled environment to check for consistency and correctness.

   Example : Running the test case in a controlled environment to observe if the system locks the account after three incorrect PIN entries as expected.

5. **Analyze Results and Identify Issues :** Analyze the outcomes of the test executions to identify inconsistencies or logical errors.

   Example: Analyzing the test outcome to determine if the system behaved logically by locking the account after three failed PIN attempts.

6. **Implement Fixes and Retest :** Correct any identified issues and verify the fixes through retesting.

   Example : If any issues are identified, correct the logic related to locking the account and retest to ensure the fix resolves the issue.

7. **Continuous Monitoring and Maintenance :** Continuously monitor the system to ensure that static interactions remain consistent over time.

   Example : Continuously monitoring the system to ensure that the static interactions related to account access and security remain consistent and error-free over time.

8.3.2 Static Interactions in Multiple Processors

Static interactions in multiple processors refer to interactions that are not dependent on timing but occur across multiple processing units. These interactions require careful synchronization and coordination between processors to ensure data consistency and integrity.

? What are Static Interactions in Multiple Processors ?

Static interactions in multiple processors refer to interactions that are not dependent on timing but occur across multiple processing units. These interactions are fixed and do not change during the execution of tasks across multiple processors.

Example **Static Interactions in Multiple Processors**

Scenario: Synchronizing ATM Maintenance Logs

1. **Log Synchronization:**
   - **Data:** Maintenance logs stored in each regional server.
   - **Event:** Daily maintenance check.
   - **Thread:** Each regional server collects and updates logs.
   - **Interaction:** Each regional server independently collects logs and synchronizes with the central server without the need for timing coordination. For instance, each ATM records its maintenance activity and sends this data to a regional server. The regional server then synchronizes the logs with the central server at a scheduled time, ensuring data consistency across the network.

2. **Software Version Check**
   - **Data:** Current software version installed on ATMs.
   - **Event:** Routine software check initiated by the central server.
   - **Thread:** Each ATM reports its software version to the central server.
   - **Interaction:** The central server collects software version data from each ATM without requiring synchronized timing. Each ATM independently sends its software version information during the routine check to ensure all ATMs are running the latest software version as verified by the central server.

Key Concepts

1. **Location of Data:** In systems with multiple processors, data may be distributed across different locations. Synchronizing this data involves ensuring that updates made to the data in one location are reflected accurately in all other locations. This synchronization is essential to maintain consistency and coherence in the system.

   Example : Each ATM maintains a local copy of configuration data that should be consistent with the master data stored on the central server. This includes language options, transaction limits, etc.

2. **Contrary Data:** Contrary data refers to situations where different processors hold conflicting information about the same entity. Resolving contrary data is crucial to prevent inconsistencies and ensure that the system operates based on accurate and up-to-date information.

   Example : Imagine ATM-A in Bangalore has a configuration that includes English and Kannada, while ATM-B in Hyderabad includes English and Telugu. If a new language (e.g., Hindi) is added to the central server but the update fails on ATM-B, this creates contrary data situations where the configurations are inconsistent.

3. **Distributed Data Challenges:** Distributed data interactions involve managing data spread across multiple processors, which can introduce complexities in maintaining consistency. Challenges include ensuring timely updates, resolving conflicts, and preventing data discrepancies that can arise due to the distributed nature of the system.

   Example : In a distributed banking system with a central server, regional servers, and ATMs, managing data consistency poses significant challenges. For instance, when a new language (e.g., Hindi) is added to the central server, this update must be propagated quickly to the regional servers and ATMs to avoid discrepancies. Thus, ensuring timely updates, conflict resolution, and data consistency are crucial for maintaining reliable distributed data interactions. Efficient synchronization protocols and distributed consensus algorithms help resolve such conflicts. Additionally, network latencies or partial failures can lead to data discrepancies, which robust error detection and correction mechanisms can mitigate. Thus, ensuring timely updates, conflict resolution, and data consistency are crucial for maintaining reliable distributed data interactions.

Procedure for Testing Static Interactions in a Multiple Processors

Testing static interactions in multiple processors involves ensuring that data interactions occur consistently and correctly across different processors. This process requires careful synchronization and monitoring to maintain data integrity across the distributed system.

1. **Identify Static Data Interactions :** Determine all data interactions that are not dependent on time and occur across multiple processors.

   Example: Identifying the synchronization of configuration settings between ATMs, regional servers and the central server in a banking system. Configuration settings such as supported languages, transaction limits, and security protocols typically remain static until explicitly changed.

2. **Define Propositions and Logical Relationships :** Use propositional logic to define and analyze relationships between data interactions across processors.

   Example: Using propositional logic to ensure that any update to the configuration settings on the central server is reflected in the regional servers.

   Proposition p: ConfigurationSettingCentral = "Enabled"

   Proposition q: ConfigurationSettingRegional = "Enabled"

   Proposition r: ConfigurationSettingAM = "Enabled"

   Logical Relationship: p, q and r shoule be equal for consistency.

3. **Develop Test Cases :** Create test cases to ensure that data interactions are consistent and logical across multiple processors.

   Example: Creating a test case to simulate updating a configuration setting on the central server and verifying that the update is correctly propagated to all regional servers and ATMs. For instance, if the central server enables a new security feature, the test case should verify that this change is applied across all regional servers as well as ATMs.

4. **Execute Tests and Monitor Results :** Run the test cases in a controlled environment to check for consistency and correctness.

   Example: Running the test case in a controlled environment where the central server updates a configuration setting, and monitoring to ensure all regional servers and ATMs reflect the change accurately.

5. **Analyze Results and Identify Issues :** Analyze the outcomes of the test executions to identify inconsistencies or logical errors.

   Example: Analyzing the test results to ensure that the configuration setting update on the central server is accurately reflected on all regional servers and ATMs without any discrepancies.

6. **Implement Fixes and Retest :** Correct any identified issues and verify the fixes through retesting.

   Example: If discrepancies are found, correcting the synchronization logic between the central server, regional servers and ATMs, and retesting to ensure the issue is resolved. For instance, if a regional server did not update the configuration setting correctly, the issue might be due to a synchronization bug that needs fixing.

7. **Continuous Monitoring and Maintenance :** Continuously monitor the system to ensure that static interactions remain consistent over time.

   Example: Implementing continuous monitoring tools to ensure that configuration settings are consistently synchronized between the servers, and that no discrepancies occur over time. Any detected inconsistencies should trigger alerts for immediate resolution.

8.3.3 Dynamic Interactions in a Single Processor

Dynamic interactions within a single processor consider the implications of time and sequence for system operations. Unlike static interactions, which are time-independent, dynamic interactions are highly sensitive to the order and timing of events. These interactions involve not only data but also events and threads, expanding the scope to cover various system states and transitions. In dynamic interactions, the system must manage the flow of operations in a way that respects the sequence and timing constraints to ensure correct functionality.

? What are Dynamic Interactions in a Single Processor ?

Dynamic interactions in a single processor refer to the behavior of components within a computing system that change over time or in response to specific inputs or events. These interactions involve the dynamic processing of data, state changes, and the execution of tasks within a single processing unit.

Example: Dynamic Interactions in a Single Processor

Scenario: ATM Transaction Authentication and Logging

1. **Card Insertion and PIN Entry:**
   - **Data:** User's card information and entered PIN.
   - **Event:** User inserts card and enters PIN.
   - **Thread:** The system reads the card data, verifies the PIN against stored information, and checks for valid authentication.
   - **Interaction:** The system dynamically verifies the PIN and authentication status in real-time, ensuring the correct PIN is entered before allowing further actions. All these operations are managed within the single processor of the ATM.
2. **Transaction Selection:**
   - **Data:** Available transaction types (e.g., balance inquiry, withdrawal).
   - **Event:** User selects the desired transaction type.
   - **Thread:** The system processes the user's selection and prepares to execute the chosen transaction.
   - **Interaction:** The system updates the display and internal state based on the user's selection, to ensure that the selected transaction is prepared for execution within the single processor.

3. **Logging the Transaction:**
   - **Data:** Details of the transaction, such as type, timestamp, and user actions.
   - **Event:** Completion of the transaction selection process.
   - **Thread:** The system logs the transaction details into a local log file for record-keeping and auditing.
   - **Interaction:** The system dynamically logs the transaction in real-time to ensure that all details are accurately recorded for future reference. This process is entirely handled within the single processor of the ATM.

Key Concepts

1. **N-connectedness :** N-connectedness refers to the degree of dependency between data items or events within a system. It can be classified into various types based on their interaction and dependency levels:
   - **0-connected data:** These are independent data items with no dependencies.

     Example: Consider an ATM system where a customer's name and account type (savings or current) are stored as independent data items. These data items do not affect each other and can exist independently.
   - **1-connected data:** Data items that are inputs to the same action.

     Example: In an ATM, when a customer initiates a transaction, both the account number and PIN are inputs to the authentication action. They are 1-connected because they are required together to perform the authentication.
   - **2-connected data:** Data items used together in computations or workflows.

     Example: When calculating the available balance after a withdrawal, the ATM uses the current balance and the withdrawal amount. These data items are 2-connected as they are used together in the balance computation process.
   - **3-connected data:** Data items that are deeply related, often involving repetition or semaphore mechanisms.

     Example: The ATM keeps a log of multiple transactions (deposits, withdrawals) for a customer. These transaction entries are 3-connected because they represent a sequence of related data items involving repetitive interactions and checks.
2. **Interaction Types**
   - **Data-Data Interactions:** Multiple data items interacting with each other.

     Example: During a funds transfer in an ATM, the system updates both the sender's and receiver's account balances. The data for both accounts interact to ensure that the transfer is correctly reflected in each account.
   - **Data-Events Interactions:** Data interactions triggered by events.

     Example: When a user requests an account balance, the ATM retrieves the account data and triggers the event to display the balance on the screen. The retrieval of account data is initiated by the user's request event.

   - **Events-Events Interactions:** Interactions between different events.

     Example: If a user cancels a transaction after entering the PIN but before selecting the transaction type, the cancel event interacts with the PIN entry event to halt the process and reset the system state.
   - **Events-Threads Interactions:** Events causing changes in thread execution.

     Example: In the ATM, the event of detecting card insertion initiates a new thread for the transaction process. The thread then handles subsequent events like PIN entry, transaction selection, and completion.
   - **Threads-Threads Interactions:** Interactions between different execution threads.

     Example: In an ATM, one thread might handle user authentication while another manages transaction logging. These threads interact to ensure that once a transaction is authenticated, it is correctly logged in the system.

Procedure to Test Dynamic Interactions in a Single Processor

1. **Identify Dynamic Interactions:** Determine all operations and data interactions dependent on timing and sequence.

   Example: Identifying steps in the withdrawal process from PIN entry to cash dispensing.
2. **Define Interaction Sequences:** Map out the sequence of events and data interactions.

   Example: Sequence from account verification, balance check, cash dispensing, and receipt printing.
3. **Develop Test Cases:** Create test cases covering all possible sequences and timing variations.

   Example: Test case for successful withdrawal, insufficient funds scenario, and card retention after incorrect PIN entries.
4. **Simulate Events:** Simulate the dynamic interactions by triggering events in the defined sequences.

   Example: Simulate card insertion, PIN entry, balance check, and cash withdrawal.
5. **Monitor System Responses:** Observe and log the system's responses to ensure they match expected outcomes.

   Example: Verify that the system correctly dispenses cash only if sufficient balance is available.
6. **Analyze and Validate Results:** Compare the actual outcomes with expected results to identify discrepancies.

   Example: Check if the receipt printed correctly reflects the transaction details.
7. **Address Issues and Retest:** Fix any identified issues and rerun tests to ensure the fixes work.

   Example: Correct any errors in the withdrawal sequence and retest for consistency.
8. **Continuous Monitoring:** Continuously monitor the system to ensure dynamic interactions remain consistent over time.

   Example: Regularly check the accuracy of transactions and system logs for anomalies.

8.3.4 Dynamic Interactions in a Multiple Processors

Dynamic interactions in multiple processors involve the coordinated execution of processes across multiple processors, where the timing and sequence of operations are crucial. These interactions are not limited to a single processor and must manage the complexities that arise from the need for synchronization, real-time data exchange, and consistency across multiple systems. This type of interaction is essential in distributed computing environments where tasks are shared among various processors to achieve efficiency and scalability.

? What are Dynamic Interactions in Multiple Processors ?

Dynamic interactions in multiple processors refer to the interactions that occur in a system with more than one processor, where data and tasks are dynamically distributed and processed across these processors. These interactions are influenced by factors such as timing, synchronization, resource sharing, and communication between processors. Managing these interactions is crucial for ensuring the system's performance, consistency, and reliability.

Example: Dynamic Interactions in Multiple Processors

Scenario: Withdrawal Process

1. **Initiating Withdrawal Request:**
   - **Data:** User's account details stored on the central bank server.
   - **Event:** User requests a withdrawal at an ATM.
   - **Thread:** The ATM sends a withdrawal request to the regional server.
   - **Interaction:** The ATM processor sends the user's account information and requested withdrawal amount to the regional server, which then forwards it to the central bank server for verification. This interaction requires communication and synchronization between the ATM's processor, the regional server, and the central bank server.
2. **Account Balance Verification:**
   - **Data:** Real-time account balance on the central bank server.
   - **Event:** Central bank server processes the withdrawal request.
   - **Thread:** The central bank server verifies the account balance and approves or denies the request.
   - **Interaction:** The central bank server checks the current balance, deducts the requested amount if funds are sufficient, and sends a confirmation back to the regional server. This process must occur dynamically and promptly to ensure the transaction status is based on real-time data.
3. **Dispensing Cash:**
   - **Data:** Approval message from the central bank server.
   - **Event:** ATM receives approval to dispense cash.
   - **Thread:** The ATM processor triggers the cash dispenser.
   - **Interaction:** Upon receiving approval from the central bank server via the regional server, the ATM processor initiates the cash dispensing mechanism. This dynamic interaction ensures that the user receives the correct amount of cash and that the transaction is logged correctly in the central bank's records.

4. **Transaction Logging and Receipt Printing:**
   - **Data:** Transaction details.
   - **Event:** Completion of the transaction.
   - **Thread:** The ATM logs the transaction and prints a receipt.
   - **Interaction:** The transaction details, including the amount dispensed and the updated account balance are logged in both the ATM's local memory and the central bank's server. The ATM then prints a receipt for the user.

Key Concepts

Dynamic interactions in multiple processors involve the coordination and synchronization of operations across different processing units. These interactions are crucial for ensuring that distributed systems function correctly and efficiently.

1. **Concurrency and Parallelism:**
   - **Concurrency:** Refers to the ability of a system to manage multiple tasks simultaneously. In a multi-processor environment, tasks can be executed concurrently, which requires careful management to avoid conflicts and ensure data consistency.
   - **Parallelism:** Involves dividing a task into smaller sub-tasks that can be executed simultaneously on different processors. This improves performance but requires mechanisms to synchronize the tasks.
2. **Synchronization:**
   - **Mutexes and Locks:** Used to prevent concurrent access to shared resources. A mutex (mutual exclusion) ensures that only one processor can access a resource at a time.
   - **Semaphores:** A signaling mechanism used to control access to shared resources by multiple processors. Semaphores help in coordinating complex interactions.
3. **Communication Protocols:**
   - **Message Passing:** Processors communicate by sending messages to each other. This is essential in distributed systems where processors do not share memory.
   - **Shared Memory:** Some systems use shared memory for communication, where processors read from and write to a common memory space. Proper synchronization mechanisms are required to manage access to shared memory.
4. **Data Consistency and Coherence:**
   - **Consistency Models:** Define the rules for how data changes are propagated across processors. Examples include strong consistency, eventual consistency, and causal consistency.
   - **Cache Coherence Protocols:** Ensure that multiple copies of data across different caches are kept consistent. Protocols like MESI (Modified, Exclusive, Shared, Invalid) are used to manage cache coherence.
5. **Fault Tolerance and Reliability:**
   - **Redundancy:** Implementing redundant components to handle failures. If one processor fails, another can take over its tasks.

- **Checkpointing and Rollback:** Periodically saving the state of a system so that it can be restored to a previous state in case of failure.

6. **Load Balancing:**
   - **Static Load Balancing:** Distributing tasks to processors at the start of execution based on predefined criteria.
   - **Dynamic Load Balancing:** Adjusting the distribution of tasks among processors during execution based on current load and performance metrics.

Procedure to Test Dynamic Interactions in Multiple Processors

1. **Identify Dynamic Data Interactions:** Determine the data interactions that depend on timing and involve multiple processors.

   Example: Coordinating account balance updates between the ATM, central server, and regional servers.
2. **Define Interaction Scenarios:** Outline scenarios that involve multiple processors interacting dynamically.

   Example: User initiates a withdrawal, and the system updates balances across different servers.
3. **Develop Test Cases:** Create test cases to validate dynamic interactions across processors.

   Example: Test cases for withdrawal transactions that verify the balance updates correctly on both the central and regional servers.
4. **Simulate Multi-Processor Environment:** Set up a testing environment that mimics the distributed system.

   Example: Use a network of servers to simulate the ATM, central server, and regional servers.
5. **Execute Tests and Monitor Interactions:** Run the test cases and monitor the interactions between processors.

   Example: Observe how the balance is updated and ensure no inconsistencies occur during transactions.
6. **Analyze Results and Resolve Issues:** Analyze the test results to identify any synchronization or data consistency issues.

   Example: If discrepancies are found in account balances, determine the cause and fix the synchronization mechanism.
7. **Implement Fixes and Retest:** Correct identified issues and retest to ensure fixes are effective.

   Example: Adjust the communication protocol between servers to ensure timely and consistent balance updates.
8. **Continuous Monitoring:** Continuously monitor the system to ensure dynamic interactions remain consistent over time.

   Example: Regularly check the accuracy of transactions and system logs for anomalies.

8.4 Client-Server Testing

Client-server systems are a fundamental architecture in computing where multiple clients (users) connect to a central server to access resources and services. These systems are common in various applications ranging from web services to banking systems like ATMs. The complexity of client-server interactions, especially dynamic ones across multiple processors, makes testing these systems challenging. Ensuring reliable and secure communication between clients and servers, as well as verifying the correctness of operations performed by the server, is crucial for the robustness of the entire system.

? What is Client Server Testing ?

Client-server testing involves evaluating the interactions and communications between the client applications and the server. The goal is to ensure that requests from the client are correctly processed by the server, and the responses are accurately and efficiently returned to the client.

Characteristics of Client-Server Testing

The key characteristics of Client-Server Testing are:

1. **Functionality :** Ensuring that the server correctly processes client requests and performs the intended operations.
   - **Importance:** Functionality testing verifies that the client-server interactions work as expected. It ensures that all functions of the software application perform as specified in the requirements.
   - **Example:** In a banking application, functionality testing would check if the server correctly processes a client's request to transfer money between accounts.
2. **Performance :** Evaluating the response time and throughput of the server under various load conditions.
   - **Importance:** Performance testing assesses how well the client-server system performs under normal and peak conditions. It identifies bottlenecks and ensures the system can handle expected user loads.
   - **Example:** Testing how quickly a server can process multiple simultaneous requests for account balance inquiries from multiple clients.
3. **Security :** Verifying that data transmission between the client and server is secure and that the system can withstand potential threats.
   - **Importance:** Security testing ensures that the client-server communication is protected against unauthorized access and vulnerabilities. It checks for data encryption, authentication mechanisms, and other security protocols.
   - **Example:** Ensuring that user login credentials are transmitted securely using HTTPS and that the server is protected against SQL injection attacks.
4. **Concurrency :** Ensuring the system handles multiple simultaneous requests without data corruption or performance degradation.

- **Importance:** Concurrency testing checks the system's ability to handle multiple transactions at the same time. It ensures that the system remains stable and accurate when accessed by many users simultaneously.
- **Example:** In an e-commerce application, testing the server's ability to handle multiple users checking out their carts at the same time without causing errors in inventory management.

5. **Reliability :** Ensuring that the system remains stable and performs consistently over time.
   - **Importance:** Reliability testing assesses the system's ability to perform its functions under expected conditions without failure. It ensures that the system can be depended upon for continuous operation.
   - **Example:** Testing a server's uptime and error rate over a prolonged period to ensure it can handle continuous client requests without crashing.

Components of Client-Server Systems

Client-server systems consist of several key components that work together to deliver services and functionality to users. These components include:

1. **Server:**
   - **Role:** The server is the backbone of the client-server architecture. It hosts the central database management system (DBMS) and the application logic.
   - **Functions:** The server processes requests from clients, performs the necessary computations or data retrieval operations, and sends back the responses to the clients.
   - **Example:** In a banking system, the central server would handle tasks such as processing transactions, updating account balances, and maintaining customer records.
2. **Client:**
   - **Role:** The client is the interface through which users interact with the system. It runs the user interface (UI) and presentation logic.
   - **Functions:** The client sends requests to the server (e.g., querying account balances or submitting transaction requests) and displays the server's responses to the user.
   - **Example:** In the same banking system, the client would be the application on a user's smartphone or computer or ATM through which they access their account information and perform banking operations.
3. **Network:**
   - **Role:** The network is the medium that connects the clients to the server to enable communication between them.
   - **Functions:** It facilitates data transmission to ensure that requests and responses can travel back and forth between the client and server reliably and efficiently.
   - **Example:** The internet or a private organizational network (intranet) serves as the network connecting clients and servers in a banking system.

Types of Client-Server Systems

Client-server systems can be categorized based on where the majority of processing takes place: on the client side or the server side.

1. **Fat Client:**
   - **Description:** In a fat client architecture, most of the processing and computational tasks are performed on the client side. The server's primary role is to manage data storage and handle requests for data retrieval.
   - **Benefits:** This approach can reduce the load on the server since much of the processing is distributed to the clients. It can also lead to faster response times for the user since the client does not have to wait for the server to process every request.
   - **Drawbacks:** Fat clients can be more complex to maintain and update since the application logic resides on the client devices. This can lead to inconsistencies if clients are not uniformly updated.
   - **Example:** A graphics-intensive application where rendering and processing of images are done on the client device, while the server stores the image files and handles data requests.
2. **Fat Server:**
   - **Description:** In a fat server architecture, most of the processing is done on the server side. The client is primarily responsible for managing the user interface and handling input/output operations.
   - **Benefits:** This approach simplifies client devices since they only need to handle the presentation logic. It also centralizes the application logic, making maintenance and updates easier and more consistent.
   - **Drawbacks:** Fat server architectures can place a significant load on the server, especially if there are many clients. This can lead to scalability issues and the need for powerful server hardware.
   - **Example:** Web applications where the server handles all the business logic and database operations, while the client (web browser) simply renders the user interface and captures user input.

Challenges in Client-Server Testing

1. **Complex Interactions:** The dynamic interactions between clients and servers across multiple processors can be difficult to predict and test.
2. **Network Issues:** Network latency, packet loss, and bandwidth constraints can affect the performance and reliability of client-server communications.
3. **Concurrency:** Handling multiple simultaneous requests without data corruption or performance degradation requires robust concurrency control mechanisms.
4. **Security:** Ensuring secure data transmission and protecting against threats such as SQL injection, cross-site scripting, and man-in-the-middle attacks.

5. **Data Consistency:** Ensuring that data remains consistent across the client and server, especially in the face of concurrent updates.

Testing Strategy

1. **Identify Client-Server Interactions:** Determine all interactions that occur between the client and server, including data requests, processing, and responses.

   Example: In the SATM system, interactions between the ATM client (user interface) and the bank's central server (processing transactions) must be tested.

2. **Develop Test Cases:** Create test cases that cover different scenarios of client-server interactions, including normal operations, error handling, and edge cases.

   Example: A test case where the ATM client requests a balance inquiry, and the server processes this request and returns the current balance.

3. **Simulate Network Conditions:** Test under various network conditions, such as latency, packet loss, and bandwidth constraints, to ensure robustness.

   Example: Simulate a network delay to see how the ATM system handles slow responses from the central server.

4. **Concurrency Testing:** Ensure that the system can handle multiple simultaneous requests without data corruption or performance degradation.

   Example: Multiple ATMs simultaneously requesting transaction processing from the central server.

5. **Security Testing:** Ensure that data transmitted between the client and server is secure and that the system can withstand potential security threats.

   Example: Test for SQL injection, cross-site scripting (XSS), and secure data transmission protocols (e.g., HTTPS).

Execution and Monitoring

1. **Run Test Cases:** Execute the test cases in a controlled environment and monitor the system's responses.

   Example: Execute a withdrawal transaction test case to verify if the server correctly deducts the amount and updates the balance.

2. **Analyze Results:** Analyze the test results to identify any issues or inconsistencies in the client-server interactions.

   Example: Check if the server correctly handles multiple simultaneous balance inquiries without causing data inconsistencies.

3. **Fix Issues and Retest:** Correct any identified issues and perform retesting to ensure that the fixes are effective.

   Example: If an issue is found in handling concurrent transactions, fix the server logic and retest with multiple ATMs.

4. **Continuous Monitoring:** Continuously monitor the system to detect any issues that may arise during normal operations.

   Example: Monitor the ATM network for any signs of performance degradation or security breaches.

8.5. Review Questions

Section - A

Each Question Carries Two Marks

1. What is Interaction Testing ?
2. What are Taxonomy of interactions?
3. What are Static Interactions in a Single Processor ?
4. What are Static Interactions in Multiple Processors ?
5. What are Dynamic Interactions in a Single Processor ?
6. What are Dynamic Interactions in Multiple Processors ?
7. What is Client Server Testing ?
8. What is Fat Client?
9. What is Fat Server?

Section - B

Each Question Carries Five Marks

1. What is Interaction Testing ? Discuss the Key Aspects of Interaction Faults in System Testing.
2. Explain the Importance of Interaction Testing.
3. Explain the Features (or) Characteristics of Interaction Testing.
4. Write the Advantages and Disadvantages of Interaction Testing.
5. Explain the Context of Interaction in Interaction Testing.
6. What are Static Interactions in a Single Processor ? Explain the Key Concepts.
7. Explain the Procedure for Testing Static Interactions in a Single Processor.
8. What are Static Interactions in Multiple Processors ? Explain the Key Concepts.
9. Explain the Procedure for Testing Static Interactions in Multiple Processors.
10. What are Dynamic Interactions in a Single Processor ? Explain the Key Concepts.
11. Explain the Procedure for Testing Dynamic Interactions in a Single Processor.
12. What are Dynamic Interactions in Multiple Processors ? Explain the Key Concepts.
13. Explain the Procedure for Testing Dynamic Interactions in Multiple Processors.

14. What is Client Server Testing ? Write the Characteristics of Client-Server Testing.
15. Explain the Components and Types of Client-Server Systems.

Section - C

Each Question Carries Eight Marks

1. What is Interaction Testing ? Explain the Features and Importance of Interaction Testing.
2. Explain the Types of Interactions in System Testing.
3. Explain Static Interactions in a Single Processor with detailed example.
4. Explain Static Interactions in Multiple Processors with detailed example.
5. Explain Dynamic Interactions in a Single Processor with detailed example.
6. Explain Dynamic Interactions in Multiple Processors with detailed example.
7. What is Client Server Testing ? Explain the Testing Startegy.

❖❖❖❖❖

UNIT - IV

CHAPTER 9

OBJECT ORIENTED TESTING & GUI TESTING

Contents

9.1 Introduction to Object Oriented Testing

Testing of Object Oriented Software is different from testing software created using procedural languages. The most of the methods for testing Object Oriented Software were just a simple extension of existing methods for conventional software. However, they have been shown to be not very appropriate. Hence, new techniques have been developed.

Object-oriented programs involve many unique features that are not present in their conventional programs. Examples are message passing, synchronization, dynamic binding, object instantiation, persistence, encapsulation, inheritance, and polymorphism. Testing for such program is, therefore, more difficult than that for conventional programs. Object-orientation has rapidly become accepted as the preferred paradigm for large-scale system design. The whole object oriented testing revolves around the fundamental entity known as "class". With the help of "class" concept, larger systems can be divided into small well defined units which may then be implemented separately.

? What is Object Oriented Testing ?

Object-Oriented Testing is a software testing process that is conducted to test the software using object-oriented paradigms like, encapsulation, inheritance, polymorphism, etc. The software typically undergoes many levels of testing from unit testing to system testing. In simple words, **Object Oriented Testing** is a collection of testing techniques to verify and validate object oriented software.

Conventional testing defined for procedural programs do not fit well in the case of testing an object-oriented program. Conventional software testing tends to focus much on the algorithmic detail of a module and the data that flows across the module interface, whereas object-oriented software tends to focus on the operations that are encapsulated by the class and the state behavior of the class.

Several object-oriented features such as data abstraction, inheritance, polymorphism, dynamic binding etc., heavily impact on testing that is not straightforward to make object-oriented systems fit the conventional testing levels. There arises the need for object-oriented testing techniques which suits for object oriented system.

9.1.1 Conventional Testing Vs Object Oriented Testing

Conventional testing defined for procedural programs do not fit well in the case of testing an object-oriented program. The three levels of testing (unit testing, integration testing, system testing) used in conventional testing is not clearly defined when it comes to object oriented testing. The main reason for this is that OO development uses incremental approach, while traditional development follows a sequential approach. In terms of unit testing, object oriented testing looks at much smaller units compared to conventional testing. Let us see the major differences between conventional testing and object oriented testing,

Conventional Testing	Object Oriented Testing
Conventional testing is the traditional approach to testing mostly done when water fall life cycle is used for development.	Object oriented testing is used when object oriented analysis and design is used for developing enterprise software

Conventional testing focuses more on decomposition and functional approaches.	Object oriented testing focuses more on composition.
In conventional testing, the module or subroutine, or procedure are considered as a unit.	In object-oriented Testing, a class is considered as a unit.
A single operation of a procedure can be tested.	We cannot test a single operation in isolation but rather as part of a class.
It uses a sequential approach in the testing process.	It uses an incremental approach in the testing process.
This testing does not have any hierarchical control structure.	This testing has a hierarchical control structure.
The three levels of testing (system, integration, unit) used in conventional testing.	Object oriented testing also has the same levels of testing but the approach is different.

9.2 Issues in Object Oriented Testing

In this section, we want to discuss the testing issues that come up when working with object-oriented software. First, we need to figure out the different levels of testing needed and understand what object-oriented units are. Then, we will look at the effects of using composition instead of functional decomposition as a design approach. Object-oriented software includes features like inheritance, encapsulation, and polymorphism, so we will explore how traditional testing methods can be adapted to handle these complexities.

9.2.1 Units for Object-Oriented Testing

In object-oriented software development, the concept of a "unit" can be defined in a few different ways:

1. **Smallest Executable Component:** A unit is the smallest software component that can be compiled and executed.
2. **Single Developer Responsibility:** A unit is a software component that would be assigned to only one developer to design and implement.

These definitions can conflict. For example, some large classes in industrial applications are too complex to be designed by a single developer. In such cases, it is more practical to define a unit based on what a single person can handle, which might be just a subset of a class's operations or methods.

Simplified Unit Testing

For smaller units (like individual methods or small groups of methods), unit testing becomes similar to traditional testing. This approach simplifies testing but shifts much of the testing burden to integration testing. It also doesn't fully utilize the benefits of encapsulation, which is key in object-oriented programming.

Class-as-Unit Approach

In object-oriented software, defining a unit can be tricky due to varying sizes and complexities of classes. While smaller units simplify initial testing, treating whole classes as units can leverage state management tools and clear integration goals. This approach can improve the overall reliability and maintainability of the software. Treating an entire class as a unit for testing offers several advantages:

1. **State Management:** In UML (Unified Modeling Language), a class often has an associated StateChart that describes its behavior. This is very helpful for identifying test cases.
2. **Clear Integration Goals:** Object-oriented integration testing aims to ensure that separately tested classes work together correctly. This goal is similar to traditional software integration testing.

9.2.2 Implication of Composition and Encapsulation

Composition and encapsulation are fundamental principles in object-oriented design that provide flexibility and modularity. However, they also present challenges for testing due to complex interactions and hidden internal states. Let us understand the challenges in testing.

1. Composition

Composition involves building a class using objects of other classes, establishing a "has-a" relationship. This principle enhances flexibility and reuse in software design.

Example Composition - A Car class composed of Engine, Transmission, and GPS classes.

```
public class Car {
    private Engine engine;
    private Transmission transmission;
    private GPS gps;

    public Car(Engine engine, Transmission transmission, GPS gps) {
        this.engine = engine;
        this.transmission = transmission;
        this.gps = gps;
    }

    public void start() {
        engine.start();
        transmission.setGear(1);
        gps.initialize();
        System.out.println("Car started.");
    }

    public void stop() {
        engine.stop();
        transmission.setGear(0);
        gps.shutdown();
        System.out.println("Car stopped.");
    }
```



```
    public boolean isReady() {
        return engine.isRunning() && gps.isInitialized() && transmission.getGear() > 0;
    }

    // Additional methods...
}
```

This Car class composes the Engine, Transmission, and GPS classes. It provides methods to start and stop the car, which involve starting/stopping the engine, setting the transmission gear, and initializing/shutting down the GPS. It also includes a method to check if the car is ready (i.e., the engine is running, the GPS is initialized, and the transmission is in gear).

This example demonstrates composition by having the Car class depend on instances of Engine, Transmission, and GPS classes. It also encapsulates the behavior of starting and stopping the car, providing a clean and controlled interface.

Implications of Composition

1. **Complex Interactions:**
   - **Challenge:** Ensuring correct interactions between composed objects.
     Testing the Car involves ensuring the Engine, Transmission, and GPS interact correctly.
   - **Example:** In the Car class, starting the car involves the interaction of the Engine, Transmission, and GPS. The Engine must start, the Transmission must be set to the appropriate gear, and the GPS must be initialized. Testing must verify that these interactions occur correctly and in the right order.
2. **Unknown Combinations:**
   - **Challenge:** Handling various configurations and versions of composed objects.
   - **Example:** The Car class might use different models of Engine, Transmission, or GPS systems, each with unique behaviors or interfaces. Ensuring compatibility and proper functionality with all possible combinations is challenging.
3. **Dependency Management:**
   - **Challenge:** Managing dependencies between the composed objects can be tricky, especially when testing in isolation.
   - **Example:** The Car class depends on Engine, Transmission, and GPS. Each dependency introduces potential points of failure and complexity in managing their interactions.

Testing Strategies

1. **Unit Testing:**
   - **Approach:** Test each component independently to ensure they work correctly in isolation.
   - **Example:** Testing the Engine alone to ensure it starts correctly.

2. **Mocking Dependencies:**
   - **Approach:** Use mock objects to simulate the behavior of dependent classes when testing the composed class.
   - **Example:** Creating mock objects for Engine, Transmission, and GPS when testing the Car to isolate the unit test.
3. **Integration Testing:**
   - **Approach:** Test the composed class with actual instances of its components to ensure they interact correctly.
   - **Example:** Testing the Car with a real Engine, Transmission, and GPS to verify the entire start-up sequence.

2. Encapsulation

Encapsulation is a core principle of object-oriented programming that involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting access to some of the object's components, which means that the internal representation of an object is hidden from the outside. This is achieved through access modifiers like private, protected, and public.

Example **Encapsulation - A Car class composed of Engine, Transmission, and GPS classes.**

The Car class in previous example demonstrates the principle of encapsulation by bundling together data (attributes) and methods (functions) that operate on that data. Encapsulation ensures that the internal workings of the Car class are hidden from the outside world, providing a controlled interface for interacting with its components (Engine, Transmission, and GPS).

- The Car class has private fields for its components: Engine, Transmission, and GPS. These fields are not directly accessible from outside the class. By making these fields private, the Car class controls how these components are used and interacted with.
- The Car class provides public methods start and stop to control the behavior of the car.These methods encapsulate the complex interactions required to start and stop the car, ensuring that the operations are performed in the correct sequence.

Implications of Encapsulation:

1. **Accessing Private Data:**
   - **Challenge:** Encapsulation hides the internal state, making it difficult to test the internal logic directly.
   - **Example:** The Car class hides its components (Engine, Transmission, GPS), making it hard to directly verify their states during testing.
2. **Ensuring Correct Interface:**
   - **Challenge:** Tests must ensure that the public interface works correctly and that internal state changes as expected through this interface.
   - **Example:** Ensuring that the start method of the Car class correctly initializes all components through the public interface.

3. **Hidden Dependencies:**
   - **Challenge:** Encapsulated components might have hidden dependencies that are not obvious, complicating testing efforts.
   - **Example:** The Car class may have implicit dependencies on the order of operations (e.g., the engine must start before the transmission can be engaged).

Testing Strategies

1. **Public Interface Testing:**
   - **Approach:** Focus on testing the public methods and ensuring they manipulate the internal state correctly.
   - **Example:** Testing the start method of the Car to ensure it correctly initializes the engine, transmission, and GPS.
2. **Behavioral Testing:**
   - **Approach:** Ensure that the class behaves correctly in all scenarios that the public interface should support.
   - **Example:** Verifying that the Car can start, drive, and stop as expected through its public methods.
3. **Integration Testing with Real Components:**
   - **Approach:** Use real components instead of mocks to test how the class interacts with its dependencies in a realistic setting.
   - **Example:** Testing the Car with an actual Engine, Transmission, and GPS to see how it behaves in real-world conditions.

9.2.3 Implication of Inheritance

Inheritance allows classes to inherit attributes and methods from other classes, promoting code reuse and logical hierarchy. However, it introduces complexities in testing, especially regarding the behavior of subclasses and their interaction with superclass components.

Consider a banking system with a base class Account and derived classes Checking Account and Savings Account.

Example **Banking System with Inheritance**

```
public class Account {
    protected String accountNumber;
    protected double balance;

    public Account(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }
```

```
    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}
public class CheckingAccount extends Account {
    private double checkProcessingCharge;

        public CheckingAccount(String accountNumber, double balance, double
        checkProcessingCharge) {
        super(accountNumber, balance);
        this.checkProcessingCharge = checkProcessingCharge;
    }

    public void processCheck(double amount) {
        withdraw(amount + checkProcessingCharge);
    }
}

public class SavingsAccount extends Account {
    private double interestRate;

    public SavingsAccount(String accountNumber, double balance, double interestRate) {
        super(accountNumber, balance);
        this.interestRate = interestRate;
    }

    public void addInterest() {
        balance += balance * interestRate;
    }
}
```

Implications of Inheritance

1. **Flattening Classes**
   - **Challenge:** Flattening involves merging all attributes and methods from superclasses into subclasses. It complicates testing due to increased complexity.
   - **Example:** Testing CheckingAccount would involve ensuring all inherited methods from Account behave correctly in the context of the subclass.

2. **Special-Purpose Test Methods**
   - **Challenge:** Adding special-purpose test methods for inherited attributes and methods raises the issue of maintaining these methods and ensuring they are not part of the final system.
   - **Example:** Creating test methods specifically for withdraw in CheckingAccount might introduce redundancy and confusion.

Testing Strategies

1. **Unit Testing of Inherited Methods**
   - **Approach:** Test inherited methods in the context of the subclass to ensure they behave correctly.
   - **Example:** Testing the withdraw method in both Account and CheckingAccount to ensure correct behavior.
2. **Integration Testing**
   - **Approach:** Ensure that interactions between subclasses and their superclasses work as expected.
   - **Example:** Testing CheckingAccount and SavingsAccount together to verify they handle common Account operations correctly.

9.2.4 Implication of Polymorphism

Polymorphism is one of the core concepts of object-oriented programming (OOP) that allows methods to be used interchangeably across different classes, as long as those classes share the same interface or base class. The term "polymorphism" means "many shapes" and it refers to the ability of different classes to be treated as instances of the same class through a common interface.

There are two main types of polymorphism in OOP:

1. **Compile-time (Static) Polymorphism:** Achieved through method overloading and operator overloading.
2. **Runtime (Dynamic) Polymorphism:** Achieved through method overriding, typically using inheritance and interfaces.

Consider a Shape interface with an area method. Different shapes like Circle and Square will implement this interface, providing their own specific calculations for the area.

Example Polymorphism

```
public abstract class Shape {
    public abstract double area();
}

public class Circle extends Shape {
    private double radius;
```

```
    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

public class Square extends Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        Shape square = new Square(4.0);

        System.out.println("Area of Circle: " + circle.area());
        System.out.println("Area of Square: " + square.area());
    }
}
```

Implications of Polymorphism

Testing All Implementations

- **Challenge:** Ensuring that all implementations of a polymorphic method behave correctly requires thorough testing.
- **Example:** Testing the area method in both Circle and Square to ensure it calculates the area correctly for different shapes. Comprehensive unit tests are necessary for each class that implements the polymorphic method.

Testing Strategies:

- Unit Testing of Polymorphic Methods: Test each implementation independently.
- Polymorphic Testing: Use a common interface to verify that all implementations behave correctly.

9.3 Levels of Object-Oriented Testing

Testing object-oriented software can be categorized into different levels, each focusing on specific aspects of the system.

1. **Unit Testing :** Unit testing in object-oriented systems involves testing individual units or components of the system in isolation to ensure they function correctly.
   a. **Operation/Method Testing:** Test individual methods or operations to ensure they perform their intended functions correctly.
   b. **Class Testing:** Test a class in isolation, including all its methods and interactions with internal components.
2. **Integration Testing:** Test interactions between classes to ensure they work together as expected.
3. **System Testing:** Test the entire system in an environment that closely resembles production to ensure it meets all specified requirements.

9.4 Object Oriented Unit Testing

Unit testing in object-oriented systems involves testing individual units or components of the system in isolation to ensure they function correctly. Unit testing focuses on testing the smallest testable parts of a software application, known as units or components. In object-oriented programming, classes are often considered as units for testing. The purpose of unit testing is to validate the behavior of individual classes, methods, or functions to ensure they meet the specified requirements.

Unlike procedural programming, object-oriented programming introduces concepts such as composition, encapsulation, inheritance, and polymorphism, which create unique challenges and opportunities for testing. Effective unit testing helps identify issues early, ensures that each part of the system behaves as expected, and facilitates easier integration and maintenance.

1. **Unit Testing of Composition :** Composition involves building a class using objects of other classes, establishing a "has-a" relationship. This principle enhances flexibility, modularity, and reuse in software design.

   Steps:

   (a) Identify Composed Classes:
   - Determine which classes are composed within another class.
   - Example: A Car class composed of Engine, Transmission, and GPS classes.

   (b) Test Each Component Class Independently:
   - Create unit tests for each component class to ensure they function correctly on their own.
   - Example: Test the Engine class to ensure it starts and stops correctly.

   (c) Ensure the Composed Class Correctly Initializes and Interacts with Its Components:
   - Write tests for the composed class to verify it initializes its components and that the components interact correctly.
   - Example: Test the Car class to ensure it correctly starts the engine, sets the transmission, and initializes the GPS.

2. **Unit Testing of Encapsulation :** Encapsulation involves bundling data and methods within a single class and restricting access to some components, usually with private access modifiers. This ensures that the internal state of the object is hidden from the outside and can only be manipulated through well-defined interfaces.

   Steps:

   (a) Focus on Testing Public Methods:

   - Write tests for the public methods of the class, as these are the primary means of interacting with the object.
   - Example: Test the start and stop methods of the Car class.

   (b) Ensure Public Methods Manipulate Internal State Correctly:

   - Verify that the public methods correctly manipulate the internal state of the object.
   - Example: Ensure that the start and stop methods correctly change the states of the Engine, Transmission, and GPS components.

3. **Unit Testing of Inheritance :** Inheritance allows a class to inherit methods and attributes from another class, promoting code reuse and logical hierarchy.

   Steps:

   (a) Test the Base Class:

   - Ensure that the base class's methods work correctly.
   - Example: Test the deposit and withdraw methods in the Account class.

   (b) Test Derived Classes to Ensure Correct Inheritance and Method Overriding:

   - Verify that the derived classes correctly inherit methods from the base class and override them as necessary.
   - Example: Ensure that CheckingAccount correctly applies a fee when withdraw is called.

4. **Unit Testing Polymorphism :** Polymorphism allows methods to be used interchangeably across different classes, provided they share the same interface or base class. This enhances flexibility but complicates testing due to the need to verify that all polymorphic implementations behave correctly.

   Steps:

   (a) Test Each Implementation of the Polymorphic Method Independently:

   - Ensure that each class implementing the polymorphic method works correctly.
   - Example: Test the area method in both Circle and Square classes.

   (b) Use Polymorphism in Tests to Verify All Implementations Behave Correctly Through a Common Interface:

   - Write tests that use the base class or interface to ensure that different implementations work as expected.
   - Example: Use the Shape interface to test the area method in both Circle and Square.

9.5 Object-Oriented Integration Testing

Integration testing is a crucial phase in software development, especially in object-oriented systems, where the goal is to ensure that different classes and their methods interact correctly after individual unit testing is complete. This process involves combining tested units (classes and methods) into larger modules and testing them as a group. Integration testing can be challenging due to the complexities introduced by object-oriented principles such as encapsulation, inheritance, and polymorphism. We will explore various integration testing strategies.

1. Operation/Method Integration Testing :

When the operation/method is chosen as the unit for testing, two levels of integration are required:

- Integrate methods into a full class.
- Integrate the class with other classes.

This strategy is used when classes are very large, and several designers are involved in developing them. The goal is to ensure that individual methods work correctly within the context of the class and that the class interacts properly with other classes.

Example **Operation/Method Integration Testing**

Consider an e-commerce application with a Cart class and a Product class. Each method in the Cart class, such as addItem and removeItem, would be integrated into the full Cart class. Then, the Cart class would be integrated with the Product class.

```
public class Cart {
    private List<Product> items;

    public Cart() {
        items = new ArrayList<>();
    }

    public void addItem(Product product) {
        items.add(product);
    }

    public void removeItem(Product product) {
        items.remove(product);
    }

    public int getTotalItems() {
        return items.size();
    }
}

public class Product {
    private String name;
    private double price;
```

```
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Getters and setters...
}
```

Explanation

- Method to Class Integration:
  - Ensure that addItem and removeItem methods correctly modify the items list.
  - Verify through unit tests that each method behaves as expected when called independently.
- Class to Class Integration:
  - Ensure that Cart interacts correctly with Product, such as verifying that a Product added to the Cart can be retrieved and has correct properties.
  - Integration tests should validate that Cart correctly handles Product objects and maintains the correct state.

Integration Test Code

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CartIntegrationTest {
    @Test
    public void testAddItem() {
        Cart cart = new Cart();
        Product product = new Product("Laptop", 1000.00);
        cart.addItem(product);
        assertEquals(1, cart.getTotalItems());
    }

    @Test
    public void testRemoveItem() {
        Cart cart = new Cart();
        Product product = new Product("Laptop", 1000.00);
        cart.addItem(product);
        cart.removeItem(product);
        assertEquals(0, cart.getTotalItems());
    }
}
```

2. Class-as-Unit Integration Testing :

For the class-as-unit choice, once unit testing is complete, two steps must occur:

- Restore the original class hierarchy if flattened classes were used.
- Remove any test methods that were added during unit testing.

This strategy is more common and focuses on ensuring that fully tested classes work correctly when integrated with other classes. It ensures that interactions between different parts of the system function as intended.

Example: Consider a library management system with Library, Book, and Member classes. After unit testing these classes individually, integration testing will focus on interactions between these classes

Example **Class-as-Unit Integration Testing**

```
public class Library {
    private List<Book> books;
    private List<Member> members;

    public Library() {
        books = new ArrayList<>();
        members = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public void registerMember(Member member) {
        members.add(member);
    }

    // Other methods...
}

public class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    // Getters and setters...
}

public class Member {
    private String name;

    public Member(String name) {
        this.name = name;
    }

    // Getters and setters...
}
```

Explanation:

Class to Class Integration:

- Ensure that Library can add Book and Member objects and interact with them correctly.
- Integration tests should verify that the Library class can handle the addition of books and members, and that these objects maintain their state and functionality within the Library.

Integration Test Code:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class LibraryIntegrationTest {
    @Test
    public void testAddBook() {
        Library library = new Library();
        Book book = new Book("Effective Java", "Joshua Bloch");
        library.addBook(book);
        // Verify that the book is added correctly
        assertTrue(library.getBooks().contains(book));
    }

    @Test
    public void testRegisterMember() {
        Library library = new Library();
        Member member = new Member("John Doe");
        library.registerMember(member);
        // Verify that the member is registered correctly
        assertTrue(library.getMembers().contains(member));
    }
}
```

3. UML Support for Integration Testing

Unified Modeling Language (UML) provides valuable tools for visualizing and designing object-oriented systems, which can also be leveraged for integration testing. UML diagrams, such as collaboration and sequence diagrams, help identify and understand the interactions between classes and methods, forming a basis for creating comprehensive integration tests.

1. **Collaboration Diagrams :** A collaboration diagram, also known as a communication diagram, shows the message traffic among classes, similar to a call graph. It depicts how objects interact to perform a specific behavior. Collaboration diagrams are useful for pairwise integration testing, where each class is tested in terms of adjacent classes that send or receive messages.

   Example: Consider a social media application with User, Post, and Comment classes. A collaboration diagram might show interactions between these classes:

   - User creates Post
   - Post has Comment
   - User likes Post

Pairwise Integration Testing:

- User and Post, with stubs for Comment.
- Post and Comment, with stubs for User.

For pairwise integration, a class is tested in terms of separate adjacent classes. For example, integrating User and Post while using stubs for Comment to ensure that interactions between User and Post work as expected.

Example **Pairwise Integration Testing**

```
import static org.junit.Assert.*;
import org.junit.Test;

public class UserPostIntegrationTest {
    @Test
    public void testUserCreatesPost() {
        User user = new User("John");
        Post post = new Post("Hello World!");

        user.createPost(post);

        assertTrue(user.getPosts().contains(post));
    }

    @Test
    public void testPostHasComment() {
        Post post = new Post("Hello World!");
        Comment comment = new Comment("Nice post!");

        post.addComment(comment);

        assertTrue(post.getComments().contains(comment));
    }
}
```

Explanation

Pairwise integration testing involves testing the interactions between pairs of classes that directly communicate with each other while using stubs for other related classes. This approach ensures that the adjacent classes' interactions are verified independently of the complete system, thus isolating and identifying issues more effectively. For example, in the provided UserPostIntegrationTest, the interaction between the User and Post classes is tested to ensure that a User can successfully create a Post and that a Post can successfully have a Comment added. The Comment class is used as a stub when testing the interaction between User and Post, and vice versa. This method simplifies the integration process by focusing on one pair of interacting classes at a time, ensuring their interaction works correctly before they are integrated into the broader system.

In the UserPostIntegrationTest:

1. **Testing User Creates Post:**
   - The test testUserCreatesPost verifies that a User object can create a Post object.
   - A new User named "John" and a Post with the content "Hello World!" are instantiated.

- The user.createPost(post) method is called, and the test asserts that the post is successfully added to the user's list of posts.

2. **Testing Post Has Comment:**
   - The test testPostHasComment verifies that a Post object can have a Comment added.
   - A new Post with the content "Hello World!" and a Comment with the content "Nice post!" are instantiated.
   - The post.addComment(comment) method is called, and the test asserts that the comment is successfully added to the post's list of comments.

This approach confirms that the User can manage Post objects and that Post objects can manage Comment objects independently, ensuring that these interactions function correctly before the complete system is tested.

2. **Sequence Diagrams :** A sequence diagram traces the execution path through a collaboration diagram, showing the sequence of messages sent between objects over time. It provides a dynamic view of interactions and can be used to create detailed integration tests.

Example: For a payment processing system with Payment, Order, and Invoice classes, a sequence diagram might show the process of creating an invoice after a payment is made:

- Create a Payment.
- Associate the Payment with an Order.
- Generate an Invoice for the Order.

A sequence diagram shows the order of interactions between objects. For the payment processing system, the sequence diagram illustrates how a Payment triggers the creation of an Order, which then leads to the generation of an Invoice.

Example **Sequence Diagram-Based Integration Testing**

```
import static org.junit.Assert.*;
import org.junit.Test;

public class PaymentProcessingTest {
    @Test
    public void testPaymentProcessing() {
        Payment payment = new Payment(100.00);
        Order order = new Order("Order123");
        Invoice invoice = new Invoice(order);

        order.setPayment(payment);
        invoice.generate();

        // Verify the invoice details
        assertEquals("Order123", invoice.getOrder().getOrderId());
        assertEquals(100.00, invoice.getAmount(), 0.01);
    }
}
```


Explanation:

The PaymentProcessingTest illustrates sequence diagram-based integration testing by validating the interactions between Payment, Order, and Invoice classes in a payment processing system. This test follows the sequence of operations typically depicted in a sequence diagram: a Payment is created, associated with an Order, and then used to generate an Invoice. The test ensures that the Order correctly records the Payment, and that the Invoice accurately reflects the details of the Order. Specifically, it verifies that the Invoice is associated with the correct Order ID and that the payment amount is accurately reflected in the Invoice. This approach ensures that the sequence of operations works as expected, validating the integration of these classes in a real-world scenario.

In the PaymentProcessingTest:

1. **Creating and Associating Objects:**
   - A new Payment of 100.00 is created.
   - An Order with the ID "Order123" is instantiated.
   - An Invoice linked to the Order is created.
2. **Setting Payment and Generating Invoice:**
   - The Payment is associated with the Order using order.setPayment(payment).
   - The Invoice is generated by calling invoice.generate().
3. **Verification:**
   - The test asserts that the Invoice is correctly linked to the Order with the ID "Order123".
   - It also verifies that the Invoice amount matches the payment amount of 100.00, with an allowable margin of error of 0.01.

9.6 GUI Testing

Graphical User Interface (GUI) testing involves testing the user interface of an application to ensure it functions correctly and provides a positive user experience. It is a crucial aspect of software testing, focusing on the front-end aspect of an application, which interacts directly with users. GUI testing aims to validate that all visual elements, such as buttons, menus, icons, and dialogs, work as expected and that the application behaves correctly in response to user interactions.

Key Aspects of GUI Testing

1. **Functional Validation:** Verify that all GUI elements perform their intended functions, such as buttons triggering actions or forms accepting input.
2. **Layout and Design:** Check the alignment, spacing, fonts, colors, and overall aesthetics of the GUI for consistency and adherence to design guidelines.
3. **Navigation and Interactivity:** Test the flow of the GUI, including menu navigation, links, pop-ups, and interactive elements like sliders or drag-and-drop features.
4. **Error Handling:** Validate how the GUI responds to user errors, such as displaying appropriate error messages or guiding users to correct input.

9.6.1 Key Objectives of GUI Testing

1. **Functionality Testing:**
   - Ensure that all GUI elements perform their intended functions.
   - Verify that buttons, links, menus, and other controls trigger the correct actions.
2. **Usability Testing:**
   - Assess the user-friendliness and intuitiveness of the application.
   - Ensure that the interface is easy to navigate and understand.
3. **Consistency Testing:**
   - Verify that the GUI follows a consistent design pattern.
   - Ensure that visual elements are uniformly styled and behave consistently across different screens and resolutions.
4. **Compatibility Testing:**
   - Ensure that the GUI works correctly across different devices, operating systems, browsers, and screen sizes.
   - Check for responsive design and adaptability to various environments.
5. **Performance Testing:**
   - Evaluate the responsiveness of the GUI.
   - Ensure that actions are performed within acceptable time limits and the interface remains responsive under load.
6. **Accessibility Testing:**
   - Verify that the GUI is accessible to users with disabilities.
   - Ensure compliance with accessibility standards such as WCAG (Web Content Accessibility Guidelines).

9.6.2 Types of GUI Testing

1. **Manual Testing:**
   - Testers manually interact with the application to validate GUI functionality.
   - Useful for exploratory testing and identifying usability issues that automated tests might miss.
2. **Automated Testing:**
   - Uses automated tools to execute predefined test scripts on the GUI.
   - Efficient for repetitive and regression testing.
3. **Exploratory Testing:**
   - Testers explore the GUI dynamically to uncover unexpected issues and assess the overall user experience.

9.6.3 Examples of GUI Testing

1. **Login Page Testing:** Verify that the login form accepts valid credentials, displays error messages for invalid inputs, and redirects users to the correct page upon successful login.
2. **E-commerce Checkout Testing:** Test the checkout process for an online store, including adding items to the cart, entering shipping details, and processing payments.
3. **Mobile App GUI Testing:** Validate the responsiveness and layout of a mobile app on different devices and screen sizes to ensure a consistent user experience.
4. **Dashboard Testing:** Test the functionality of a dashboard interface, including data visualization, filtering options, and drill-down capabilities.

9.6.4 Tools for GUI Testing:

1. **Selenium:** A popular open-source tool for automating web application testing, including GUI testing.
2. **TestComplete:** A comprehensive GUI testing tool that supports desktop, web, and mobile applications.
3. **Applitools:** A visual testing tool that automates GUI validation for layout, design, and content changes.

9.6.5 GUI Testing Strategies

GUI testing strategies focus specifically on testing the graphical user interface of a software application to ensure its functionality, usability, and visual appeal. Here are some GUI testing strategies along with explanations:

1. **Functional Testing :**
   - **Objective:** Verify that all GUI elements work as intended and perform the specified functions.
   - **Approach:** Testers interact with buttons, menus, input fields, and other UI components to validate their behavior.
   - **Examples:** Clicking buttons, entering text in input fields, selecting options from dropdown menus, and verifying form submissions.
2. **Usability Testing:**
   - **Objective:** Evaluate the user-friendliness and intuitiveness of the GUI design.
   - **Approach:** Real users or testers assess the layout, navigation, responsiveness, and overall user experience of the interface.
   - **Examples:** Testing navigation flow, assessing color schemes, checking font sizes, and ensuring consistency in design elements.

3. **Compatibility Testing:**
   - **Objective:** Ensure that the GUI displays correctly and functions properly across different devices, browsers, and screen sizes.
   - **Approach:** Test the application on various platforms and configurations to identify compatibility issues.
   - **Examples:** Testing on different browsers (Chrome, Firefox, Safari), devices (desktop, mobile), and operating systems (Windows, macOS, iOS, Android).
4. **Localization Testing:**
   - **Objective:** Verify that the GUI adapts to different languages, regions, and cultural preferences.
   - **Approach:** Testers validate text translations, date formats, currency symbols, and other localized elements.
   - **Examples:** Testing language support, checking alignment of translated text, and ensuring proper display of special characters.
5. **Accessibility Testing:**
   - **Objective:** Ensure that the GUI is accessible to users with disabilities and complies with accessibility standards.
   - **Approach:** Testers assess the interface for screen reader compatibility, keyboard navigation, color contrast, and other accessibility features.
   - **Examples:** Testing with screen readers, keyboard-only navigation, checking alt text for images, and verifying color contrast ratios.
6. **Cross-Browser Testing:**
   - **Objective:** Validate that the GUI functions consistently across different web browsers.
   - **Approach :** Test the application on multiple browsers to identify rendering issues, layout discrepancies, and functional inconsistencies.
   - **Examples:** Testing on Chrome, Firefox, Edge, Safari, and ensuring compatibility with older browser versions.
7. **GUI Automation Testing:**
   - **Objective:** Automate GUI tests to improve efficiency, coverage, and regression testing.
   - **Approach:** Use tools like Selenium, TestComplete, or Appium to automate test scripts for GUI interactions.
   - **Examples:** Writing test scripts to automate button clicks, form submissions, validations, and UI interactions.

By implementing a combination of these GUI testing strategies based on the specific requirements and characteristics of the software application, teams can ensure that the graphical user interface meets quality standards, provides a seamless user experience, and functions reliably across different environments.

9.7 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Object Oriented Testing ?
2. How Unit Testing is done in Object Oriented Software?
3. What is GUI Testing?
4. Mention the types of GUI Testing.
5. Mention any two tools for GUI Testing.

Section - B

Each Question Carries Five Marks

1. Differentiate Between Conventional Testing Vs Object Oriented Testing.
2. Explain the Implication of Composition and Encapsulation in Software Testing.
3. Explain the Implication of Inheritance in Software Testing.
4. Explain the Implication of Polymorphism in Software Testing.
5. What is GUI Testing? Explain the Objectives of GUI Testing.
6. Explain GUI Testing Strategies.

Section - C

Each Question Carries Eight Marks

1. Explain the Issues in Object Oriented Testing.
2. Explain and Eloborate on Object Oriented Unit Testing.
3. Explain and Eloborate on Object-Oriented Integration Testing
4. Explain and Eloborate on GUI Testing.

❖❖❖❖❖

Note

UNIT - IV

CHAPTER 10

EXPLORATORY TESTING & MODEL BASED TESTING

Contents

10.1 Exploratory Testing

Consider a person admitted to a hospital emergency room (ER) who has trouble in breathing. The ER physician is tasked with identifying the underlying problem and then devising a medical response. How does the ER physician proceed? First, a case history of relevant information about the patient is gathered. The next likely step is a few broad-spectrum tests that are intended to eliminate common causes of the breathing difficulty. Knowledge obtained from one test usually leads to follow-up, more specific tests. Throughout this process, the ER physician is guided by extensive experience and domain knowledge. This same pattern applies to software testing in the process known as exploratory testing.

This structured methodology mirrors the principles of exploratory testing in software testing, where testers navigate through the application, leveraging their expertise to uncover issues, iterate on testing strategies, and ensure the software's quality and functionality.

10.1.1 What is Exploratory Testing ?

? What is Exploratory Testing ?

Exploratory testing is an approach to software testing where testers simultaneously design and execute test cases based on their domain knowledge, experience, and intuition. Unlike traditional scripted testing, exploratory testing involves testers exploring the software application dynamically, without predefined test cases, Testers learn about the application as they test, uncovering defects, understanding functionalities, and identifying potential risks through an iterative and explorative approach. The goal of exploratory testing is to discover issues that might not be found through structured testing techniques.

Example Understanding Exploratory Testing

An example of exploratory testing could involve a tester tasked with testing a new e-commerce website. Instead of following predefined test cases, the tester explores the website by navigating through different pages, adding items to the cart, and simulating various user interactions like creating an account, making a purchase, and updating personal information. During this exploration, the tester may encounter issues such as incorrect pricing display, broken links, or usability issues like confusing navigation. By dynamically adjusting their testing based on these findings and documenting the process, the tester can provide valuable feedback to improve the website's quality before its release.

10.1.2 The Context-Driven School

The Context-Driven School of thought as developed by Cem Kaner, James Bach, and Bret Pettichord, emphasizes understanding the specific context of a testing situation and applying the most effective practices based on that understanding. This philosophy acknowledges that there is no one-size-fits-all approach to testing and that what works in one situation may not be ideal in another.

The Principles of the Context-Driven School:

1. **The value of any practice depends on its context:** This principle recognizes that the effectiveness of a testing practice is influenced by the specific context in which it is applied.

2. **There are good practices in context, but there are no best practices:** This principle Emphasizes that testing practices should be evaluated based on their suitability for the given context rather than assuming a universal "best" practice.
3. **People, working together, are the most important part of any project's context:** This principle highlights the significance of collaboration and teamwork in testing projects.
4. **Projects unfold over time in ways that are often not predictable:** This principle acknowledges the dynamic and evolving nature of software projects, requiring adaptability in testing approaches.
5. **The product is a solution:** This principle explains that the ultimate goal of software is to solve a problem, and testing should ensure that the product meets this objective.
6. **Good software testing is a challenging intellectual process:** This principle recognizes testing as a complex and intellectually demanding activity that requires critical thinking and skill.
7. **Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products:** This principle highlights the importance of judgment, skill, and collaboration in conducting effective testing throughout the project lifecycle.

Exploratory Testing aligns with the principles of the Context-Driven School by encouraging testers to be investigative, adaptive, and context-aware in their testing approach. Testers leverage their knowledge of the product and context to make informed decisions about what to test next, reflecting the principles of adaptability, collaboration, and the value of human testers emphasized by the Context-Driven School.

10.1.3 Exploring Exploratory Testing

Exploratory testing is a dynamic and informal approach to software testing where testers actively engage with the software to uncover defects and learn about its behavior simultaneously. This method emphasizes simultaneous test design and execution, making it highly interactive, creative, and adaptive. Andy Tinkham and Cem Kaner, prominent figures in this field, describe five essential characteristics of exploratory testing

Characteristics of Exploratory Testing

1. **Interactive:** Exploratory testing involves real-time interaction with the software. Testers actively engage with the application, exploring its functionalities and behavior.

   Example: A tester navigates through a new social media app, trying out various features like posting updates, sending messages, and changing privacy settings to identify potential issues.
2. **Concurrent Cognition and Execution:** Testers think and test simultaneously, learning about the system while interacting with it. This allows them to adjust their testing strategy based on immediate feedback.

   Example: While testing a shopping cart feature, a tester might notice an issue with item quantities updating incorrectly and immediately focus on testing different scenarios related to this behavior.

3. **Highly Creative:** Exploratory testing requires creativity and intuition. Testers use their skills and experience to identify potential problem areas and design tests on the fly.

   Example: A tester creatively manipulates input fields on a form by entering various types of data (e.g., special characters, long strings) to see how the system handles them.

4. **Quick Results:** The goal is to quickly uncover defects and issues. Exploratory testing is often used when there is a need for rapid feedback or when time constraints limit the ability to perform extensive scripted testing.

   Example: During a tight release cycle, testers use exploratory testing to quickly assess the stability of new features in a mobile app update.

5. **Reduced Emphasis on Formal Documentation:** Unlike traditional testing, exploratory testing relies less on predefined test cases and detailed documentation. Instead, testers document their findings as they go.

   Example: A tester explores a new feature and notes any bugs or issues directly in a shared document or bug tracking system without following a strict test case template.

Examples **Exploratory Testing in Various Scenarios**

Example 1: E-Commerce Application

Context: A team is working on an e-commerce application with a tight deadline. The focus is on quickly identifying critical defects in the checkout process.

Testing Approach:

The tester starts by adding various products to the cart, checking for issues with product details, prices, and cart updates.

They proceed to the checkout process, entering different types of payment information and shipping addresses to test validation and error handling.

The tester also explores edge cases, such as applying multiple discount codes and attempting to purchase out-of-stock items.

Outcome: The tester discovers several issues, including incorrect price calculations with multiple discounts, validation errors with international addresses, and crashes when trying to purchase more items than available.

Example 2: Medical Device Software

Context: The software controls a medical device and must comply with strict regulatory standards. Thorough testing is required, but time is also a factor.

Testing Approach:

The tester explores the user interface, checking how the software responds to various user inputs and device settings.

They simulate different patient scenarios, adjusting device parameters and monitoring responses.

The tester documents findings in real-time, focusing on potential safety issues and compliance with regulatory requirements.

Outcome: The tester identifies critical issues, such as incorrect parameter settings under specific conditions and inconsistencies in the user interface that could lead to user errors.

What is Lewis and Clark's Expedition ?

The Lewis and Clark Expedition, also called the Corps of Discovery Expedition, was a big exploration trip led by Captain Meriwether Lewis and Second Lieutenant William Clark. It was ordered by President Thomas Jefferson after the U.S. bought the Louisiana Territory in 1803. The main goals were to explore and map the new land, claim it for the U.S. before others could, study its resources and geography, and find a way to the Pacific Ocean.

Starting in May 1804 from St. Louis, Missouri, the team traveled up the Missouri River, facing tough weather, rough land, and meeting Native American tribes. They carefully recorded their discoveries, making maps, studying plants and animals, and learning about the Native peoples.

Sacagawea, a Shoshone woman, played a crucial role as a guide and translator, using her knowledge and connections with tribes.

They reached the Pacific Ocean in November 1805 and returned to St. Louis in September 1806. Their findings greatly expanded America's knowledge of the West and encouraged further exploration. The expedition is a famous part of American history, representing exploration, learning, and bravery.

Comparison of Exploratory Testing and the Lewis and Clark Expedition

Exploratory testing is a dynamic and adaptive approach to software testing where testers investigate the software's functionalities and behavior in real-time, similar to how explorers navigate uncharted territories. This method can be likened to the historic Lewis and Clark Expedition, a significant journey of discovery and documentation across the newly acquired western territories of the United States in the early 19th century.

1. **Goal-Oriented Exploration:**
   - **Lewis and Clark:** Their mission was to find a practical route to the Pacific Ocean and gather comprehensive information about the newly acquired territory.
   - **Exploratory Testing:** The objective is to discover defects and understand the software's behavior by investigating its functionalities.
2. **Team Composition and Resources:**
   - **Lewis and Clark:** They assembled a diverse team, including military personnel, hunters, trappers, craftsmen, naturalists, and a guide familiar with the terrain.
   - **Exploratory Testing:** Testers use a combination of skills, knowledge, tools, and techniques to explore and test the software effectively.
3. **Learning Through Exploration:**
   - **Lewis and Clark:** They learned about the land, people, flora, and fauna as they traveled, adjusting their plans based on new information.
   - **Exploratory Testing:** Testers continuously learn about the system under test, using their findings to inform and refine their testing strategy.
4. **Detailed Documentation:**
   - **Lewis and Clark:** They meticulously documented their observations, findings, and interactions with native tribes.

- **Exploratory Testing:** Testers document their testing process, discoveries, and any defects found to provide valuable information to stakeholders.

5. **Adaptive Approach:**
   - **Lewis and Clark:** They adapted their route and methods based on the challenges and opportunities they encountered during their expedition.
   - **Exploratory Testing:** Testers adapt their testing techniques and focus areas based on the results of their ongoing exploration and the specific context of the software.

10.1.4 Exploring a Familiar Example - The Commission Problem

The commission problem offers a practical scenario for understanding the principles of exploratory testing. This approach to testing goes beyond verifying the presence of faults and seeks to uncover the nature and root causes of those faults.

Scenario Overview :

In this scenario, a salesperson sells interchangeable rifle parts: locks, stocks, and barrels. The pricing for these parts is as follows:

Locks cost $45

Stocks cost $30

Barrels cost $25

A complete rifle costs $100. The commission structure for the salesperson is:

10% on the first $1000 in sales

15% on sales between $1001 and $1800

20% on sales over $1800

Initially, the salesperson's commissions are accurate when sales are under $1000. However, discrepancies arise as sales increase, prompting an exploratory investigation to identify the underlying issues.

Exploratory Process

1. **First Exploration :**

   The salesperson begins by examining simple cases to verify the correctness of the basic commission calculations. The following equations are used:

   Sales Calculation:

   Sales = 45 * locks + 30 * stocks + 25 * barrels

   Commission Calculations:

   Commission = 0.10 * sales for $0 ≤ sales ≤ $1000

   Commission = $100 + 0.15 * (sales - $1000) for $1000 < sales ≤ $1800

   Commission = $220 + 0.20 * (sales - $1800) for sales > $1800

Test Cases:

1. **Simple Sales:** Sales of one of each item (locks, stocks, barrels) result in expected and computed commissions matching perfectly.
2. **Increasing Sales:** As sales increase, expected commissions are correct up to $1000. Discrepancies start to appear above $1000, indicating potential issues with the commission calculation formula.

Results;

Case No.	Locks	Stocks	Barrels	Sales	Expected Commission	Computed Commission	Pass	Difference
1	1	1	1	$100.00	$10.00	$10.00	Pass	$0.00
2	8	8	8	$800.00	$80.00	$80.00	Pass	$0.00
3	10	10	10	$1000.00	$100.00	$100.00	Pass	$0.00
4	11	11	11	$1100.00	$115.00	$100.00	Fail	$15.00
5	17	17	17	$1700.00	$205.00	$190.00	Fail	$15.00
6	18	18	18	$1800.00	$220.00	$205.00	Fail	$15.00
7	19	19	19	$1900.00	$240.00	$260.00	Fail	-$20.00

2. **Second Exploration:**

   To further investigate, the salesperson tests scenarios with only one type of item sold to isolate potential errors in the sales calculation.

   Test Cases: Selling only locks, stocks, or barrels individually.

   Results:

Case No.	Locks	Stocks	Barrels	Sales	Expected Commission	Computed Commission	Pass	Difference
1	10	0	0	$450.00	$45.00	$45.00	Pass	$0.00
2	0	10	0	$300.00	$30.00	$30.00	Pass	$0.00
3	0	0	10	$250.00	$25.00	$25.00	Pass	$0.00

3. **Third Exploration:**

   Next, the salesperson devises sales scenarios near the $1000 commission threshold to pinpoint where the calculation error occurs.

   Test Cases: Testing sales amounts just below, at, and above $1000.

   Results:

Case No.	Locks	Stocks	Barrels	Sales	Expected Commission	Computed Commission	Pass	Difference
1	22	0	0	$990.00	$99.00	$99.00	Pass	$0.00
2	21	0	2	$995.00	$99.50	$99.50	Pass	$0.00
3	21	1	1	$1000.00	$100.00	$100.00	Pass	$0.00
4	21	2	0	$1005.00	$100.75	$85.75	Fail	$15.00

Root Cause Analysis

Upon analyzing the discrepancies, the salesperson discovers that the fault lies in the amount subtracted from sales in the commission calculation for sales over $1000. The formula erroneously subtracts $1100 instead of $1000:

Expected Formula: Commission = $100 + 0.15 * (sales - $1000)

Faulty Formula: Commission = $100 + 0.15 * (sales - $1100)

This example illustrates the exploratory testing process, where the tester (salesperson) uses domain knowledge to identify potential faults, conducts targeted tests to isolate the issue, and applies analytical reasoning to uncover the root cause. This approach not only verifies the presence of faults but also helps understand their nature, making it an effective testing strategy in various scenarios.

10.1.5 Exploratory and Context-Driven Testing Observations

James Bach argues that anyone who tests software does a certain amount of exploratory testing. It is more accurate to say that debugging one's own code is exploratory testing. Because the descriptions of these forms of testing are so general and dynamic. Given the broad and somewhat general nature of exploratory testing, drawing precise conclusions can be challenging. However, based on experience, the following are the key observations:

1. **Suitability in Agile Environments:**
   - **Observation :** Exploratory testing is appropriate, but difficult, in an agile programming environment.
   - **Conclusion :** While exploratory testing is valuable in agile contexts, it relies on having a reasonably stable application. The iterative nature of agile can make it difficult to conduct follow-up tests based on previous results if the application is frequently changing.
2. **Dependence on Domain Experience:**
   - **Observation :** The effectiveness of exploratory testing is heavily reliant on the tester's domain knowledge and expertise.
   - **Conclusion:** Just as a computer science professor might struggle to conduct an oral examination of chemistry, a tester without sufficient domain knowledge may miss critical issues. Domain experience enables testers to ask relevant questions and design meaningful tests.
3. **Tester Motivation and Creativity:**
   - **Observation :** Successful exploratory testing requires a motivated, curious, and creative tester.
   - **Conclusion :** A disinterested tester will struggle to create effective follow-up tests. The tester's ability to think creatively and explore different scenarios is crucial for uncovering hidden defects.
4. **Challenge of Predictive Measurement:**
   - **Observation :** Exploratory testing resists predictive measurement, similar to other creative activities.

   - **Conclusion :** Estimating the amount of testing needed or the number of remaining faults is theoretically impossible. Effective exploratory testers rely on their judgment to decide when to stop testing, usually when no new faults are being discovered. This unpredictability parallels the uncertainty Meriwether Lewis would have faced in predicting the completion date of his expedition.
5. **Management of Exploratory Testing:**
   - **Observation:** Managing exploratory testing involves ensuring a clear testing charter and requiring detailed documentation of tests and results.
   - **Conclusion:** Clear objectives and thorough documentation are essential to keep track of what has been tested and the findings. This structured approach helps manage the inherently unstructured nature of exploratory testing.
6. **Effectiveness Relative to System Size and Complexity:**
   - **Observation:** The effectiveness of exploratory testing diminishes as the size and complexity of the system increase.
   - **Conclusion:** While exploratory testers can explore large, complex systems, keeping track of all follow-up tests becomes challenging. Exploratory testing is most effective for smaller, more comprehensible systems, where an individual tester can manage and remember the various aspects being tested.

10.1.6 Advantages and Disadvantages of Exploratory Testing

Exploratory testing is a dynamic and flexible approach to software testing that emphasizes real-time learning, test design, and execution. While it offers several benefits, it also has its drawbacks. Understanding these can help testers and organizations effectively integrate exploratory testing into their overall testing strategy.

Advantages of Exploratory Testing

1. **Adaptability and Flexibility:**
   Advantage: Testers can quickly adapt their testing strategy based on immediate findings and feedback from the system.
   Example: If a tester discovers an unexpected behavior in the software, they can immediately focus on that area to investigate further without waiting for predefined test cases.
2. **Rapid Feedback:**
   Advantage: Exploratory testing provides quick feedback on the system's behavior and stability, which is particularly valuable in fast-paced development environments like Agile.
   Example: During a sprint review, a tester can rapidly explore new features to identify any critical issues before the feature is deemed complete.
3. **In-depth Investigation:**
   Advantage: Testers can delve deeply into specific areas of the application, exploring edge cases and complex scenarios that might be overlooked by scripted testing.
   Example: A tester investigating a payment processing system can explore various combinations of payment methods, currencies, and user inputs to uncover subtle defects.

4. **Creative Problem Solving:**

   Advantage: The approach encourages testers to think creatively and use their intuition and experience to uncover hidden defects.

   Example: A tester might try entering unexpected input values, like special characters or extremely long strings, to see how the system handles them.

5. **Real-Time Learning:**

   Advantage: Testers learn about the system as they test, which helps them build a more comprehensive understanding of its behavior and potential issues.

   Example: As a tester explores different functionalities, they gain insights that can help them identify related defects and areas that need more thorough examination.

6. **Reduced Documentation Overhead:**

   Advantage: Less upfront planning and documentation are required compared to traditional testing, allowing testers to focus more on actual testing.

   Example: Instead of writing extensive test cases in advance, testers document their findings and insights as they test, making the process more efficient.

Disadvantages of Exploratory Testing

1. **Unpredictable Coverage:**

   Disadvantage: It is difficult to ensure comprehensive coverage of all functionalities, as exploratory testing relies on the tester's intuition and experience.

   Example: Testers might miss some parts of the application if they focus too much on specific areas that seem more problematic.

2. **Dependence on Tester Skill:**

   Disadvantage: The effectiveness of exploratory testing heavily depends on the tester's expertise, creativity, and domain knowledge.

   Example: An inexperienced tester might not be able to identify subtle defects or might miss important test scenarios that a more experienced tester would catch.

3. **Lack of Reproducibility:**

   Disadvantage: Because exploratory testing is less structured, it can be challenging to reproduce the exact steps that led to the discovery of a defect.

   Example: If a tester finds a bug but does not document the exact steps taken, it might be difficult for developers to reproduce and fix the issue.

4. **Difficult to Measure Progress:**

   Disadvantage: It is hard to measure progress and coverage since there are no predefined test cases or metrics to track.

   Example: Managers might find it challenging to assess how much testing has been done or how much more is needed without clear documentation and metrics.

5. **Limited to Individual Testing:**

   Disadvantage: Exploratory testing is less effective when multiple testers need to work together, as it relies on individual intuition and exploration.

   Example: Coordinating and sharing findings among a team of testers can be difficult without a structured approach, leading to potential gaps in testing.

6. **Potential for Incomplete Documentation:**

   Disadvantage: The focus on real-time exploration can lead to incomplete or inconsistent documentation of the testing process and findings.

   Example: If testers do not consistently document their activities and findings, it can be challenging to track what has been tested and what still needs attention.

10.2 Model Based Testing

Model-Based Testing (MBT) is an innovative approach to software testing that utilizes models to design, execute, and analyze test cases. In MBT, a model represents the intended behavior of the system under test, capturing its functionalities, interactions, and constraints. These models can take various forms such as finite state machines, statecharts, or UML diagrams, depending on the complexity and requirements of the system.

The core idea behind MBT is to generate test cases automatically from these models to ensure comprehensive test coverage while reducing manual effort and human error. By systematically deriving test cases from the model, testers can uncover potential defects early in the development lifecycle and validate the system against its specified requirements.

MBT offers several advantages including improved test coverage, faster test case generation, and enhanced traceability between requirements and test cases. Additionally, by maintaining a clear separation between the model and test implementation, MBT promotes reusability and scalability in testing efforts.

? What is Model Based Testing (MBT) ?

Model-Based Testing (MBT) is a methodology in software testing where test cases are derived from models that represent the desired behavior of the system under test. These models can be viewed as abstract representations of the system's functionalities, capturing the various states, transitions, inputs, and outputs. The primary goal of MBT is to ensure that the system behaves as expected by systematically exploring these models to generate comprehensive test cases.

10.2.1 Key Components of Model Based Testing

1. **Models:**
   - **Definition:** Models are simplified representations of the system that capture essential behavior and logic. They can take various forms, such as state machines, flowcharts, Petri nets, or UML diagrams.
   - **Purpose:** Models serve as the basis for generating test cases and providing a blueprint of the system's expected behavior.
2. **Test Case Generation:**
   - **Process:** Test cases are automatically or manually derived from the models. This involves identifying all possible paths or sequences of actions within the model that need to be tested.

- **Benefits:** This systematic approach helps ensure coverage of different scenarios, edge cases, and potential points of failure.

3. **Execution and Validation:**
   - **Execution:** The generated test cases are executed on the actual system to verify its behavior against the model.
   - **Validation:** Results are compared with the expected outcomes defined in the model. Any deviations are analyzed to identify defects or inaccuracies in the model or the system.

10.2.2 Features or Characteristics of Model-Based Testing (MBT)

Model-Based Testing (MBT) boasts several distinctive features that differentiate it from traditional testing approaches. These characteristics make MBT a powerful and efficient methodology for ensuring software quality.

1. **Model-Driven Approach:**

   In MBT, models serve as the foundation for the entire testing process. These models represent the expected behavior of the system under various conditions.

   Models provide an abstract view of the system, focusing on key behaviors and interactions rather than implementation details.

2. **Automated Test Case Generation:**

   Test cases are systematically derived from the models to ensure comprehensive coverage of the system's behavior. Automation Tools can automate the generation of test cases from models by significantly reducing the time and effort required to create and maintain test suites.

3. **Increased Test Coverage:**

   By exploring all possible paths within a model, MBT ensures thorough testing, including edge cases and unexpected scenarios that might be missed in manual testing. Path Coverage ensures that different sequences of actions and events are tested.

4. **Early Detection of Defects:**

   By creating models early in the development process, MBT can identify inconsistencies, ambiguities, and defects before the system is fully implemented.

5. **Consistency and Traceability:**

   Models serve as a single source of truth for the system's expected behavior to ensure that all test cases are aligned with this reference. MBT provides clear traceability from requirements to test cases, making it easier to understand the impact of changes in the requirements on the tests.

6. **Reusable Models:**

   Models can be reused across different projects or different versions of the same project, providing a consistent testing approach and saving time in the long run.

7. **Cost-Effectiveness:**

   By automating test case generation and reducing manual effort, MBT can lower the overall cost of testing. Models and automated test cases require less maintenance compared to traditional test scripts, which can become outdated as the system evolves.

8. **Support for Complex Systems:**

   MBT is particularly effective for complex systems with numerous states and interactions, where manual testing would be impractical. MBT is capable of modeling and testing concurrent processes and time-dependent behaviors, which are challenging to test manually.

9. **Improved Communication:**

   Models provide clear and concise documentation of the system's behavior, which can be easily understood by all stakeholders, including developers, testers, and business analysts.

10. **Adaptability to Changes:**

    Models can be easily updated to reflect changes in requirements or design to ensure that test cases remain relevant and accurate.

10.2.3 Testing Based on Models

Testing Based on Models involves creating a model of the system's behavior to derive insights and generate test cases. The process typically includes the following steps:

1. **Model the system:** The first step in MBT is to create a model that accurately reflects the system's behavior. This model serves as an abstract representation of the system and can take various forms, such as finite state machines, Petri nets, or StateCharts. The chosen model type depends on the nature of the system and the specific behaviors that need to be tested.

2. **Identify threads of system behavior in the model:** Once the model is created, the next step is to identify threads of system behavior within the model. A thread of behavior is a sequence of actions or events that the system can undergo. This involves tracing possible paths through the model that represent valid sequences of user interactions or system operations.

3. **Transform these threads into test cases:** The identified threads of behavior are then transformed into concrete test cases. Each test case corresponds to a specific path through the model and includes the necessary inputs, preconditions, and expected outputs.

4. **Execute the test cases:** The generated test cases are executed on the actual system. This involves running the system through the specified sequences of actions and observing the outcomes. The results are recorded to verify whether the system behaves as expected.

5. **Revise the model:** Based on the results of test case execution, the model may need to be revised. If discrepancies are found between the model's predictions and the actual system behavior, the model should be updated to reflect the observed behavior. This iterative process helps refine the model and improve its accuracy.

Examples: Testing an ATM System

1. **Model the System:** Create a StateChart model of the ATM system with states and transactions.

   States:

   Idle

   Card Inserted

   PIN Entered

   Transaction Type Selected

   Processing Transaction

   Transaction Completed

   Transitions:

   Insert Card (from Idle to Card Inserted)

   Enter PIN (from Card Inserted to PIN Entered)

   Select Transaction (from PIN Entered to Transaction Type Selected)

   Process Transaction (from Transaction Type Selected to Processing Transaction)

   Complete Transaction (from Processing Transaction to Transaction Completed)

   Return to Idle (from Transaction Completed to Idle)

2. **Identify Threads of System Behavior:**

   Thread 1: User inserts card -> Enters correct PIN -> Selects withdrawal -> System dispenses cash -> Returns card.

   Thread 2: User inserts card -> Enters incorrect PIN -> System prompts for re-entry -> User re-enters correct PIN -> Selects balance inquiry -> System displays balance -> Returns card.

3. **Transform These Threads into Test Cases:**

   Test Case 1:

   Inputs: Card insertion, correct PIN entry, withdrawal selection.

   Preconditions: Card is valid, account has sufficient funds.

   Expected Outputs: Cash dispensed, card returned, transaction receipt printed.

   Test Case 2:

   Inputs: Card insertion, incorrect PIN entry, correct PIN re-entry, balance inquiry selection.

   Preconditions: Card is valid.

   Expected Outputs: System prompts for re-entry of PIN after incorrect entry, displays balance after correct PIN and selection.

4. **Execute the Test Cases:**

   Execution of Test Case 1: Insert card into ATM, enter correct PIN, select withdrawal. Observe and record whether cash is dispensed and card is returned.

   Execution of Test Case 2: Insert card, enter incorrect PIN, re-enter correct PIN, select balance inquiry. Observe system prompts and balance display.

5. **Revise the Model:**

   If during execution of Test Case 1, the ATM does not return the card, update the model to reflect this behavior and generate additional test cases to explore variations of this scenario.

   If during Test Case 2, the system fails to prompt for re-entry of PIN after an incorrect entry, update the model to include this potential issue and test further.

10.2.4 Appropriate Models

It crucial to select a model that accurately reflects the system's behavior without adding unnecessary complexity. Appropriate models in Model-Based Testing (MBT) should strike a balance between expressiveness and practicality. The model's choice highly influences the testing approach and outcomes.

Avvinare : The term **avvinare**, an Italian word, is used metaphorically to emphasize the importance of preparing properly for an effective MBT process. In the same way that wine bottles need to be thoroughly rinsed to ensure the quality of the wine, selecting the correct model is essential to ensure the accuracy and effectiveness of MBT. An inappropriate model can lead to incomplete testing and missed defects, just as improperly prepared bottles can spoil the wine.

Peterson's Lattice

Peterson's Lattice is a conceptual framework developed by James Peterson in 1981 to categorize different computational models based on their expressive power. The lattice structure visualizes the relationships between various models, indicating how one model's expressiveness compares to another. In essence, Peterson's Lattice helps in understanding which models can be used to represent specific behaviors in a system and assists in selecting the most appropriate model for a given application.

Key Concepts of Peterson's Lattice

1. **Expressive Power:** The primary focus of Peterson's Lattice is to compare the expressive power of different models. A model with greater expressive power can represent more complex behaviors and scenarios than a less expressive one. For example, if Model A can express everything that Model B can, plus more, then Model A is considered more expressive.
2. **Hierarchy of Models:** The lattice arranges models in a hierarchical manner. More expressive models appear higher in the lattice, while less expressive models are lower. This hierarchy helps in choosing a model that is sufficiently expressive to capture the required behavior without adding unnecessary complexity.
3. **Use Cases:** Peterson's Lattice is particularly useful in Model-Based Testing (MBT) for determining which model to use based on the specific requirements of the system under test. It ensures that the chosen model is capable of representing all necessary aspects of the system's behavior.

Example Models:

1. **Finite State Machines (FSMs):**

   Characteristics: Suitable for systems with a limited number of discrete states and straightforward transitions.

   Expressive Power: Can model simple sequential behaviors but struggles with concurrency and complex interactions.

2. **Petri Nets:**

   Characteristics: Ideal for modeling systems with concurrent processes and synchronization.

Expressive Power: More expressive than FSMs, capable of representing concurrent activities and resource sharing.

3. **StateCharts:**

   Characteristics: Useful for complex systems with hierarchical and concurrent states.

   Expressive Power: More expressive than both FSMs and Petri Nets. It allows detailed modeling of complex behaviors including concurrency and nested states.

4. **Vector Addition Systems, Semaphore Systems:**

   Characteristics: These models capture specific computational behaviors such as synchronization and resource management.

   Expressive Power: Positioned in the lattice based on their ability to represent particular types of system behavior.

Visualization of Peterson's Lattice

Peterson's Lattice can be visualized as a directed graph where nodes represent different models and directed edges indicate the "more expressive than" relationship as shown in below diagram.

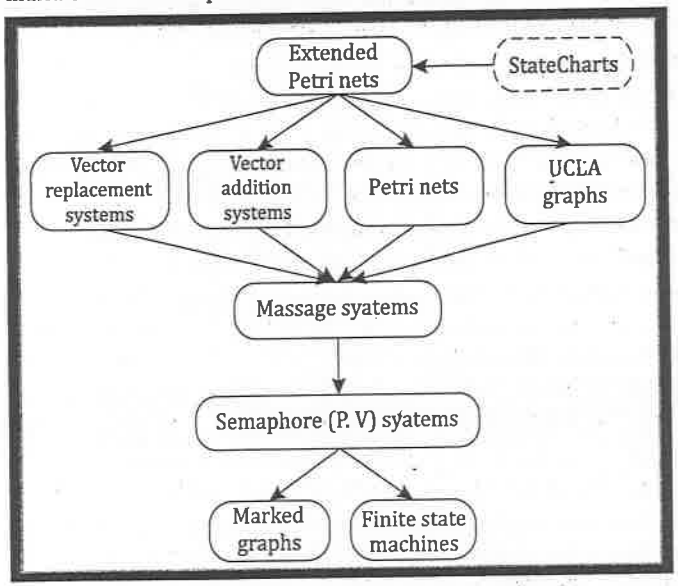


Fig 10.1 : Visualization of Peterson's Lattice

In this lattice:

- Finite State Machines and Marked Graphs are at the lower end, representing basic state transitions.
- Semaphore Systems are more expressive, capturing additional behaviors such as synchronization.

- Petri Nets offer higher expressiveness, suitable for concurrent systems.
- StateCharts are at a similar or higher level of expressiveness than Extended Petri Nets, capturing complex hierarchical and concurrent behaviors.

Expressive Capabilities of Mainline Models

Peterson categorizes models based on their ability to express various behavioral issues. These issues include aspects such as data flow, control flow, mutual exclusion, synchronization, and more. Each type of model has its strengths and weaknesses in addressing these issues.

Example: Petri Nets is highly expressive for representing concurrency and synchronization in systems, making them suitable for modeling network protocols. Petri nets can illustrate how multiple processes interact and coordinate with each other, which is essential for understanding and testing complex, concurrent systems.

Modeling Issues

Modeling issues involve understanding both the structural and behavioral aspects of a system. Structural models describe what a system is, focusing on its components and their relationships. Behavioral models describe what a system does, focusing on the interactions and state changes over time.

Examples:

1. **Structural Model:** UML class diagram showing the relationships between classes, attributes, and methods. It provides a blueprint of the system's static aspects.

   Use Case: Useful in understanding the system's architecture and ensuring that all necessary components are included.

2. **Behavioral Model:** UML sequence diagram showing the interaction between objects over time. It captures the dynamic behavior and flow of messages.

   Use Case: Useful in capturing the sequence of operations, message exchanges, and interactions among system components during execution.

Making Appropriate Choices

Choosing the right model involves understanding the system's nature and aligning it with the model's capabilities. The model should be simple enough to be manageable but sufficiently expressive to capture all necessary aspects of the system's behavior.

Examples:

1. **Finite State Machine (FSM):** For a system primarily involving user interactions with clear, distinct states, such as an elevator control system, an FSM might be sufficient. It can clearly represent each state (e.g., floors) and transitions (e.g., moving up/down).
2. **StateChart:** For a system with complex, event-driven behavior, such as an automated teller machine (ATM), a StateChart might be more appropriate. StateCharts can represent nested states and concurrent regions, providing a detailed view of the system's behavior in response to different events and conditions.

Example **Choosing the Right Model for an E-Commerce Platform**

Scenario: Testing the Checkout Process

1. **Finite State Machine (FSM):**
   - **Simplicity:** The checkout process can be modeled with states such as "Cart," "Billing Information," "Payment," and "Confirmation."
   - **Use Case:** Suitable if the process is linear and has a few distinct steps with clear transitions.
2. **Petri Net:**
   - **Concurrency:** If the checkout process involves concurrent activities, such as handling multiple payment methods or real-time inventory checks, a Petri Net would be more appropriate.
   - **Use Case:** Represents parallel actions and synchronization points effectively.
3. **StateChart:**
   - **Complexity:** If the checkout process involves complex user interactions, such as modifying the cart, applying discounts, or handling asynchronous events like payment confirmations, a StateChart would capture these behaviors more effectively.
   - **Use Case:** Suitable for detailed modeling of interactions and hierarchical states, providing a comprehensive view of the system's behavior.

10.2.5 Commercial Tool Support for Model-Based Testing

Model-Based Testing (MBT) leverages various tools to support the testing process. According to Alan Hartman (2003), these tools can be categorized into three main groups: Modeling Tools, Model-Based Test Input Generators, and Model-Based Test Generators. Each group plays a specific role in the MBT workflow, contributing to the overall efficiency and effectiveness of the testing process.

1. **Modeling Tools :** Modeling tools are designed to create and manage models that represent the system's behavior. These tools are crucial for the initial phase of MBT, where the system's functionalities and interactions are modeled.

   Examples:
   - **IBM's Rational Rose:** A comprehensive software modeling tool that uses UML (Unified Modeling Language) to create visual models of software architecture and behavior.
   - **Telelogic's Rhapsody:** A modeling tool that supports UML and SysML (Systems Modeling Language) for designing and analyzing complex systems.
   - **Stalemate:** Another tool that provides modeling capabilities, although less common in mainstream use.

   These tools primarily provide inputs to true model-based test generators but do not generate test cases by themselves.
2. **Model-Based Test Input Generators :** Model-Based Test Input Generators are a step up from basic modeling tools. They can automatically generate the input portion of test cases from the models. However, they do not generate the expected output portion of the test cases.

Functionality:
- Generate input data and conditions based on the model's specifications.
- Automate the creation of various test scenarios by exploring different paths through the model.

3. **Model-Based Test Generators :** Model-Based Test Generators go beyond generating inputs; they also generate the expected outputs for the test cases. These tools require an oracle to identify and validate the expected outputs, making the testing process more automated and comprehensive. Several proprietary and university-developed test generation systems claim to offer full model-based test generation capabilities. However, widespread commercial viability is still limited.

   In the context of software testing, the term "oracle" refers to a mechanism or entity that determines the expected outcome of a test. A test oracle is a source of information that determines the correct behavior of a system for a given set of inputs. It can be a formal specification, a mathematical model, an existing system, or even a human expert. The primary role of a test oracle is to validate the outputs generated by the system under test (SUT) against the expected outputs.

   Challenges:

   In Model-Based Testing (MBT), the oracle is crucial for automating the validation process. Full model-based test generators require an oracle to automatically generate both the inputs and the expected outputs for the test cases. Without an oracle, the validation step remains manual, which reduces the efficiency and effectiveness of MBT.

10.2.6 Advantages and Disadvantages of Model Based Testing

Model-Based Testing (MBT) offers a structured and automated approach to testing by leveraging models that represent the desired behavior of the system under test. While it brings numerous benefits, it also comes with certain challenges.

Advantages of Model-Based Testing

1. **Systematic Approach:** MBT ensures comprehensive coverage of the system's behavior by systematically exploring all possible paths and scenarios defined in the model.
2. **Edge Case Identification:** Helps identify edge cases and unusual scenarios that might be missed in traditional testing methods.
3. **Automated Test Case Generation:** Models can be used to automatically generate test cases, reducing the time and effort required for test design.
4. **Consistent Test Execution:** Automation ensures that tests are executed consistently across different test runs, reducing human error.
5. **Early Modeling:** Defects can be identified early in the development lifecycle by creating and validating models before the system is fully implemented.
6. **Shift-Left Testing:** Encourages early testing, which helps catch defects sooner and reduces the cost and effort of fixing them later.

7. **Clear Documentation:** Models provide clear and visual documentation of system behavior, improving communication among stakeholders, including developers, testers, and business analysts.
8. **Shared Understanding:** Models serve as a single source of truth, ensuring that all team members have a shared understanding of the system's requirements and expected behavior.
9. **Reusable Models:** Models can be reused across different projects or different versions of the same project, providing consistency and saving time in the long run.
10. **Scalability:** MBT supports scalability, allowing models to be expanded or modified as the system evolves.
11. **Reduced Maintenance:** Automated test case generation and execution reduce the maintenance effort compared to manual test scripts, which can become outdated quickly.
12. **Efficiency Gains:** By automating many aspects of the testing process, MBT can lower the overall cost of testing.

Disadvantages of Model-Based Testing

1. **Complexity:** The initial setup of MBT can be complex, requiring a thorough understanding of modeling techniques and tools.
2. **Learning Curve:** Teams may face a steep learning curve in adopting MBT, especially if they are new to model-based approaches.
3. **Tool Availability:** The availability of comprehensive MBT tools that can generate both test inputs and expected outputs is still limited.
4. **Integration Challenges:** Integrating MBT tools with existing development and testing environments can be challenging.
5. **Model Accuracy:** The effectiveness of MBT heavily relies on the accuracy of the models. Inaccurate or incomplete models can lead to ineffective testing.
6. **Maintenance Effort:** Keeping models up to date with evolving system requirements and designs can be labor-intensive.
7. **Cost of Tools:** MBT tools can be expensive, and organizations may need to invest in purchasing and maintaining these tools.
8. **Training Costs:** Additional training may be required for team members to effectively use MBT tools and techniques.
9. **Single Source of Truth:** While models provide a clear documentation of the system, over-reliance on them can be problematic if they are not properly maintained or validated.
10. **Modeling Skill Requirement:** Effective MBT requires skilled modelers who can create accurate and comprehensive models of the system.
11. **Expected Output Generation:** Full model-based test generators require an oracle to identify expected outputs, which can be a significant challenge. Without an oracle, automated validation of test results is difficult.

10.3 Use Case Based Testing

Use Case-Based Testing is a testing approach that focuses on deriving test cases from use cases, which are descriptions of how users interact with a system to achieve specific goals or tasks. By aligning test cases with the intended user interactions outlined in use cases, this method ensures that the software meets the functional requirements and user expectations.

? What is Use Case Based Testing ?

Use Case Based Testing is a software testing approach that derives test cases from use cases. Use cases are detailed descriptions of how users interact with a system to achieve specific goals. This testing methodology ensures that the system meets the user's functional requirements by validating that each use case is handled correctly.

10.3.1 Key Concepts of Use Case Based Testing

1. **Use Case Definition:**
   - **Use Case:** A use case is a description of a system's behavior as it responds to a request from one of the stakeholders (usually an actor). It defines the interactions between the user (actor) and the system to achieve a goal.
   - **Actors:** Entities that interact with the system (e.g., users, other systems).
   - **Scenarios:** Different paths through the use case, including main success scenarios and alternate scenarios for error conditions or exceptions.
2. **Components of a Use Case:**
   - **Title:** A descriptive name for the use case.
   - **Primary Actor:** The main user initiating the interaction.
   - **Preconditions:** Conditions that must be true before the use case can start.
   - **Postconditions:** Conditions that must be true after the use case completes.
   - **Main Success Scenario:** The standard sequence of steps to achieve the goal.
   - **Alternate Scenarios:** Variations of the main scenario to handle different conditions, errors, or exceptions.

10.3.2 Steps in Use Case Based Testing

The process of Use Case-Based Testing involves the following steps:

1. **Identify Use Cases:** Gather and document use cases that describe the functional requirements of the system. Each use case should outline the interactions between the actor and the system.
2. **Analyze Use Cases:** Break down each use case into individual steps and scenarios. Identify the main success scenario as well as alternate scenarios that represent different paths the interaction could take.
3. **Create Test Cases:** Derive test cases from each scenario described in the use case. Ensure that test cases cover the main success scenario and all alternate scenarios, including error handling and exceptions.
4. **Prioritize Test Cases:** Prioritize the test cases based on factors such as criticality to business, frequency of use, and potential impact of failure.
5. **Execute Test Cases:** Execute the derived test cases on the system under test to verify that each use case is handled correctly. Record the results of each test execution.
6. **Validate Results:** Compare the actual results with the expected outcomes defined in the use case to determine if the system behaves as intended. Investigate and document any discrepancies.

Example Use Case Based Testing

Let's consider an e-commerce platform and a use case for the checkout process.

Use Case: Checkout Process

Title: Checkout Process

Primary Actor: Customer

Preconditions: Customer is logged in and has items in the shopping cart.

Postconditions: Order is placed, and confirmation is sent to the customer.

1. **Main Success Scenario:**
   Customer views the shopping cart.
   Customer proceeds to checkout.
   Customer enters shipping information.
   Customer selects a payment method and enters payment details.
   Customer reviews the order and confirms the purchase.
   System processes the payment and places the order.
   System sends order confirmation to the customer.
2. **Alternate Scenarios:**
   - **Invalid Payment Information:** Customer enters invalid payment details.
   - **Out of Stock Item:** An item in the cart becomes out of stock before the order is placed.
   - **Address Validation Failure:** Customer enters an invalid shipping address.

Derived Test Cases:

1. **Main Success Scenario:**
   - **Test Case 1:** Verify that a customer can successfully complete the checkout process with valid information.
2. **Alternate Scenarios:**
   - **Test Case 2:** Verify that the system displays an error message when the customer enters invalid payment information.
   - **Test Case 3:** Verify that the system informs the customer and removes the out-of-stock item if an item becomes unavailable during checkout.
   - **Test Case 4:** Verify that the system prompts the customer to correct the address if the shipping address validation fails.

Execution and Validation:

- **Execution:** Execute each test case on the e-commerce platform to simulate the customer's actions and the system's responses.
- **Validation:** Check if the actual outcomes match the expected results. Ensure the system handles each scenario as described in the use case.

10.3.3 Advantages and Disadvantages of Use Case Based Testing

Use Case-Based Testing offers several benefits in terms of user-centric testing and comprehensive coverage, but it is also essential to consider its limitations and potential challenges to effectively leverage this approach in software testing.

Advantages of Use Case-Based Testing

1. **Alignment with User Requirements:** Use Case-Based Testing ensures that the testing process is closely aligned with user requirements and expectations, as test cases are derived directly from user interactions described in use cases.
2. **Comprehensive Test Coverage:** By deriving test cases from use cases that cover various user scenarios, this approach helps achieve comprehensive test coverage, ensuring that the software functions as intended under different conditions.
3. **Early Defect Detection:** Testing based on use cases allows for early detection of defects or deviations from expected behavior, as test cases are designed to validate the system's response to user interactions.
4. **Improved Communication:** Use cases serve as a common language between stakeholders, facilitating better communication and understanding of system functionality, which can lead to more effective testing.
5. **Reusability:** Test cases derived from use cases can be reused across different testing phases or iterations, promoting efficiency and reducing duplication of testing efforts.

Disadvantages of Use Case-Based Testing

1. **Limited Scope:** Use Case-Based Testing may focus primarily on functional aspects driven by user interactions, potentially overlooking non-functional requirements such as performance, security, or usability testing.
2. **Dependency on Use Case Quality:** The effectiveness of Use Case-Based Testing is highly dependent on the quality and completeness of the use cases. Inaccurate or incomplete use cases can lead to inadequate test coverage and potentially missed defects.
3. **Complexity in Large Systems:** In large and complex systems with numerous use cases, managing and deriving test cases from all use cases can become challenging and time-consuming.
4. **Maintenance Overhead:** As the system evolves and use cases are updated or modified, test cases derived from these use cases may require frequent updates and maintenance to remain relevant and effective.
5. **Risk of Overlooking Edge Cases:** Use Case-Based Testing may not always capture all edge cases or exceptional scenarios that could lead to critical defects, especially if these cases are not explicitly covered in the use cases.

10.4 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Exploratory Testing ?
2. Give an example for Exploratory Testing.
3. What is Lewis and Clark's Expedition ?
4. What is Model Based Testing (MBT) ?
5. What is Use Case Based Testing ?

Section - B

Each Question Carries Five Marks

1. Explain the Context-Driven School and Its Principles.
2. Explain the Characteristics of Exploratory Testing.
3. Compare Exploratory Testing and the Lewis and Clark Expedition.
4. What are the Advantages and Disadvantages of Exploratory Testing?
5. Explain Exploratory and Context-Driven Testing Observations.
6. What is Model Based Testing (MBT) ? Explain the Key Components of Model Based Testing.
7. Explain the Features or Characteristics of Model-Based Testing (MBT).
8. Explain the Process of Testing based on Models.
9. Explain Peterson's Lattice.
10. Write a note on Commercial Tool Support for Model-Based Testing.
11. Write the Advantages and Disadvantages of Model Based Testing.
12. What is Use Case Based Testing ? Explain the Steps in Use Case Based Testing.
13. Explain the Advantages and Disadvantages of Use Case Based Testing

Section - C

Each Question Carries Eight Marks

1. Explain and Elaborate on Exploratory Testing with Detailed Examples.
2. Explain How Exploratory Testing is done for the Commission Problem.
3. Explain and Elaborate on Model Based Testing with Detailed Examples.
4. Explain How to Chose Appropriate Models in Model Based Testing.
5. Explain and Elaborate on Use Case Based Testing with Detailed Examples.

❖❖❖❖❖

UNIT - IV

CHAPTER 11

TEST-DRIVEN DEVELOPMENT & SOFTWARE TESTING EXCELLENCE

Contents

11.1 Introduction to Test Driven Development (TDD)

Test-Driven Development (TDD) is a software development methodology that has gained significant popularity in the software engineering community, particularly within agile development practices. The core principle of TDD is to transform testing from a downstream activity to an integral part of the software development process, ensuring that testing drives the design and implementation of the code.

Before the advent of TDD, traditional software development often followed a linear or waterfall approach, where coding was done first, and testing was an afterthought. This often led to a disconnect between what the software was supposed to do and what it actually did, resulting in late discovery of defects, higher costs of bug fixes, and sometimes, subpar software quality.

Agile methodologies including Extreme Programming (XP) emphasize iterative development, customer collaboration, and adaptability to change. TDD originated from XP and popularized by Kent Beck in the early 2000s. TDD aligns well with agile principles by promoting a test-first approach and close integration of testing and development activities.

The adoption of TDD represents a shift in development practices towards a more proactive and iterative approach to software development. It emphasizes the importance of continuous testing, refactoring, and ensuring that code meets the expected behavior as defined by the tests.

? What is Test Driven Development (TDD) ?

Test-Driven Development (TDD) is a software development methodology that emphasizes writing tests before writing the actual code. This approach is characterized by short, iterative cycles where a test case is written first, followed by the minimum amount of code required to pass the test. TDD promotes the development of robust, high-quality software through continuous testing and refactoring.

11.2 Features or Characteristics of TDD

Test-Driven Development (TDD) is characterized by several key features and principles that distinguish it from traditional software development approaches. Some of the prominent features of TDD are:

1. **Test-First Approach:** In TDD, developers write automated tests before writing the actual code. This test-first approach ensures that the code is designed to meet specific requirements and that the tests act as a specification for the code's behavior.
2. **Incremental Development:** TDD promotes incremental development by breaking down the implementation into small, manageable steps. Developers write tests and code in short iterations, allowing for continuous integration and feedback.
3. **Red-Green-Refactor Cycle:** TDD follows a cycle known as Red-Green-Refactor. Developers start with a failing test (Red), write the minimum code to make the test pass (Green), and then refactor the code to improve its design without changing its behavior.
4. **Automated Testing:** TDD relies on automated testing frameworks to run tests quickly and frequently. Automated tests ensure that the code behaves as expected and can be easily executed to validate changes.

5. **Code Quality:** TDD emphasizes writing clean, maintainable code. By focusing on passing tests, developers are encouraged to write code that is modular, well-structured, and easy to understand.
6. **Regression Testing:** TDD helps in creating a comprehensive suite of tests that can be run automatically to detect regressions. Any changes made to the codebase are validated against existing tests to ensure that new features do not break existing functionality.
7. **Early Bug Detection:** TDD facilitates early bug detection by identifying issues at the unit level before they propagate to higher levels of the application. This leads to faster bug resolution and reduces the cost of fixing defects.
8. **Improved Design:** TDD often results in better software design as developers need to consider the interface and behavior of the code upfront. Refactoring during the process helps in continuously improving the design without compromising functionality.
9. **Collaboration:** TDD encourages collaboration between developers and testers as they work together to define test cases and ensure that the code meets the specified requirements. This collaborative approach fosters a shared understanding of the system's behavior.
10. **Increased Test Coverage :** TDD promotes thorough test coverage by requiring tests for every piece of functionality before it is implemented.
11. **Continuous Refactoring :** After making a test pass, the code is refactored to improve its structure and maintainability without changing its behavior. Continuous refactoring ensures that the codebase remains clean, efficient, and easy to maintain over time.
12. **Immediate Feedback :** TDD provides immediate feedback on the correctness of the code through automated tests. This rapid feedback loop helps developers quickly identify and fix issues, ensuring that the code remains functional and bug-free.

11.3 Test-Then-Code Cycles

In TDD, the development process is broken down into short, iterative cycles known as **"Test-Then-Code" Cycles.** Test-Then-Code Cycles are iterative development cycles where tests are written before the actual code implementation. It is also known as Test-Driven Development (TDD) cycles. The process typically involves the following steps:

Test-Driven Development (TDD) is a software development approach where tests are written before the actual code implementation. This process involves the following steps:

1. **Write a Test:**
   - Begin by defining a test case that outlines the expected behavior of a specific functionality.
   - The test is designed to fail initially as the corresponding functionality has not been implemented yet.
2. **Run the Test:**
   - Execute the test to confirm that it fails as expected.
   - This step validates the test's correctness and ensures that the feature is genuinely missing.

3. **Write Code:**
   - Implement the simplest code necessary to make the failing test pass.
   - The focus is on writing code that fulfills the requirements of the test case.
4. **Run All Tests:**
   - After implementing the new functionality, run all existing tests, including the newly added one.
   - This step verifies that the new code integrates smoothly with the existing system and does not break any existing functionality. If the test fails, it indicates that the code implementation is incorrect or incomplete.
5. **Refactor:**
   - Refactor the code to enhance its structure, readability, and performance without altering its external behavior.
   - Refactoring ensures that the code remains clean and maintainable.
   - It is essential to ensure that all tests continue to pass after refactoring.
6. **Repeat:**
   - Iterate through this cycle for each new piece of functionality or improvement.
   - By continuously following this process, developers build a robust codebase with comprehensive test coverage.

Example **Test-Then-Code Cycles**

Let's illustrate TDD with a simple example of creating a Java function to determine if a given year is a leap year.

Step 1: Write a Test

The first step in TDD is to write a test that defines the desired behavior of the function. In this case, we want to determine if a given year is a leap year. Create a test class (LeapYearTest) and write test methods using assertions to specify the expected outcomes.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class LeapYearTest {
    @Test
    public void testIsLeapYear() {
          assertTrue(LeapYear.isLeapYear(2024)); // 2024 is a leap year
          assertFalse(LeapYear.isLeapYear(2023)); // 2023 is not a leap year
          assertFalse(LeapYear.isLeapYear(1900)); // century year not divisible by 400
          assertTrue(LeapYear.isLeapYear(2000)); // 2century year divisible by 400
    }
}
```

In this test, we have four assertions to check different cases:

2024 is a leap year.
2023 is not a leap year.
1900 is not a leap year.
2000 is a leap year.

Step 2: Run the Test

At this point, running the test will fail because the isLeapYear method is not yet implemented. Run the test using Java IDE or build tool. Expect to see failures or errors indicating missing implementations.

Step 3: Write the Code

Now we implement the isLeapYear method with just enough code to make the test pass. Create the LeapYear class and implement the isLeapYear method with basic logic to check if a year is a leap year.

```
public class LeapYear {
    public static boolean isLeapYear(int year) {
        if (year % 4 == 0) {
            if (year % 100 == 0) {
                if (year % 400 == 0) {
                    return true;
                } else {
                    return false;
                }
            } else {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

Step 4: Run All Tests

Run the LeapYearTest again. This time, the test should pass because we have implemented the isLeapYear method. If it passes, it means the implementation is correct for the tested scenarios.

Step 5: Refactor

Refactoring is an essential part of TDD. We improve the code structure without changing its behavior. In this case, we can simplify the isLeapYear method.

```
public class LeapYear {
    public static boolean isLeapYear(int year) {
        return (year % 4 == 0) && (year % 100 != 0 || year % 400 == 0);
    }
}
```

Step 6: Repeat

Add more test cases and repeat the cycle for further validation and enhancements. Add new test cases for additional scenarios and repeat the TDD cycle.

This example demonstrates the TDD process using Java. We started with a test, implemented just enough code to pass the test, ran the tests, refactored the code, and repeated the cycle for further validation. By following these steps, developers can ensure that their code meets specified requirements and is well-tested, maintainable, and reliable.

11.4 Automated Test Execution

Automated Test Execution refers to the process of running tests automatically using specialized software tools known as **Testing Frameworks**. Testing Frameworks are software tools that provide a structured environment for writing, organizing, and executing automated tests. These frameworks typically include libraries and tools that help define test cases, manage test data, and report test results. Testing frameworks are available for most programming languages and development environments. These frameworks help developers and testers write, organize, and execute tests efficiently to ensure that software behaves as expected and meets the specified requirements.

11.4.1 Goals and Purpose of Automated Test Execution

Automated test execution is a crucial aspect of modern software development, especially within the context of Test-Driven Development (TDD) and other agile methodologies. The primary goals and benefits of automated test execution include:

1. **Efficiency:** Automated tests can be run quickly and repeatedly with minimal human intervention, saving time and reducing the manual effort required for testing.
2. **Consistency:** Automated tests ensure that tests are executed in a consistent manner, reducing the risk of human error and variability in test execution.
3. **Immediate Feedback:** Automated tests provide immediate feedback on the quality and correctness of the code to enable developers to detect and fix issues early in the development process.
4. **Regression Testing:** Automated tests make it easy to run regression tests to ensure that new code changes do not introduce new bugs or break existing functionality.
5. **Continuous Integration:** Automated tests allow for continuous testing as part of the development workflow.

11.4.2 Common Features of Testing Frameworks

1. **Test Case Management:** Frameworks provide structures and conventions for defining and organizing test cases.
2. **Assertions:** Frameworks include assertion methods to compare expected outcomes with actual results.
3. **Test Runners:** Frameworks provide test runners to execute test cases and collect results.
4. **Reporting:** Frameworks generate detailed reports on test results, including pass/fail status and error messages.
5. **Integration:** Frameworks often integrate with development environments and CI/CD tools to support automated and continuous testing.

11.4.3 Importance of Testing Frameworks in TDD

1. **Ease of Test Creation and Execution:** Testing frameworks provide a structured environment for writing and executing test cases. This makes it straightforward to define test inputs, expected outputs, and to run these tests automatically.

2. **Automated Test Execution:** Frameworks automate the process of running tests and checking results, reducing the manual effort involved and increasing the efficiency and reliability of the testing process.
3. **Consistent Test Results:** By using a standardized framework, tests are executed consistently across different environments to ensure that results are reproducible and reliable.
4. **Integration with Development Tools:** Most frameworks integrate seamlessly with development environments (IDEs) and continuous integration (CI) tools, allowing for automated test runs as part of the build process.
5. **Facilitation of Test-Driven Development:** Frameworks make it easy to follow the TDD cycle (write a test, write code to pass the test, run the test, refactor, repeat) by providing tools and features specifically designed for this purpose.

11.4.4 Examples of Testing Frameworks

Some examples of popular testing frameworks for various programming languages are listed below

Programming Language	Testing Framework
Java:	JUnit
Python:	unittest (PyUnit), pytest
C#/.NET:	NUnit, MSTest
JavaScript:	Jest, Mocha
Ruby:	Test::Unit, RSpec
PHP:	PHPUnit
C++:	CppUnit, Google Test
Objective-C:	OCUnit
Swift:	XCTest
Go:	testing package

11.4.5 Advantages and Disadvantages of Automated Test Execution

Automated test execution is a critical component of modern software development by offering numerous benefits but also presenting some challenges. Below are the key advantages and disadvantages of automated test execution.

Advantages of Automated Test Execution

1. **Efficiency and Speed**
   - **Rapid Execution:** Automated tests run much faster than manual tests, enabling quick feedback and faster development cycles.
   - **Repeatability:** Tests can be executed repeatedly with the same inputs and expected outcomes, ensuring consistent results.

2. **Cost-Effectiveness**
   - **Reduced Manual Effort:** Automating repetitive test cases reduces the time and labor costs associated with manual testing.
   - **Long-Term Savings:** Although the initial setup may be costly, automation can lead to significant cost savings over time by reducing the need for extensive manual testing.
3. **Accuracy and Reliability**
   - **Minimized Human Error:** Automated tests eliminate the possibility of human errors during test execution.
   - **Consistent Results:** Automated tests provide consistent and reliable results, ensuring that tests are executed in the same manner every time.
4. **Increased Test Coverage**
   - **Comprehensive Testing:** Automation allows for a broader scope of tests, including complex scenarios that might be impractical to test manually.
   - **Regression Testing:** Automated tests make it easy to run extensive regression tests to ensure that new changes do not break existing functionality.
5. **Continuous Integration and Deployment**
   - **CI/CD Integration:** Automated tests can be integrated into continuous integration and continuous deployment pipelines, allowing for continuous testing and faster release cycles.
   - **Immediate Feedback:** Developers receive immediate feedback on the impact of their changes, enabling faster identification and resolution of issues.
6. **Reusability**
   - **Reusable Test Scripts:** Automated test scripts can be reused across different projects or different versions of the same project, providing consistent test coverage and reducing the effort required to create new tests.
7. **Documentation and Reporting**
   - **Detailed Reports:** Automated testing tools often generate detailed reports and logs, providing insights into test execution and helping diagnose issues quickly.
   - **Traceability:** Automated tests provide a traceable record of what was tested, which can be valuable for compliance and audit purposes.

Disadvantages of Automated Test Execution

1. **Initial Setup Cost and Effort**
   - **High Initial Investment:** The initial setup of automated testing frameworks and the creation of test scripts can be time-consuming and expensive.
   - **Learning Curve:** Teams may need to invest in training to acquire the skills necessary to develop and maintain automated tests.
2. **Maintenance Overhead**
   - **Test Script Maintenance:** Automated test scripts require regular maintenance to keep up with changes in the application. If the application changes frequently, maintaining test scripts can become burdensome.
   - **False Positives/Negatives:** Automated tests can sometimes produce false positives or negatives due to issues with the test scripts themselves, requiring additional effort to debug and resolve.

3. **Limited Scope**
   - **Not All Tests Can Be Automated:** Some tests, particularly those requiring human judgment, such as usability and exploratory testing, cannot be effectively automated.
   - **Complex Setup for Certain Tests:** Automating tests for certain scenarios, such as those involving hardware interactions or complex user interfaces, can be challenging and may require sophisticated tools and setups.
4. **Upfront Time and Resources**
   - **Significant Initial Time Investment:** Setting up an automated testing environment and writing initial test scripts can take considerable time and resources, delaying immediate project timelines.
   - **Resource-Intensive:** Automated tests can require significant computational resources, especially for large test suites, potentially impacting other development activities.
5. **Over-Reliance on Automation**
   - **Neglecting Manual Testing:** Over-reliance on automated tests can lead to the neglect of manual testing, which is still essential for identifying issues that automated tests might miss.
   - **Complacency:** Teams might become complacent, assuming that automated tests will catch all issues, which can lead to gaps in test coverage.

11.5 Java and JUnit Example

In modern software development, writing robust, reliable, and maintainable code is crucial. Test-Driven Development (TDD) and automated testing are practices that help achieve these goals by integrating testing into the development process. Java programming language along with JUnit provides a strong foundation for implementing TDD and automated testing in Java based applications.

- Java is a object-oriented programming language that is used for building enterprise-scale applications due to its platform independence, robustness, and extensive libraries.
- JUnit is a widely-used testing framework for Java that supports test-driven development and automated testing. JUnit helps developers write and run repeatable tests, ensuring that the code behaves as expected.

Key Features of JUnit

1. **Annotations:** JUnit uses annotations to define test methods and setup/teardown methods. Common annotations include:
   - **@Test:** Marks a method as a test method.
   - **@Before:** Specifies a method that runs before each test.
   - **@After:** Specifies a method that runs after each test.
   - **@BeforeClass and @AfterClass:** Define methods that run once before and after all tests in a class, respectively.
2. **Assertions:** JUnit provides a set of assertion methods to verify expected outcomes. Common assertions include:
   - **assertEquals(expected, actual):** Checks that two values are equal.
   - **assertTrue(condition):** Checks that a condition is true.

- **assertFalse(condition):** Checks that a condition is false.
- **assertNotNull(object):** Checks that an object is not null.

3. **Test Runners:** JUnit test runners execute tests and provide feedback on test results. The default test runner runs all test methods in a class and reports the results.
4. **Integration:** JUnit integrates seamlessly with Java IDEs (like Eclipse and IntelliJ IDEA) and build tools (like Maven and Gradle). This integration facilitates continuous testing by allowing tests to be run automatically as part of the build process.

Example **Java and JUnit Example**

To illustrate how to use JUnit for TDD in Java, let's walk through an example. We'll create a simple calculator function to add, subtract, multiply, and divide two numbers.

Step 1: Write a Test

Purpose: Define the expected behavior of the calculator functions. Writing the test first ensures that you clearly understand the requirement.

Action: Create a test class (CalculatorTest) and write test methods using assertions to specify the expected outcomes. The CalculatorTest class contains test methods for addition, subtraction, multiplication, and division, including a test for division by zero which should throw an ArithmeticException.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void testAddition() {
        assertEquals(5, Calculator.add(2, 3));
        assertEquals(0, Calculator.add(-1, 1));
        assertEquals(-5, Calculator.add(-3, -2));
    }

    @Test
    public void testSubtraction() {
        assertEquals(1, Calculator.subtract(3, 2));
        assertEquals(-2, Calculator.subtract(-1, 1));
        assertEquals(-1, Calculator.subtract(-3, -2));
    }
    @Test
    public void testMultiplication() {
        assertEquals(6, Calculator.multiply(2, 3));
        assertEquals(-1, Calculator.multiply(-1, 1));
        assertEquals(6, Calculator.multiply(-3, -2));
    }
```



```
    @Test
    public void testDivision() {
        assertEquals(2, Calculator.divide(6, 3));
        assertEquals(-1, Calculator.divide(-1, 1));
        assertEquals(1, Calculator.divide(-3, -3));
    }

    @Test(expected = ArithmeticException.class)
    public void testDivisionByZero() {
        Calculator.divide(1, 0);
    }
}
```

Step 2: Run the Test

Initially, the test will fail because the Calculator methods are not yet implemented. Running the test in an IDE Eclipse will show that the tests fail, indicating that we need to implement the Calculator methods.

Step 3: Write the Code

Purpose: Implement the minimum amount of code required to pass the test.

Action: Create the Calculator class and implement the methods with basic logic for addition, subtraction, multiplication, and division.

```
public class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int subtract(int a, int b) {
        return a - b;
    }

    public static int multiply(int a, int b) {
        return a * b;
    }

    public static int divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Step 4: Run All Tests

Run the CalculatorTest again. This time, the tests should pass because the Calculator methods are correctly implemented. Running the test now will show that all assertions pass, confirming that the Calculator methods work as expected for the given test cases.

Step 5: Refactor

Purpose: Improve the code's structure, readability, and maintainability without changing its behavior.

Action: Since the implementation is straightforward, no immediate refactoring is necessary. However, if we identified any improvements, such as optimizing calculations or improving readability, we would do so while ensuring that all tests still pass.

Step 6: Repeat

Purpose: Iteratively enhance the functionality and ensure comprehensive test coverage.

Action: Add new test cases for additional scenarios and repeat the TDD cycle.

11.6 Remaining Questions (TDD Considerations and Challenges)

11.6.1 Is TDD Code Based or Specification Based?

Test-Driven Development (TDD) has characteristics that make it appear to be both specification-based and code-based. The nature of TDD involves writing test cases before the actual code, which can be seen as a form of low-level specification. However, since these test cases are closely tied to the implementation of code, it also has elements of being code-based testing.

1. **Specification-Based Aspects:**
   - **Low-Level Specifications:** Test cases in TDD serve as low-level specifications that define what the code should do. They specify the expected behavior of the code for given inputs.
   - **User Stories:** In the context of agile methodologies, each test case can be considered as a part of a user story, which is a higher-level specification accepted by customers.
2. **Code-Based Aspects:**
   - **Close Association with Code:** Test cases are written in conjunction with the code and are executed to validate the code's correctness. This creates a tight integration between testing and coding.
   - **Code Coverage:** Achieving high code coverage is a natural outcome of TDD, as tests are written to cover various paths through the code, ensuring thorough validation.
3. **Balancing Specification and Code:**
   - **Incremental Steps vs. Larger Chunks:** Some practitioners argue that instead of tiny, incremental steps, larger test cases followed by larger chunks of code might be preferable. This approach introduces an element of code design and reduces the frequency of refactoring, combining bottom-up development with top-down thinking.

11.6.2 Is Configuration Management Challenging in TDD?

TDD may seem challenging for configuration management due to the numerous versions a program may go through from inception to completion. However, the refactoring process inherent in TDD helps manage this complexity.

Configuration Management Actions in TDD

In Test-Driven Development (TDD), effective configuration management is essential to handle the multiple versions and iterations of the code. Here's how configuration management actions can be applied during the TDD process:

1. **Promoting to Configuration Item:**
   - **Refactoring Points:** When refactoring is successfully completed (meaning all tests pass after the refactoring), the code at this point is stable and reliable. This stable version of the code can be promoted to a configuration item. A configuration item is a managed component that is tracked and versioned in the configuration management system. This means it is now officially tracked and versioned as a reliable component in your project.
   - **Example:** After refactoring and ensuring all tests pass, the current version of the **Calculator** program is considered stable and is promoted to a configuration item. This means it is now a tracked version in the configuration management system.
2. **Demoting to Design Object:**
   - **Failed Tests on New Code:** If new code changes cause previously passing tests to fail, this indicates a problem. The configuration item (stable version of the code) should be re-evaluated. In this case, the configuration item can be demoted back to a design object. A design object is a less stable version that is subject to further development and changes. It means that further development and adjustments are required before it can be considered stable again.
   - **Example:** After adding new functionality to the **Calculator** program, if some of the existing tests fail, the previously stable version (configuration item) is demoted to a design object. This indicates it is still under development and needs further work to meet the stability criteria.

By following these practices, we can maintain control over the different versions of code and ensure that only stable, reliable versions are promoted and tracked as configuration items.

11.6.3 How Does Granularity Affect the TDD Process?

Granularity refers to the level of detail in user stories and the corresponding tests and code increments in TDD. The granularity of user stories can significantly impact the development process.

1. **Fine-Grained User Stories:**
   - **Detailed Steps:** Using fine-grained user stories involves breaking down functionality into very detailed steps. Each small increment is developed and tested individually.

- **Example:** In the sequence of user stories for a simple calculator program, each functionality (e.g., addition, subtraction, multiplication, division) is treated as a separate, detailed task.

2. **Larger Granularity (Story-Driven Development):**
   - **Larger User Stories:** Larger user stories include broader functionality, which is then broken down into finer tasks for development and testing.
   - **Advantages:** This approach preserves fault isolation while also allowing for some code design considerations, potentially reducing the frequency of refactoring.
   - **Example:** Instead of testing the operations of addition, subtraction, multiplication, and division separately, a larger user story might involve implementing and validating all basic calculator functions together. This larger user story is then broken down into tasks like implementing and testing the add, subtract, multiply, and divide methods collectively.
3. **Comparison:**
   - **Fine-Grained:** Offers detailed control and immediate feedback but may require frequent refactoring and can be more cumbersome for complex functionalities.
   - **Larger Granularity:** Balances detailed testing with higher-level design, potentially reducing the need for constant refactoring and allowing for more holistic code improvements.

11.7 Advantages, Disadvantages and Open Questions of TDD

Test-Driven Development (TDD) has both advantages and disadvantages along with some open questions that are still being explored in the software development community.

Advantages of TDD

1. **Working Code:** TDD ensures that something always works due to the tight test/code cycles, leading to a more reliable codebase.
2. **Fault Isolation:** TDD excels in fault isolation as any failing test indicates that the issue lies in the most recently added code.
3. **Supportive Test Frameworks:** TDD is supported by a variety of test frameworks such as JUnit for Java, making it easier to implement and maintain tests.
4. **Early Detection of Issues:** TDD encourages developers to write tests before writing the actual code, leading to early detection of potential issues and bugs.
5. **Improved Code Quality:** By continuously running tests and refactoring code, TDD can result in higher code quality and maintainability over time.

Disadvantages of TDD

1. **Dependency on Test Frameworks:** TDD heavily relies on test frameworks, and without them, it becomes challenging to practice TDD effectively.
2. **Limited Design Opportunities:** The bottom-up nature of TDD may limit opportunities for elegant design as it focuses on incremental improvements through refactorings.
3. **Inadequate for Deep Faults:** TDD may not effectively reveal deeper faults that require a comprehensive understanding of the code such as those uncovered by data flow testing.
4. **Learning Curve:** TDD may have a steep learning curve for developers who are new to the practice, as it requires a shift in mindset and workflow.
5. **Time-Consuming:** Initially, TDD may seem time-consuming as developers need to write tests alongside the code, which can slow down the development process compared to traditional methods.

Open Questions about TDD:

1. **Can TDD scale effectively to large applications?**

   There are concerns about the scalability of TDD to large applications and whether individuals can effectively manage the complexity of extensive codebases. The heavy load on developers increases with the size of the application, making it difficult to manage and maintain.
2. **Can TDD handle the complexity and reliability required for large, complex systems?**

   Complex systems often require sophisticated models and comprehensive testing beyond the scope of TDD's incremental approach.
3. **How well does TDD support long-term maintenance?**

   TDD advocates often argue against extensive documentation, relying on test cases as specifications and well-written code for self-documentation. The effectiveness of this approach over long-term projects remains uncertain.

These open questions highlight areas where further research and exploration are needed to fully understand the implications and limitations of Test-Driven Development in real-world software development scenarios.

11.8 Retrospective on MDD versus TDD

In software development, understanding different methodologies and their impacts on the development process is crucial. Model-Driven Development (MDD) and Test-Driven Development (TDD) are two such methodologies, each offering unique perspectives and approaches.

Both MDD and TDD offer valuable perspectives for software development. MDD provides a high-level, structured approach that ensures completeness and consistency, while TDD focuses on incremental development and strong fault isolation. Combining the strengths of both methodologies can lead to a more robust and maintainable software development process, catering to both big-picture design and detailed, test-driven implementation.

Aspect	Model-Driven Development (MDD)	Test-Driven Development (TDD)
Approach	High-level modeling to guide development	Writing tests before implementing the code
Perspective	Eagle's view (big picture, high-level structure)	Mouse's view (detailed, incremental steps)
Process	Starts with a comprehensive model (Example, decision tables)	Starts with writing test cases for specific functionality
Example	Using a decision table to derive the `isLeapYear` function	Incrementally developing the `isLeapYear` function based on test cases
Complexity Management	Structured approach with potentially higher initial complexity	Simpler initial approach, but may require multiple condition coverage
Cyclomatic Complexity	Higher cyclomatic complexity (Example, 4)	Lower cyclomatic complexity (Example, 2)
Fault Isolation	Ensures completeness and consistency through comprehensive models	Excellent fault isolation; failing tests indicate issues in the most recent changes
Development Effort	Initial longer development time due to detailed modeling	Potentially shorter initial development time but may require more iterations
Maintenance	Comprehensive models may aid in long-term maintenance (Eagle's view)	Test cases provide detailed fault isolation, helping in fault recreation (Mouse's view)
Skill Dependency	Relies on the developer's modeling skills	Relies on the developer's testing skills
Advantages	Ensures complete and consistent logical structure	Continuous validation and immediate feedback
Challenges	May require more initial effort and higher complexity	May miss deeper faults without thorough testing
Suitability for Large Applications	Can manage large applications through structured modeling	May struggle with scalability and complexity in very large applications
Support from Tools	Tools for modeling (Example, UML tools)	Extensive test frameworks available (Example, JUnit for Java)
Example Tool Support	UML, decision tables	JUnit, NUnit, pytest
Documentation	Models serve as documentation	Test cases act as live documentation

11.9 Software Testing Excellence

Software Testing Excellence refers to the highest standards and practices in the field of software testing to ensure that software is reliable, efficient, and meets both functional and non-functional requirements. Achieving excellence in software testing involves a combination of skills, methodologies, and attitudes that collectively enhance the quality of the software development process.

Different Views on Software Testing and Their Contribution to Excellence

Completing a project can be as challenging as starting one. In both writing and software development, there is a constant temptation to revisit and refine "finished" work. This temptation arises from the anxieties that surface as deadlines approach. In software testing, this tendency is prevalent, emphasizing the importance of continuous improvement to achieve excellence.

Over time, software testing tools and techniques have evolved significantly. To understand the current landscape of software testing, imagine a line that begins with Art, progresses through Craft, moves to Science, and ends in Engineering. Where does software testing fit on this line?

1. **Software Testing as an Engineering:** Tool vendors often view software testing as an engineering discipline. They argue that their tools automate and streamline the testing process, reducing the need for manual intervention and creativity.

   Contribution to Excellence: This perspective emphasizes precision, efficiency, and scalability through the use of advanced tools and methodologies. Automation and systematic approaches ensure that testing is thorough and repeatable, contributing to the overall quality of the software.

2. **Software Testing as a Science:** The process community considers software testing to be a science. They believe that following a well-defined process is essential for effective testing.

   Contribution to Excellence: This approach involves structured methods, repeatable procedures, and systematic experimentation and analysis. Scientific methods ensure that testing is reliable and comprehensive, leading to a higher standard of quality assurance.

3. **Software Testing as an Art:** The context-driven school regards software testing as an art. They emphasize the importance of creativity and individual skill in identifying defects.

   Contribution to Excellence: This perspective values innovative thinking, unique approaches, and the personal insights that experienced testers bring to the process. Creative testers uncover hidden issues and exploring edge cases that might be missed by more systematic approaches.

4. **Software Testing as a Craft:** Viewing software testing as a craft highlights the importance of skill and practice. Testers continuously develop their abilities and apply various tools and techniques to improve the quality of their work.

   Contribution to Excellence: This perspective values experience and the continuous improvement of testing skills through ongoing practice and learning. Skilled testers bring a depth of knowledge and a refined approach to testing, ensuring meticulous and high-quality results.

Understanding software testing through these different lenses helps in appreciating its complexity and importance. Whether viewed as an art, a craft, a science, or an engineering discipline, achieving excellence in software testing involves integrating the best aspects of each approach.

Achieving Software Testing Excellence

To achieve excellence in software testing, it is crucial to integrate the best aspects of various perspectives:

1. **Precision and Efficiency (Engineering):** Leverage advanced tools and methodologies to streamline the testing process, ensuring thorough and efficient testing.
2. **Structured and Reliable Methods (Science):** Adhere to well-defined processes and methodologies to maintain consistency and reliability throughout the testing phases.
3. **Creativity and Insight (Art):** Apply creative thinking and innovative approaches to uncover defects that might be missed by conventional testing methods.
4. **Skill and Continuous Improvement (Craft):** Continuously develop and refine testing skills through practice and experience, always striving for improvement and excellence.

11.9.1 Craftsmanship

Craft involves creating unique handmade objects using skill and expertise in a specific trade like pottery, knitting, woodworking, or jewelry making. Craftsmanship, on the other hand, refers to the quality and skill demonstrated in crafting these items, showcasing mastery, attention to detail, and dedication to producing high-quality and aesthetically pleasing products. For instance, a skilled potter showcases craft by shaping clay into functional pottery with precision and artistry, highlighting craftsmanship through smooth finishes, intricate designs, and overall quality in the final pieces.

Craftsmanship in Software Testing

Craftsmanship in software testing refers to the dedication to mastering the art and science of testing, with a focus on delivering high-quality software through skill, practice, and continuous improvement. A true craftsman in software testing is characterized by their commitment to excellence, attention to detail, and pride in their work. This approach emphasizes the human aspect of testing, where the tester's experience, intuition, and expertise play a crucial role.

Craftsmanship in software testing is about more than just following procedures; it's about bringing a high level of skill, dedication, and pride to the testing process. It involves mastering the tools and techniques, continuously improving practices, and delivering high-quality work consistently.

Key Attributes of Craftsmanship in Software Testing:

1. **Mastery of the Subject Matter:**
   - A deep and comprehensive understanding of software testing principles, methodologies, and best practices.
   - Continuous learning to stay updated with the latest trends, tools, and techniques in software testing.
2. **Mastery of Tools and Techniques:**
   - Proficiency with a wide range of testing tools, both manual and automated.
   - Expertise in various testing techniques such as black-box testing, white-box testing, exploratory testing, performance testing, and security testing.

3. **Ability to Make Appropriate Choices:**
   - Skill in selecting the right tools and techniques for the specific context of the project.
   - Understanding the trade-offs and implications of different testing strategies and making informed decisions.
4. **Extensive Experience:**
   - A wealth of hands-on experience in testing various types of software, from simple applications to complex systems.
   - Experience in different industries and domains, bringing a broad perspective to testing challenges.
5. **History of High-Quality Work:**
   - A proven track record of delivering high-quality work consistently.
   - Recognition from peers and stakeholders for the ability to find critical defects and improve software quality.
6. **Commitment to Continuous Improvement:**
   - Regularly reviewing and refining testing processes to enhance efficiency and effectiveness.
   - Seeking feedback and learning from past projects to continually improve testing skills and practices.
7. **Pride in Work:**
   - Taking pride in delivering thorough, meticulous, and reliable testing.
   - Striving for excellence in every aspect of the work, from planning and execution to reporting and communication.

Importance of Craftsmanship in Software Testing

1. **Quality Assurance:** High craftsmanship ensures that software is tested thoroughly, identifying and addressing defects before they reach the end users.
2. **Reliability:** Experienced testers can anticipate potential issues and design tests that ensure the software performs reliably under various conditions.
3. **User Satisfaction:** Well-tested software leads to higher user satisfaction, as it is less likely to have defects that disrupt the user experience.
4. **Cost Efficiency:** Identifying and fixing defects early in the development process saves time and resources, reducing the overall cost of software development.

11.9.2 Best Practices of Software Testing

Software testing is a important part of the software development lifecycle to ensure the quality, reliability, and performance of software products. Over the years, the software development field has seen numerous proposed solutions to its challenges. From high-level programming languages like FORTRAN and COBOL to modern methodologies such as Agile programming and test-driven development, the industry has explored various approaches to enhance software development

processes. Defining best practices in software testing can be subjective, but certain characteristics are commonly accepted: they are usually defined by practitioners, proven to be effective ("tried and true"), dependent on the subject matter, and have a significant history of success. Below is a list of notable best practices in software testing.

1. **High-Level Programming Languages:** The languages like FORTRAN and COBOL abstracted machine-level details, making it easier for developers to write and maintain code. They allowed for more efficient coding practices and reduced the likelihood of errors associated with low-level programming.
2. **Structured Programming:** This methodology improved code readability and maintainability by using clear control structures like loops and conditionals. It emphasized the use of blocks, functions, and subroutines to create well-organized code.
3. **Third-Generation Programming Languages:** The languages like C and Pascal further enhanced programming efficiency and capability by providing more sophisticated abstractions and control structures.
4. **Software Reviews and Inspections:** Peer reviews and formal inspections helped detect defects early in the development process. This practice ensured that many errors were caught before the software moved into later stages, reducing the cost and effort of fixing bugs.
5. **The Waterfall Model:** This linear and sequential approach to software development divided the process into distinct phases, such as requirements, design, implementation, testing, and maintenance. Each phase needed to be completed before moving on to the next.
6. **The Object-Oriented Paradigm:** This paradigm promoted the use of objects and classes to encapsulate data and behavior. It supported code reuse, modularity, and abstraction, leading to more maintainable and scalable software systems.
7. **Various Replacements for the Waterfall Model:** The models like Spiral model and V-Model introduced iterative and incremental approaches, allowing for more flexibility and the ability to refine requirements and designs through multiple iterations.
8. **Rapid Prototyping:** This approach involved quickly creating a working model of the software to validate requirements and design choices. It enabled early user feedback and facilitated better understanding of user needs.
9. **Software Metrics:** Metrics provided quantitative measures to assess software quality and progress. Common metrics included code coverage, defect density, and test case effectiveness.
10. **CASE (Computer-Aided Software Engineering) Tools:** These tools automated parts of the software development process, including design, coding, testing, and maintenance, thereby improving productivity and reducing human error.
11. **Commercial Tools for Project, Change, and Configuration Management:** These tools helped manage and track various aspects of software projects, such as changes in requirements, code versions, and project timelines, ensuring better control and organization.
12. **Integrated Development Environments (IDEs):** IDEs provided a unified environment that integrated various development tools, such as code editors, compilers, debuggers, and version control systems, streamlining the development process.

13. **Software Process Maturity and Assessment:** Models like CMMI (Capability Maturity Model Integration) evaluated and improved software development processes, leading to more predictable and high-quality outcomes.
14. **Software Process Improvement:** Continuous enhancement of development processes aimed to improve efficiency, reduce defects, and increase overall quality.
15. **Executable Specifications:** These specifications could be directly executed to verify that the software met the defined requirements, bridging the gap between specification and implementation.
16. **Automatic Code Generation:** Tools that automatically generated code from higher-level specifications increased productivity and reduced manual coding errors.
17. **UML (Unified Modeling Language):** UML provided standardized visual modeling of software systems, facilitating better design and communication among developers.
18. **Model-Driven Development (MDD):** MDD emphasized creating and utilizing domain models to drive the development process, improving consistency and reducing errors.
19. **Extreme Programming (XP):** XP promoted flexible, iterative development with practices such as pair programming, continuous integration, and frequent releases, enhancing software quality and responsiveness to change.
20. **Agile Programming:** Agile methodologies focused on iterative development, collaboration, and adaptability, allowing teams to respond quickly to changes and deliver high-quality software incrementally.
21. **Test-Driven Development (TDD):** TDD involved writing tests before code, ensuring that the software met its requirements and reducing defects early in the development process.
22. **Automated Testing Frameworks:** These frameworks improved the efficiency and coverage of testing by automating repetitive test cases, enabling faster feedback and more reliable software.

While the list of best practices is extensive, it is not exhaustive. Software development remains a challenging endeavor, and practitioners continuously seek new or improved methods to enhance quality and efficiency. Understanding and applying these best practices can help software testers and developers achieve better outcomes and contribute to the ongoing evolution of the field.

11.9.3 Top 10 Best Practices for Software Testing Excellence

Achieving excellence in software testing requires a deep understanding of the craft, access to the right tools, and sufficient time to perform tasks carefully. A good tester is creative, curious, disciplined, and a "can-I-break-it" mentality. Below are top 10 best practices for software testing.

1. **Model-Driven Agile Development:** Combining traditional model-driven development (MDD) with agile practices creates a powerful approach. Models help identify details that might be overlooked and are useful for maintenance, while agile practices like test-driven development (TDD) enhance flexibility and responsiveness.

   Example: Using UML diagrams to model the software architecture while applying TDD to develop features incrementally.

2. **Careful Definition and Identification of Levels of Testing :** Defining clear levels of testing, such as unit, integration, and system testing ensures that each level focuses on specific objectives and avoids redundancy.

   Example: Implementing unit tests to check individual functions, integration tests to verify the interaction between modules, and system tests to validate the entire application.

3. **System-Level Model-Based Testing :** Using executable specifications allows automatic generation of system-level test cases to ensure comprehensive testing aligned with requirements.

   Example: Generating test cases from a model specified in a tool directly links to the requirements.

4. **System Testing Extensions :** For complex systems, beyond basic thread testing, include thread interaction testing, stress testing, and risk-based testing to uncover deeper issues.

   Example: Applying stress testing to a banking application to reveal thread interaction faults under heavy load, and prioritizing tests based on the risk of failure.

5. **Incidence Matrices to Guide Regression Testing :** An incidence matrix records the relationships between features and procedures (or use cases and classes), guiding the order of builds, fault isolation, and regression testing.

   Example: Creating an incidence matrix for an e-commerce platform to track dependencies between user features and backend services, facilitating efficient regression testing.

6. **Use of MM-Paths for Integration Testing :** MM-paths, which track multiple modules' interaction paths, are effective for integration testing and can be used alongside incidence matrices for better test coverage.

   Example: Using MM-paths to test the integration of different services in a banking application ensures all interaction paths are validated.

7. **Intelligent Combination of Specification-Based and Code-Based Unit-Level Testing:** Combining specification-based testing (focusing on what the software should do) with code-based testing (focusing on how the software works) provides comprehensive coverage.

   Example: Writing unit tests for a payment processing module using both the specifications provided (to verify functionality) and examining the code to ensure all paths are tested.

8. **Code Coverage Metrics Based on the Nature of Individual Units :** Selecting appropriate code coverage metrics based on the specific characteristics of the code ensures relevant and effective testing.

   Example: For a critical security module, aiming for 100% branch coverage to ensure all possible execution paths are tested.

9. **Exploratory Testing during Maintenance :** Exploratory testing, where testers actively explore the software without predefined test cases, is especially useful for understanding and testing legacy code.

   Example: Conducting exploratory testing on an old inventory management system to identify hidden issues before implementing new features.

10. **Test-Driven Development (TDD) :** TDD involves writing tests before coding, ensuring excellent fault isolation and continuous integration of small, tested increments.

    Example: Developing a new feature for a mobile app by first writing unit tests that define the desired functionality, then writing the code to pass those tests, and iterating this process.

These best practices represent a combination of tried-and-true methods and innovative approaches that can significantly enhance the quality and effectiveness of software testing. By integrating these practices, testers can achieve a high level of software testing excellence to ensure that software is reliable, efficient, and meets user expectations.

11.10 Review Questions

Section - A

Each Question Carries Two Marks

1. What is Test Driven Development (TDD) ?
2. What is Test-Then-Code Cycle?
3. What do you mean by Automated Test Execution?
4. What are Testing Frameworks? Give an example.
5. What is JUnit?
6. Define Software Testing Excellence.
7. How to achieve Software Testing Excellence?

Section - B

Each Question Carries Five Marks

1. What is Test Driven Development (TDD) ? Explain the Features or Characteristics of TDD.
2. Mention the Goals and Purpose of Automated Test Execution.
3. Explain the Common Features of Testing Frameworks.
4. Explain the Importance of Testing Frameworks in TDD.
5. Write the Advantages and Disadvantages of Automated Test Execution.
6. Explain TDD with Java and JUnit Example.
7. What is JUnit? Explain Key Features of JUnit.
8. Is TDD Code Based or Specification Based? Discuss.

9. Is Configuration Management Challenging in TDD? Discuss.
10. How Does Granularity Affect the TDD Process?
11. Write a note on Advantages, Disadvantages and Open Questions of TDD.
12. Explain the different views on Software Testing and Their Contribution to Excellence.
13. Write the Importance of Craftsmanship in Software Testing.
14. What are the Key Attributes of Craftsmanship in Software Testing?

Section - C

Each Question Carries Eight Marks

1. What is Test Driven Development (TDD) ? Explain the Features, Advantages and Disadvantages of TDD.
2. Explain Test-Then-Code Cycles With Detailed Example.
3. Explain TDD with Java and JUnit Example.
4. Differentiate Between Model Driven Development versus Test Driven Development.
5. Explain Craftsmanship in Software Testing. Write the Importance of Craftsmanship in Software Testing.
6. Explain the Best Practices of Software Testing.
7. Write Top 10 Best Practices for Software Testing Excellence.

❖❖❖❖❖

Model Question Papers — Appendix - A

Model Question Paper - 1

Time : 2.5 Hours **Max. Marks : 60**

Instructions : Answer All Sections

Section -A

I. Answer any Four questions. Each question carries Two marks **(4 X 2 = 8)**

1. Define Software Testing.
2. What is Equivalence Class Testing (ECT)? Give an example.
3. What is Define-Use Testing?
4. What is Top- Down Integration Testing?
5. What is Client Server Testing ?
6. What is Object Oriented Testing ?

Section -B

II. Answer any Four question. Each question carries Five marks **(4 x 5 = 20)**

7. What is Boundary Value Testing ? Explain the Importance of Boundary Value Testing?
8. Differentiate Between Weak Normal Vs Strong Normal Equivalence Class Testing.
9. Write the Advantages and Disadvantages of Equivalence Class Testing.
10. How Pair wise Integration Testing Works?
11. What is GUI Testing? Explain GUI Testing Strategies.
12. Write the Importance of Craftsmanship in Software Testing.

Section - C

III. Answer any Four questions. Each question carries Eight marks **(4 X 8 = 32)**

13. Explain the Testing Life Cycle.
14. (a) Explain Robust Worst-Case Boundary Value Testing (RWCBVT) with an example.
    (b) What is Special Value Testing? Explain Characteristics of Special Value Testing.
15. (a) Explain Equivalence Class Test Cases for the NextDate Function
    (b) Write the Advantages and Disadvantages of Def-Use Testing.
16. (a) How Bottom-Up Integration Testing Works? Explain with an example.
    (b) Discuss the Basic Concepts for Requirements Specification in System Testing.
17. (a) Explain Static Interactions in a Single Processor with detailed example.
    (b) Explain Exploratory and Context-Driven Testing Observations
18. What is Test Driven Development (TDD) ? Explain the Features, Advantages and Disadvantages.

❖❖❖❖❖

Model Question Paper - 2

Time : 2.5 Hours **Max. Marks : 60**

Instructions : Answer All Sections

SECTION -A

I. Answer any Four questions. Each question carries Two marks (4 X 2 = 8)

1. Define Test and Testcase.
2. What is Decision Table Based Testing?
3. What is Slice Based Testing?
4. What is Call Graph-Based Integration?
5. What is Exploratory Testing ?
6. What is Test-Then-Code Cycle?

SECTION -B

II. Answer any Four question. Each question carries Five marks (4 x 5 = 20)

7. Explain the Importance of Software Testing.
8. Differentiate Between Weak Robust Vs Strong Robust Equivalence Class Testing.
9. How Def-Use Testing Works? Explain with an example.
10. What is Atomic System Function (ASF)? Explain the Importance of ASFs.
11. Explain Object-Oriented Integration Testing.
12. What is Test Driven Development (TDD) ? Explain the Features or Characteristics of TDD.

SECTION - C

III. Answer any Four questions. Each question carries Eight marks (4 X 8 = 32)

13. Explain Fundamental Approaches to Apply Test Cases with examples.
14. (a) Explain Worst-Case Boundary Value Testing (WCBVT) with an example.
    (b) Explain Equivalence Class Test Cases for the Commission Problem.
15. (a) Explain the Guidelines and Observations of Decision Table Testing.
    (b) What is Data Flow Testing? Explain the Characteristics of Data Flow Testing.
16. (a) How Path-Based Integration Testing Works?
    (b) Explain the Characteristics and Importance of a Thread in System Testing.
17. (a) Explain Static Interactions in Multiple Processors with detailed example.
    (b) What is Use Case Based Testing ? Explain the Steps in Use Case Based Testing.
18. (a) Explain TDD with Java and JUnit Example.
    (b) Explain Exploratory and Context-Driven Testing Observations

❖❖❖❖❖

Model Question Paper - 3

Time : 2.5 Hours **Max. Marks : 60**

Instructions : Answer All Sections

SECTION -A

I. Answer any Four questions. Each question carries Two marks (4 X 2 = 8)

1. What is Specification-Based Testing? Give an example
2. What is Weak Robust Equivalence Class Testing (WRECT) ?
3. Define du-path and dc-path.
4. What is Path-Based Integration Testing?
5. What is Use Case Based Testing ?
6. What is Test Driven Development (TDD) ?

SECTION -B

II. Answer any Four question. Each question carries Five marks (4 x 5 = 20)

7. Explain Normal Boundary Value Testing (NBVT) with an example.
8. Explain the Generation of Test Cases for the NextDate Function in Boundary Value Testing.
9. Discuss the Guidelines and Observations for Data Flow Testing.
10. What is System Testing? What are the Key Objectives of System Testing.
11. Explain the Issues in Object Oriented Testing.
12. Explain the Process of Testing based on Models

SECTION - C

III. Answer any Four questions. Each question carries Eight marks (4 X 8 = 32)

13. (a) Explain the Components of a Test Case.
    (b) Explain the Limitations of Boundary Value Analysis.
14. (a) Explain Random Testing. Mention its advantages and disadvantages.
    (b) Explain the Guidelines and Observations About Equivalence Class Testing.
15. (a) Explain Decistion Table Design Format with an example.
    (b) Explain the Importance or Benefits of Slice-Based Testing.
16. (a) How Neighborhood Integration Testing Works?
    (b) Explain the General Procedure for Finding Threads in System Testing.
17. (a) Explain Dynamic Interactions in a Single Processor with detailed example.
    (b) Explain the Features or Characteristics of Model-Based Testing (MBT).
18. Write Top 10 Best Practices for Software Testing Excellence.

❖❖❖❖❖

Model Question Paper - 4

Time : 2.5 Hours **Max. Marks : 60**

Instructions : Answer All Sections

SECTION -A

I. Answer any Four questions. Each question carries Two marks **(4 X 2 = 8)**

1. What is Code Based Testing? Give an example.
2. What is Data Flow Testing ?
3. What is Pair wise Integration Testing?
4. What is a Thread in System Testing ?
5. What are Taxonomy of interactions?
6. What is Model Based Testing (MBT) ?

SECTION -B

II. Answer any Four question. Each question carries Five marks **(4 x 5 = 20)**

7. Explain the Levels of Testing in V-Model.
8. What is Decision Table? Explain the Characteristics of Decision Tables.
9. Discuss the Commission Problem using Define-Use Testing.
10. Explain the Features or Characterisers of System Testing.
11. Explain the Context of Interaction in Interaction Testing.
12. Differentiate Between Model Driven Development versus Test Driven Development

SECTION - C

III. Answer any Four questions. Each question carries Eight marks **(4 X 8 = 32)**

13. (a) Explain Generalizing Boundary Value Analysis (BVA).
    (b) Explain Robust Boundary Value Testing (RBVT) with an example.
14. Explain the Forms or Variations of Equivalence Class Testing with examples.
15. (a) Explain the Test Cases for the Triangle Problem in Decision Table Based Testing.
    (b) How Sandwich Integration Testing Works? Explain with an example.
16. (a) Explain Dynamic Interactions in Multiple Processors with detailed example.
    (b) What is Client Server Testing ? Explain the Testing Startegy.
17. (a) Explain the process of Finding Threads in the SATM System.
    (b) What is Lewis and Clark's Expedition ? Compare Exploratory Testing and the Lewis and Clark Expedition.
18. Explain Test-Then-Code Cycles With Detailed Example

❖❖❖❖❖