Software Testing

A Perspective on Testing

- **Why do we test?**
  - Two primary reasons: to make a judgment about quality or acceptability, and to discover problems.
  - Testing is necessary because humans are fallible, particularly in the domain of software and software-controlled systems.
- **Purpose of Testing**
  - Quality assessment: Assessing the quality or acceptability of the software product.
  - Problem discovery: Identifying and addressing issues and defects within the software.

- **Significance in Software Domain**
  - Software and software-controlled systems are prone to errors and vulnerabilities due to their complexity.
  - Testing serves as a critical process to identify and rectify these errors, ensuring software reliability and functionality.
- **Objective of the Chapter**
  - Establish a framework for examining software testing.
  - Provide a structured approach to understanding the principles and practices of software testing.

Basic Definitions

- **Complexity of Testing Terminology**
  - Testing literature often contains confusing and inconsistent terminology.
  - Evolution of testing technology over decades and contributions from numerous authors contribute to this complexity.
- **ISTQB Glossary**
  - The International Software Testing Qualification Board (ISTQB) offers an extensive glossary of testing terms.
  - ISTQB's glossary provides standardized definitions to mitigate confusion in testing terminology.

- **Compatibility with Standards**
  - The terminology used in the syllabus, as well as in the ISTQB glossary, aligns with definitions established by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society.
- **Progression of Terms**
  - The syllabus offers a useful progression of terms to facilitate understanding and navigation through testing concepts and principles.

Error

- Definition: Errors are human actions that result in deviation from intended behavior or specification.
- Example: A programmer mistakenly writes incorrect code that causes the software to crash when certain inputs are provided.
- Synonym: Mistake

- **Bug**
- Definition: Bugs are errors in software code resulting from human mistakes during programming.
- Example: The software fails to execute a specific function as intended due to an error in the code, resulting in unexpected behavior.
- Relation to Error: Errors in coding are often referred to as bugs, reflecting the unintentional deviation from desired functionality.

Fault

- Definition: A fault represents an error in the software system. It is the manifestation or expression of an error, conveyed through various modes of expression such as narrative text, diagrams, charts, and source code.
- Synonyms: Defect, Bug
- Example: In a software application, a fault may be evident in the source code as a logical error that causes the program to produce incorrect outputs.

- **Elusiveness of Faults**
  - Faults can be elusive, making them challenging to detect and resolve.
  - Example: A subtle error in the code logic may only manifest under specific conditions or input scenarios, making it difficult to identify during testing.

- **Faults of Commission vs. Faults of Omission**
  - **Faults of Commission**: Occur when incorrect information is entered into the representation.
    - Example: A programmer mistakenly assigns the wrong variable value in the source code, leading to a fault that causes unexpected behavior.
  - **Faults of Omission**: Occur when correct information is missing from the representation.
    - Example: Failure to include a necessary condition or requirement in the software design documentation results in a fault of omission.

- **Difficulty of Detecting and Resolving Faults of Omission**
  - Faults of omission are more challenging to detect and resolve compared to faults of commission.
  - Example: Identifying missing requirements or specifications in the early stages of development requires thorough analysis and collaboration among stakeholders.

Failure

- Definition: A failure occurs when the code associated with a fault executes, resulting in observable deviations from the expected behavior of the software.
- Example: When a user inputs certain data into the software, and the program crashes or produces incorrect output due to a fault in the code, a failure occurs.

- **Subtleties of Failure**
  - Failures are observed in executable representations, typically source code or loaded object code.
  - Failures are directly linked to faults of commission, where incorrect information is present in the representation.

- **Challenges with Faults of Omission**
  - Failures resulting from faults of omission pose challenges as they involve missing or incomplete information in the representation.
  - Example: If a critical requirement is omitted from the software design documentation, it may lead to a failure when the software is executed in real-world scenarios.
- **Addressing Faults of Omission**
  - Reviews play a crucial role in preventing failures by identifying faults, including those of omission.
  - Well-executed reviews have the potential to detect and address faults of omission before they lead to failures in the software.

- **Review Process**
  - Reviews involve thorough examination and analysis of software artifacts, such as requirements documents, design specifications, and source code.
  - By identifying and rectifying faults early in the development process, reviews help mitigate the risk of failures in the software.

Incident

- Definition: An incident is the symptom or observable manifestation associated with a failure that alerts the user, customer, or tester to the occurrence of the failure.
- Example: In a software application, an incident could be a system crash, an error message, or unexpected behavior that indicates a failure has occurred.
- Importance: Incidents serve as indicators of potential issues within the software, prompting further investigation and resolution by the development or testing team.

Test

- Definition: A test is the process of exercising software by executing predefined test cases.
- Purpose: Tests are conducted with two distinct goals:
  - Finding failures: Identifying errors, faults, or defects within the software that result in observable failures or incidents.
  - Demonstrating correct execution: Verifying that the software behaves as expected under specified conditions, indicating that it meets the desired requirements and specifications.

Test Case

- Definition: A test case is a structured set of inputs, execution conditions, and expected outcomes designed to test a specific program behavior or functionality.
- Components:
  - Identity: Each test case is uniquely identified to facilitate tracking and management.
  - Associated Program Behavior: Test cases are linked to specific program behaviors or functionalities that they aim to verify.
  - Inputs and Expected Outputs: Test cases include a set of inputs or stimuli and the corresponding expected outputs or responses.

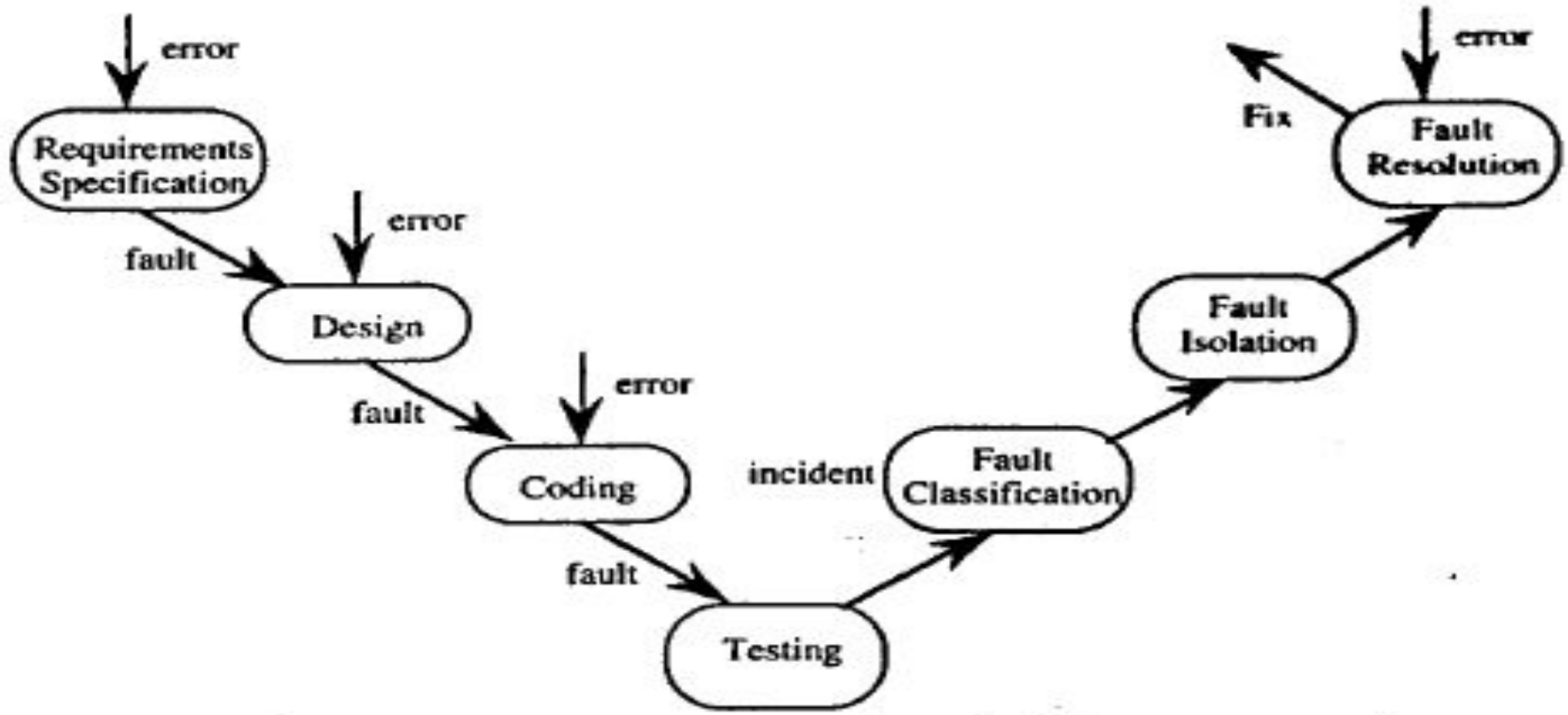
Importance of Test Cases

- Central Position in Testing: Test cases play a crucial role in the testing process, serving as the primary means to verify the correctness and reliability of software.
- Testing Process Steps:
  - Test Planning: Involves defining objectives, selecting appropriate test cases, and outlining testing strategies.
  - Test Case Development: Involves creating structured test cases based on requirements and specifications.
  - Running Test Cases: Involves executing the test cases against the software under test to evaluate its behavior.
  - Evaluating Test Results: Involves analyzing the outcomes of test case executions to assess software quality and identify defects.

Life Cycle Model for Testing

- Figure 1.1 portrays a life cycle model for testing, illustrating the various phases involved in the testing process.
- Opportunities for Error: The development phases present opportunities for errors to occur, resulting in faults that may propagate through subsequent stages.
- Fault Resolution: The process of resolving faults introduces additional opportunities for errors to occur, potentially leading to new defects or issues.

1.1



A testing life cycle.

Test Case Essentials

- Definition: A test case is a recognized work product in software testing.
- Components of a Complete Test Case:
  - Test Case Identifier
  - Brief Statement of Purpose (e.g., a business rule)
  - Description of Preconditions
  - Actual Test Case Inputs
  - Expected Outputs
  - Description of Expected Postconditions
  - Execution History (for test management use)

- **Importance of Output**
  - The output portion of a test case is often overlooked but is crucial, especially in complex scenarios.
  - Example: In testing software for determining optimal aircraft routes, defining the expected optimal route is challenging. Solutions like reference testing, involving expert judgments, can help address this challenge.

- **Test Case Execution Process**
  - Test case execution involves several steps:
    - Establishing necessary preconditions
    - Providing test case inputs
    - Observing outputs
    - Comparing observed outputs with expected outputs
    - Ensuring expected postconditions exist to determine test success or failure.

- **Value of Test Cases**
- Test cases are as valuable as source code in the software development process.
- They need to be developed, reviewed, used, managed, and saved to ensure effective testing and quality assurance.

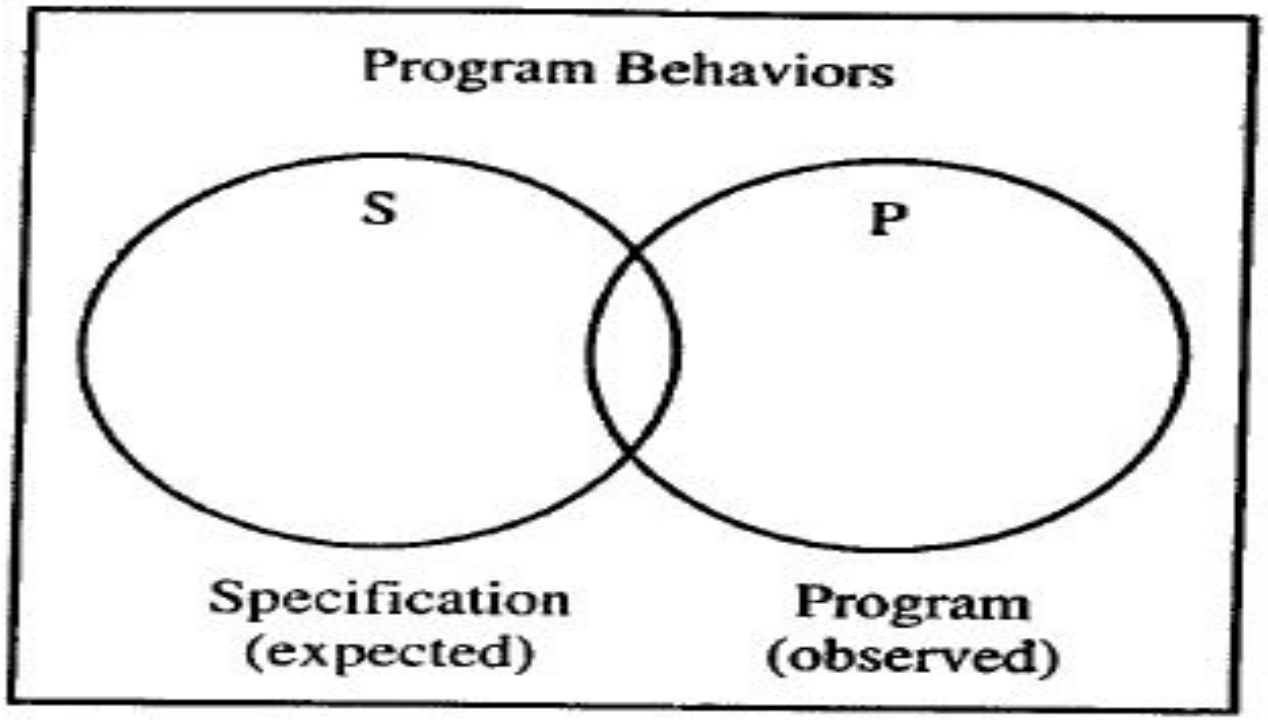
Venn Diagram

- **Fundamental Focus of Testing**
  - Testing is primarily concerned with behavior rather than the code-based perspective common among software developers.
  - Distinction: The code-based view focuses on what the software is (its structure and implementation), while the behavioral view considers what it does (its functionality and behavior).

- **Source of Difficulty for Testers**
  - Testers often encounter challenges because the base documents, such as requirements and specifications, are typically written by and for developers.
  - Emphasis on Code-based Information: Base documents tend to prioritize code-based information over behavioral aspects, making it challenging for testers to derive the expected behavior solely from these documents.
- **Role of a Venn Diagram**
  - A simple Venn diagram is introduced to clarify several persistent questions about testing.
  - The diagram helps illustrate the relationship between code-based information and behavioral aspects, providing insights into the intersection and differences between the two perspectives.

- **Universe of Program Behaviors**
  - Testing focuses on understanding the essence of program behaviors.
  - Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors.
- **Relationship between Specified and Programmed Behaviors**
  - Figure 1.2 illustrates the relationship between specified and programmed behaviors.
  - Specified behaviors are represented in the circle labeled S, while programmed behaviors are in circle P.
  - The intersection of S and P represents the "correct" portion, where behaviors are both specified and implemented.

1.2



Specified and implemented program behaviors.

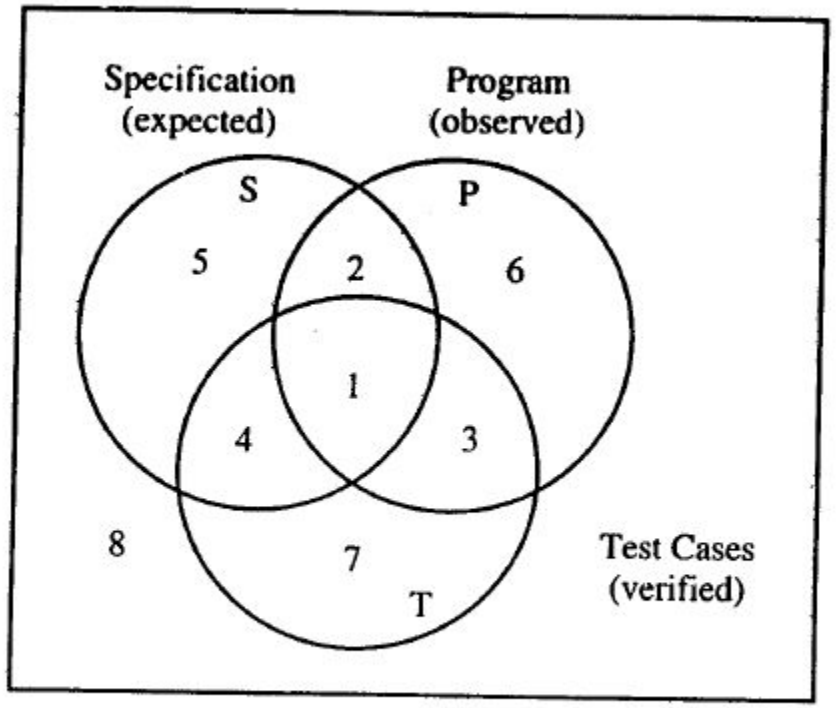
- **Challenges Confronting Testers**
  - Testers face challenges when certain specified behaviors are not programmed (faults of omission) or when certain programmed behaviors are not specified (faults of commission).
  - Intersection of S and P: Represents behaviors that are both specified and implemented, which is the desired outcome of testing.
- **View of Testing**
  - Testing is viewed as the determination of the extent of program behavior that is both specified and implemented.
  - "Correctness" is relative and depends on the alignment between specification and implementation.

Introduction of Test Cases

- Figure 1.3 introduces a new circle representing test cases in addition to the circles for specified (S) and programmed (P) behaviors.

- **Relationships Among Sets S, P, and T**
  - Specified behaviors that are not tested:
    - Region 2: Specified behaviors that are not covered by test cases.
  - Specified behaviors that are tested:
    - Region 1: Specified behaviors that are covered by test cases.
  - Test cases corresponding to unspecified behaviors:
    - Region 3: Test cases that cover behaviors not specified.
  - Programmed behaviors that are not tested:
    - Region 6: Programmed behaviors that are not covered by test cases.
  - Programmed behaviors that are tested:
    - Region 1: Programmed behaviors that are covered by test cases.
  - Test cases corresponding to behaviors that were not implemented:
    - Region 7: Test cases that cover behaviors that were not implemented.

1.3



Specified, implemented, and tested behaviors.

- **Discrepancy with the Universe of Discourse**
  - There may be a slight discrepancy between the universe of discourse and the sets of program behaviors and test cases because a test case causes a program behavior.
  - This discrepancy is understandable, given that test cases induce program behaviors and may not perfectly align with the specified and programmed behaviors.

Significance of Each Region

- Region 2 (Specified behaviors not tested): Incomplete testing occurs when specified behaviors exist but are not covered by test cases. This highlights the importance of ensuring comprehensive test coverage to address all specified behaviors.

- Region 3 (Test cases covering unspecified behaviors): Several possibilities arise:
  - Unwarranted test case: Test cases covering behaviors not specified may be unnecessary and could lead to inefficient testing practices.
  - Deficient specification: The presence of test cases corresponding to unspecified behaviors may indicate deficiencies in the specification document, necessitating clarification or revision.
  - Tester's objective: Testers may deliberately include test cases for behaviors not specified to verify that certain non-behaviors do not occur. This practice demonstrates the importance of skilled testers in identifying potential gaps or ambiguities in the specification.

- Involvement of Testers in Specification and Design Reviews:
  - Good testers often participate in specification and design reviews to contribute their insights and propose test cases, particularly those aimed at verifying non-behaviors or edge cases.
  - Their involvement helps improve the quality of specifications and ensures that test cases align with the intended behavior of the software.

Identifying Test Cases

- **Two Fundamental Approaches**
  - Traditional terms: Functional testing and structural testing.
  - More descriptive terms: Specification-based and code-based approaches.

- **Specification-Based Approach**
  - Focuses on testing against the specified requirements and functionality.
  - Test case identification methods align with the specified behavior of the software.

- **Code-Based Approach**
  - Focuses on testing the implementation of the software.
  - Test case identification methods align with the code structure and logic.

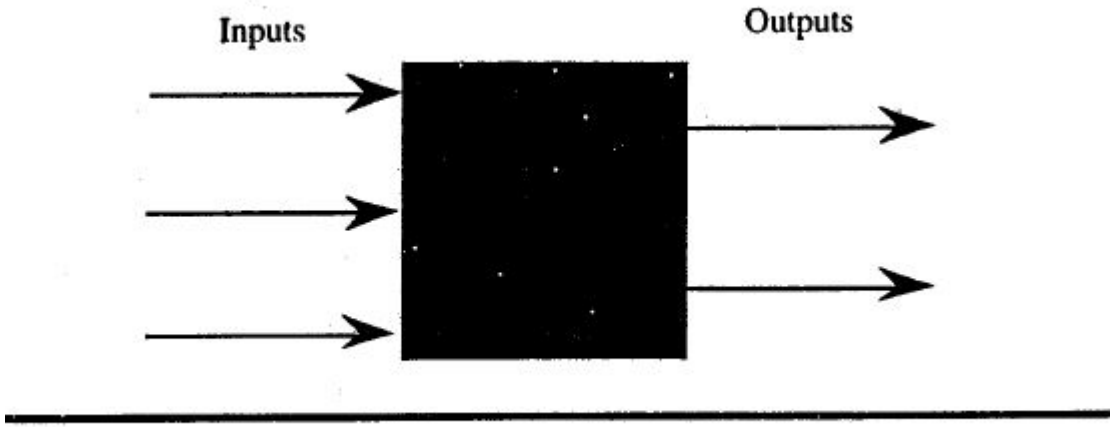
- **Methodical Approach**
  - Test case identification methods are generally referred to as testing methods.
  - These methods are systematic and methodical, ensuring consistency in test case generation.
  - Two testers following the same method are likely to devise very similar or equivalent test cases.
- **Equivalence of Test Cases**
  - Testers following the same method may devise equivalent test cases, ensuring thorough coverage of the software's behavior and implementation.

Specification-Based Testing

- **Functional View of Programs**
  - Programs can be viewed as functions that map inputs from their domain to outputs in their range.
  - This perspective is common in engineering, especially when treating systems as black boxes.

Black Box Testing

- Synonymous with specification-based testing.
- In black box testing, the internal implementation of the system (the "black box") is not known to the tester.
- The function of the black box is understood solely in terms of its inputs and outputs.



An engineer's black box.

- **Advantages:**
  - **Independence from Implementation:** Test cases are solely based on the software specification, making them independent of how the software is implemented. This means that even if the implementation changes, the test cases remain relevant and useful.
  - **Parallel Development:** Test case development can occur concurrently with the implementation of the software. This parallel development approach helps in reducing the overall project development interval, leading to potentially faster time-to-market.

- **Disadvantages:**
  - **Redundancies:** Specification-based test cases may suffer from significant redundancies, where multiple test cases cover similar or overlapping scenarios. This redundancy can lead to inefficiencies in testing.
  - **Possibility of Gaps:** There is a risk of gaps in test coverage, where certain aspects of the software may remain untested. This can occur due to oversight or limitations in the specification's coverage.

- **Comparison of Test Case Results**
  - Figure 1.5 depicts the results of test cases identified by two specification-based methods, Method A and Method B.
  - Method A identifies a larger set of test cases compared to Method B.
  - Both methods generate test cases that are completely contained within the set of specified behavior.
- **Limitation of Specification-Based Methods**
  - Because specification-based methods rely solely on the specified behavior, it is challenging for them to identify behaviors that are not explicitly specified.
  - These methods focus on testing against the specified requirements and functionality, which inherently limits their ability to identify unspecified behaviors.

Figure 1.5

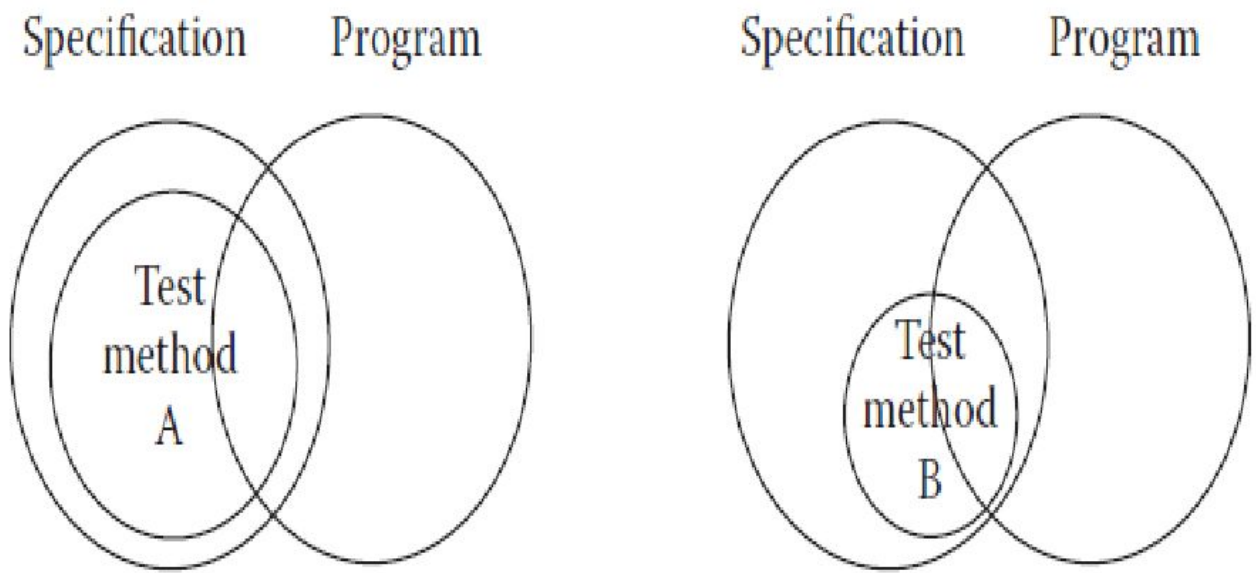


Figure 1.5 Comparing specification-based test case identification methods.

1.4.2 Code-Based Testing

- **Fundamental Approach:**
  - Code-based testing is another fundamental approach to test case identification.
  - It contrasts with black box testing and is sometimes referred to as white box or clear box testing.
- **White Box Testing:**
  - White box testing emphasizes the ability to see inside the black box, referring to the knowledge of the internal implementation of the software.
  - Another term for white box testing is clear box testing, which may be more appropriate as it highlights the transparency of the internal implementation.

- **Theoretical Basis:**
  - Code-based testing is grounded in strong theories, requiring familiarity with concepts such as linear graph theory for a deeper understanding.
- **Description of Tested Elements:**
  - With knowledge of linear graph theory concepts, testers can rigorously describe exactly what aspects of the code are being tested.
- **Test Coverage Metrics:**
  - Code-based testing facilitates the definition and use of test coverage metrics.
  - Test coverage metrics provide a quantitative measure of the extent to which a software item has been tested.

- **Benefits of Test Coverage Metrics:**
  - Test coverage metrics allow for the explicit statement of the level of testing achieved.
  - They make testing management more meaningful by providing insights into the completeness and thoroughness of the testing process.
- **Management of Testing:**
  - Test coverage metrics enable testing managers to make informed decisions about the testing process.
  - They help identify areas of the code that require further testing and prioritize testing efforts effectively.

- **Comparison of Test Case Results:**
  - Figure 1.6 depicts the results of test cases identified by two code-based methods, Method A and Method B.
  - Method A identifies a larger set of test cases compared to Method B.
- **Evaluation of Test Case Quantity:**
  - The question arises whether a larger set of test cases is necessarily better.
  - Code-based testing provides important ways to develop an answer to this question.

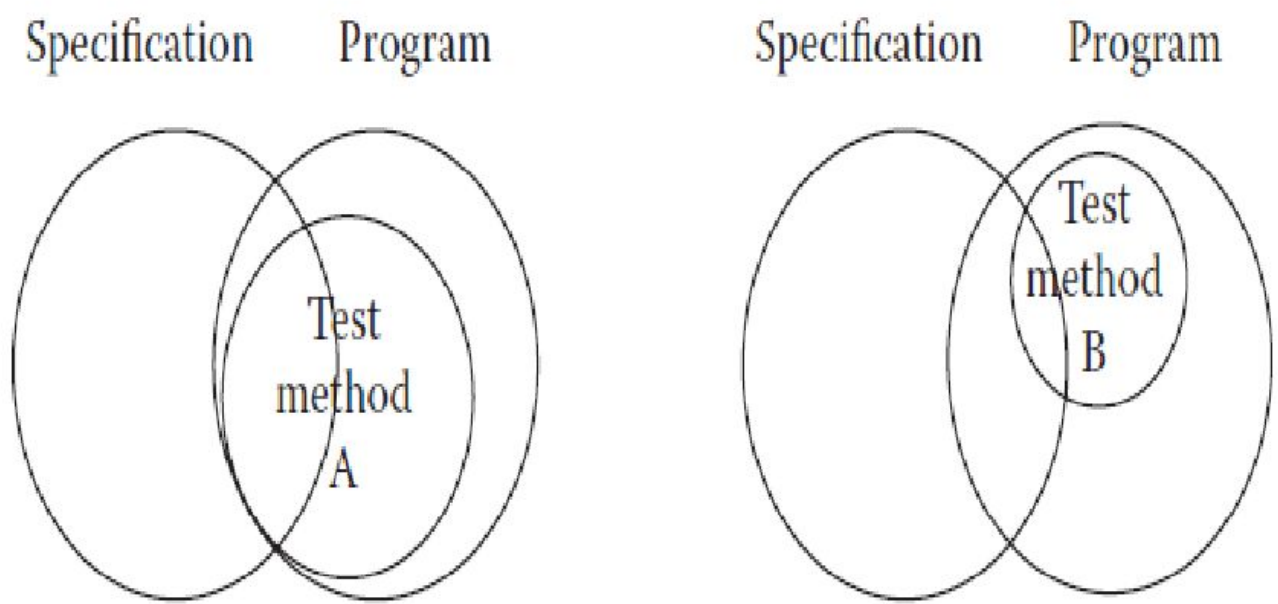


Figure 1.6 Comparing code-based test case identification methods.

- **Containment within Programmed Behavior:**
  - For both methods, the set of test cases is completely contained within the set of programmed behavior.
  - Code-based methods focus on the program itself, making it unlikely for them to identify behaviors that are not programmed.
- **Relative Size of Code-Based Test Cases:**
  - It is conceivable that a set of code-based test cases is relatively small compared to the full set of programmed behaviors.
  - This raises considerations about the effectiveness and efficiency of code-based testing in achieving sufficient test coverage.

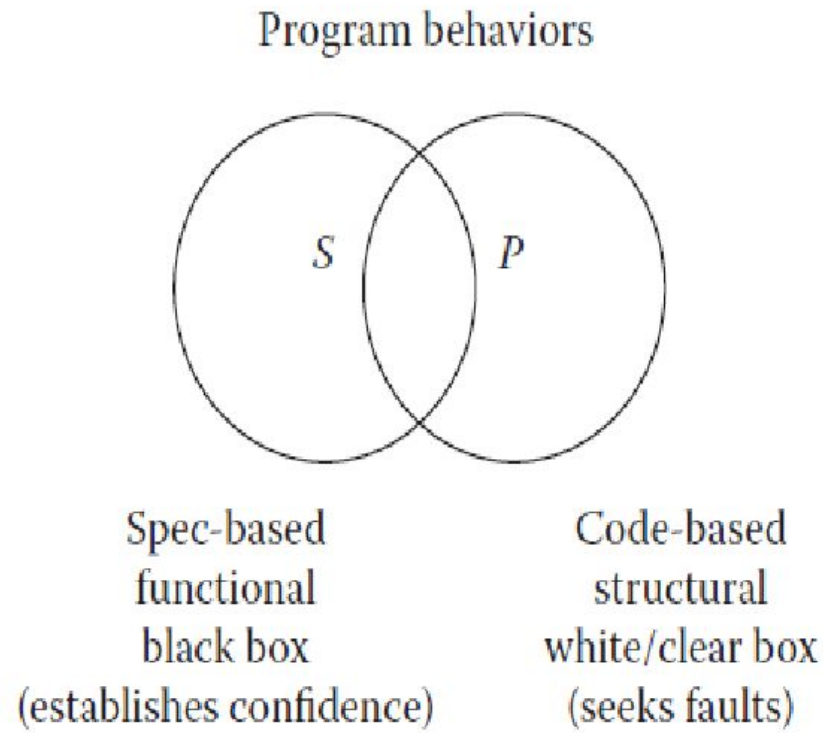


Figure 1.7 Sources of test cases.

Specification-Based versus Code-Based Debate

- **Fundamentally Different Approaches:**
  - Specification-based and code-based testing are two distinct approaches to test case identification.
  - Each approach uses different methods and criteria for identifying test cases.
- **Debate Over Superiority:**
  - There is a longstanding debate in the literature about which approach is better.
  - Strong adherents can be found for both specification-based and code-based testing.
- **Resolution from Venn Diagrams:**
  - The Venn diagrams presented provide a strong resolution to this debate.
  - Both approaches aim to identify test cases, but they use different bases: specification for specification-based testing and program source code (implementation) for code-based testing.

- **Insufficiency of Individual Approaches:**
  - Neither approach alone is sufficient to ensure comprehensive test coverage.
  - If specified behaviors are not implemented, code-based testing will not detect this.
  - Conversely, if the program implements behaviors not specified, specification-based testing will not uncover them.

- **Need for a Combination Approach:**
  - Both specification-based and code-based approaches are necessary.
  - A judicious combination of both approaches is advocated.
  - This combination provides the confidence of specification-based testing and the measurement capability of code-based testing.
- **Benefits of Combined Approach:**
  - When specification-based test cases are executed in combination with code-based test coverage metrics:
    - Redundancies and gaps in specification-based testing can be identified and resolved.
    - The strengths of each approach complement each other, leading to more effective testing.

- **Relationship Between Sets T, S, and P:**
  - The Venn diagram view of testing illustrates the relationship between the sets of test cases (T), specified behaviors (S), and implemented behaviors (P).
  - Test cases in set T are determined by the test case identification method used.
- **Appropriateness and Effectiveness of Test Case Identification Method:**
  - The appropriateness or effectiveness of the test case identification method used is a crucial consideration.
  - It determines the quality and coverage of the test cases generated.

- **Causal Trail from Error to Incident:**
  - Recalling the causal trail from error to fault, failure, and incident, understanding the types of errors and faults is essential.
  - Knowledge of potential errors and faults helps in employing more appropriate test case identification methods.
- **Testing as a Craft:**
  - The ability to select and apply the most suitable test case identification methods based on the nature of errors and faults reflects the craft of testing.
  - Testing transcends mere methodology and becomes an art form that requires skill, experience, and intuition.

Fault Taxonomies

- **Definition of Error and Fault:**
  - Error refers to a mistake made in the process, while fault is the manifestation of that error in the product.
  - Process relates to how something is done, while product is the end result of the process.
- **Testing and Software Quality Assurance (SQA):**
  - Testing is more product-oriented, focused on discovering faults in the product.
  - SQA aims to improve the product by enhancing the development process to reduce errors.
- **Product Orientation of Testing:**
  - Testing is primarily concerned with identifying faults in the product.
  - It focuses on evaluating the product's behavior and functionality to uncover defects.

- **Process Improvement in SQA:**
  - SQA focuses on improving the development process to prevent errors from occurring in the product.
  - By enhancing the process, SQA aims to minimize the occurrence of faults and improve overall software quality.
- **Classification of Faults:**
  - Faults can be classified based on various criteria, such as the development phase where the error occurred, consequences of corresponding failures, difficulty to resolve, and risk of no resolution.
  - An alternative classification method is based on anomaly occurrence: one-time-only, intermittent, recurring, or repeatable.
- **Importance of Clear Definitions:**
  - Both testing and SQA benefit from clear definitions of types of faults.
  - Clear classification helps in understanding and addressing different types of faults effectively.

IEEE Standard Classification for Software Anomalies and fault classifications:

- **IEEE Standard Classification for Software Anomalies:**
  - The IEEE Standard Classification for Software Anomalies provides a comprehensive treatment of different types of faults.
  - An anomaly, as defined in the document, is "a departure from the expected."
- **Anomaly Resolution Process:**
  - The IEEE standard outlines a detailed anomaly resolution process consisting of four phases: recognition, investigation, action, and disposition.
  - This process forms another life cycle in software development aimed at identifying, investigating, addressing, and resolving anomalies.
- **Useful Anomalies:**
  - Tables 1.1 through 1.5 in the IEEE standard present various useful anomalies categorized based on their characteristics and impact.
  - These anomalies serve as reference points for identifying and addressing faults during the software development lifecycle.

Tables 1.1

1. Mild	Misspelled word
2. Moderate	Misleading or redundant information
3. Annoying	Truncated names, bill for $0.00
4. Disturbing	Some transaction(s) not processed
5. Serious	Lose a transaction
6. Very serious	Incorrect transaction execution
7. Extreme	Frequent "very serious" errors
8. Intolerable	Database corruption
9. Catastrophic	System shutdown
10. Infectious	Shutdown that spreads to others

Faults classified by severity.

Tables 1.2

Table 1.1 Input/Output Faults

Type	*Instances*
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Tables 1.3

Table 1.2 Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

Tables 1.4

Table 1.3 Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Table 1.4 Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

Tables 1.5

Table 1.5 Data Faults

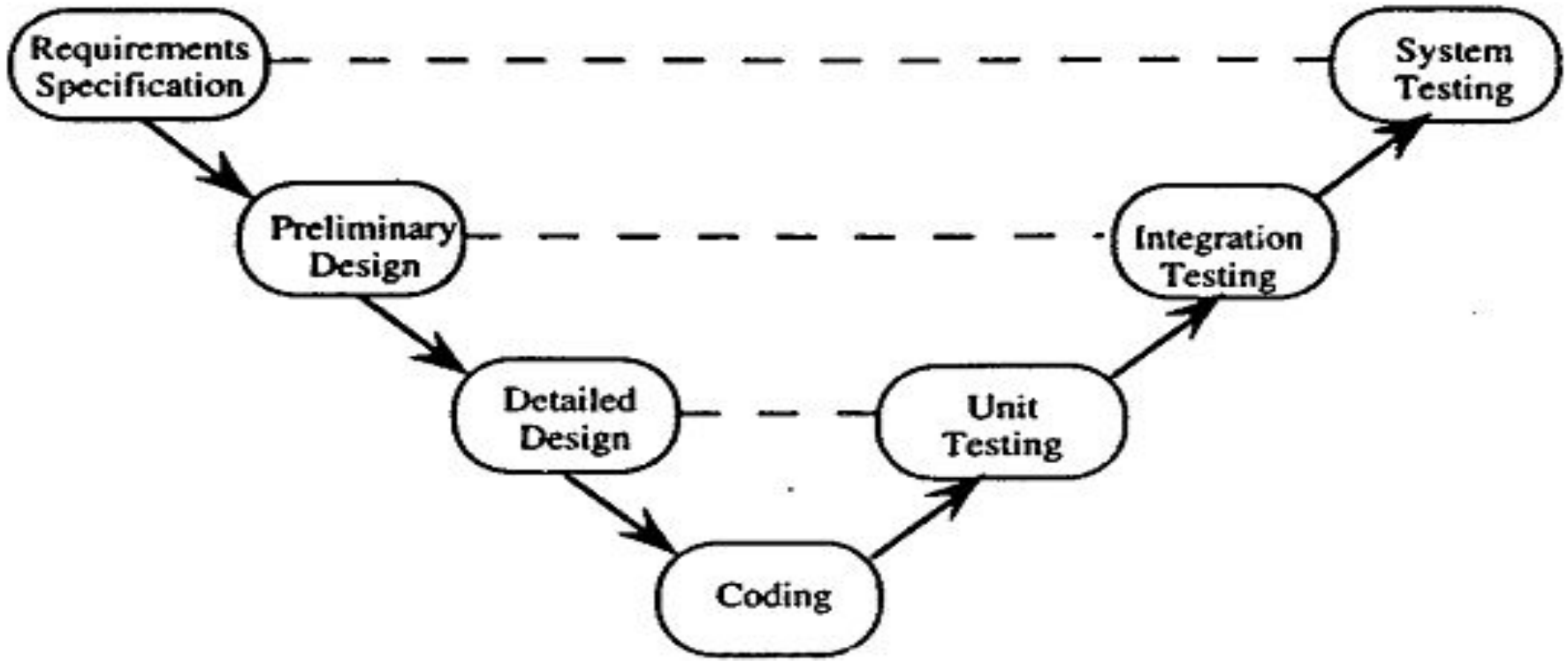
Incorrect initialization
Incorrect storage/access
Wrong flag/index value
Incorrect packing/unpacking
Wrong variable used
Wrong data reference
Scaling or units error
Incorrect data dimension
Incorrect subscript
Incorrect type
Incorrect data scope
Sensor data out of limits
Off by one
Inconsistent data

- **Software Review Checklists:**
  - Software reviews are essential for identifying faults in software artifacts.
  - Review checklists, such as those provided by Karl Wiegers on his website, offer another source of fault classifications and guidelines for conducting effective reviews.
- **Importance of Fault Classifications:**
  - Fault classifications provided by the IEEE standard and review checklists aid in systematically identifying, categorizing, and addressing different types of faults.
  - They serve as valuable tools for improving software quality, enhancing development processes, and mitigating risks.

Levels of Testing

- **Levels of Abstraction in Testing:**
  - Levels of testing correspond to levels of abstraction found in the waterfall model of the software development life cycle.
  - The waterfall model, despite its drawbacks, serves as a useful framework for identifying distinct levels of testing and clarifying their objectives.
- **The V-Model:**
  - A diagrammatic variation of the waterfall model, known as the V-Model, emphasizes the correspondence between testing and design levels.
  - The V-Model illustrates how different levels of testing align with different stages of software design, including specification, preliminary design, and detailed design.

Figure :1.8



Levels of abstraction and testing in the Waterfall Model.

- **Correspondence between Testing Levels and Design Levels:**
  - In terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) directly correspond to three levels of testing—system, integration, and unit testing.
- **Relationship with Specification-Based and Code-Based Testing:**
  - Most practitioners agree that code-based testing is most appropriate at the unit level, while specification-based testing is most appropriate at the system level.
  - This relationship is partly influenced by the information produced during the requirements specification, preliminary design, and detailed design phases.
- **Development of Testing Structures:**
  - Constructs defined for code-based testing are typically suitable for the unit level.
  - Efforts are underway to develop similar constructs for integration and system levels of testing, both for traditional and object-oriented software.

Examples

- Three examples to illustrate various unit Testing methods.
- *These examples raise most of the issues that testing craftsperson's will encounter at the unit level.*
- For the purpose of structural testing, pseudocode implementation of 3 unit-level eg. are given.
  - The triangle problem
  - NextDate
  - Commission problem

Generalized Psuedocode

- Pseudocode provides a *“language neutral”* way to express program source code.
- Pseudocode given here is based on visual basic.

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	' <text>
Data structure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Simple condition	<expression> <relational operator> <expressio
Compound condition	<simple condition> <logical connective> <Simple condition>
Sequence	statements in sequential order

Simple selection	If <condition> Then
	<then clause>
	EndIf

Selection	If <condition> Then <then clause> Else <else clause> EndIf
Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> ... Case n: <predicate> <Case clause> EndCase

Counter-controlled repetition	For <counter> = <start> To <end> <loop body> EndFor
Pretest repetition	Do While <condition> <loop body> EndWhile
Posttest repetition	Do <loop body> Until <condition>
Procedure definition (similarly for functions and o-o methods)	<procedure name> (Input: <variable list>; Output: <variable list>) <body> End <procedure name>
Interunit communication	Call <procedure name> (<variable list>; <variable list>)
Class/Object definition	<name> (<attribute list>; <method list>, <body> End <name>
Interunit communication	msg <destination object name>.<method name> (<variable list>)
Object creation	Instantiate <class name>.<object name> (attribute values)

Table 2.1 Generalized Pseudocode *(Continued)*

Language Element	*Generalized Pseudocode Construct*
Object destruction	Delete <class name>.<object name>
Program	Program <program name> <unit list> End<program name>

Triangle Problem

- Problem statement
- Simple version: The triangle program accepts 3 integers a, b, c as input to be sides of a triangle
- o/p is type of triangle determined by 3 sides
- Equilateral, Isosceles, Scalene, Not a triangle.

Improved version

Sides of triangle integer a, b, c must satisfy the following conditions

c1. $1 \leq a \leq 200$
c2. $1 \leq b \leq 200$
c3. $1 \leq c \leq 200$
c4. $a < b + c$
c5. $b < a + c$
c6. $c < a + b$

One of the 4 mutually exclusive output 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 fails, the program output is NotATriangle.

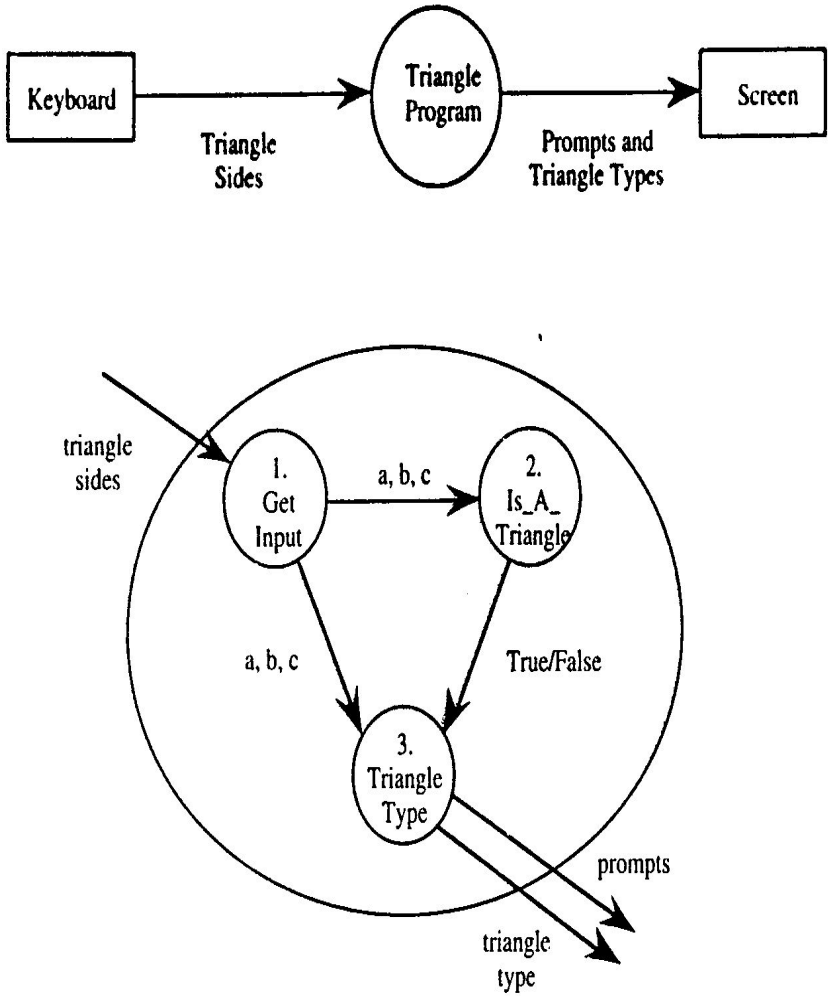


Figure 2.2 Dataflow diagram for a structured triangle program implementation.

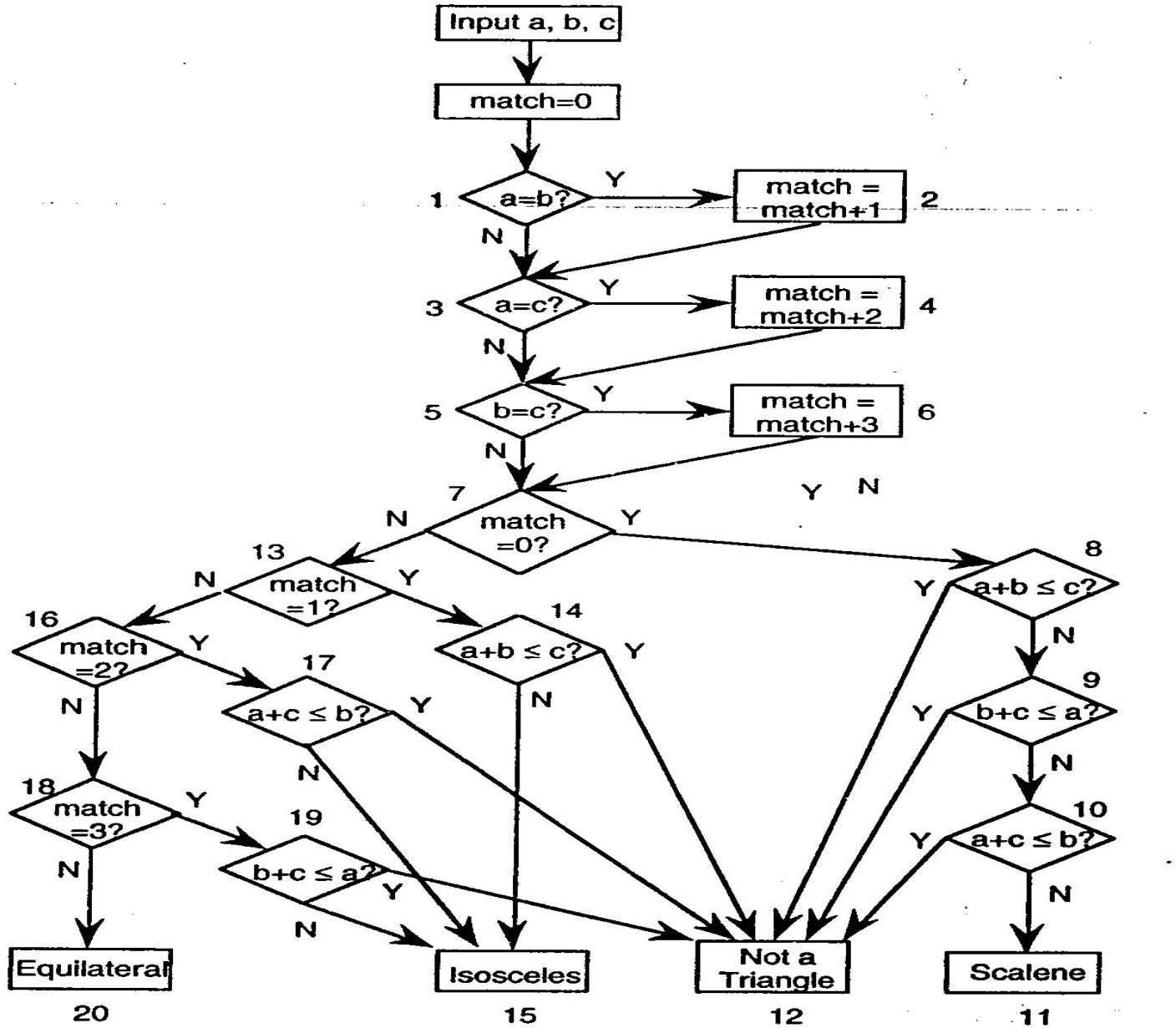
```
Program triangle2 'Structured programming version of simpler specification
'
Dim a,b,c As Integer
Dim IsATriangle As Boolean
'
'Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
'
```

```
'Step 3:  Determine Triangle Type
If IsATriangle
    Then    If (a = b) AND (b = c)
                Then Output ("Equilateral")
                Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                            Then    Output ("Scalene")
                            Else    Output ("Isosceles")
                        EndIf
            EndIf
    Else    Output("Not a Triangle")
EndIf
'
End triangle2
```

```
Program triangle3 'Structured programming version of improved specification
'
Dim a,b,c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'
'Step 1: Get Input
Do
    Output("Enter 3 integers which are sides of a triangle")
    Input(a,b,c)
    c1 = (1 <= a) AND (a <= 200)
    c2 = (1 <= b) AND (b <= 200)
    c3 = (1 <= c) AND (c <= 200)
    If NOT(c1)
        Then   Output("Value of a is not in the range of permitted values")
    EndIf
    If NOT(c2)
        Then   Output("Value of b is not in the range of permitted values")
    EndIf
    If NOT(c3)
        Then   Output("Value of c is not in the range of permitted values")
    EndIf
Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
```

```
'Step 2:  Is A Triangle?
If (a < (b + c)) AND (b < (a + c)) AND (c < (a + b))
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
'
'Step 3:  Determine Triangle Type
If IsATriangle
    Then    If (a = b) AND (b = c)
                Then Output ("Equilateral")
                Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                            Then    Output ("Scalene")
                            Else    Output ("Isosceles")
                        EndIf
            EndIf
    Else    Output("Not a Triangle")
EndIf
'
End triangle3
```

Traditional Implementation



```
Program triangle1 'Fortran-like version
'
Dim a,b,c,match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
match = 0
If a = b                                                          '(1)
    Then  match = match + 1                                       '(2)
EndIf
If a = c                                                          '(3)
    Then  match = match + 2                                       '(4)
EndIf
If b = c                                                          '(5)
    Then  match = match + 3                                       '(6)
EndIf
If match = 0                                                      '(7)
    Then   If (a+b)<=c                                            '(8)
               Then   Output("NotATriangle")                      '(12.1)
               Else   If (b+c)<=a                                 '(9)
                          Then   Output("NotATriangle")           '(12.2)
                          Else   If (a+c)<=b                      '(10)
                                     Then   Output("NotATriangle") '(12.3)
                                     Else   Output ("Scalene")    '(11)
                                 EndIf
                      EndIf
          EndIf
```

```
Else    If match=1                                                    '(13)
            Then    If (a+c)<=b                                       '(14)
                        Then    Output("NotATriangle")                '(12.4)
                        Else    Output ("Isosceles")                  '(15.1)
                    EndIf
            Else    If match=2                                        '(16)
                        Then    If (a+c)<=b
                                    Then    Output("NotATriangle")    '(12.5)
                                    Else    Output ("Isosceles")      '(15.2)
                                EndIf
                        Else    If match=3                            '(18)
                                    Then    If (b+c)<=a               '(19)
                                                Then    Output("NotATriangle")  '(12.6)
                                                Else    Output ("Isosceles")    '(15.3)
                                            EndIf
                                    Else    Output ("Equilateral")    '(20)
                                EndIf
                    EndIf
        EndIf
EndIf
'
End Triangle1
```

The NextDate Function

- Illustrate complexity
- Logical relationship among the i/p variables

Problem statement:

- NextDate is a function of 3 variables Month, Day, Year.
- It returns the date of the day after the i/p date.
- condition

c1. $1 \leq month \leq 12$
c2. $1 \leq day \leq 31$
c3. $1812 \leq year \leq 2012$

Problem statement

- Responses for invalid values of i/p values for day, month, year.
- Responses for invalid combination of i/p june 31 any year.
- If any of the conditions C1, C2, or C3 fails
  - Corresponding variables has out-of-range values.
  - Eg. “Value of month not in range 1…12”
- If invalid day-month- year combination exist NextDate collapses these into one message

  “Invalid input date”

Discussion

- Two source of complexity
  - Complexity of input domain
  - Rule that determine when a year is leap year.
- A year is 365.2422 days long
- Leap years are used for the “extra day” problem.
- According to Gregorian calendar
  - A year is a leap year if it is divisible by 4, unless it is a century year.
  - Century years are leap years only if they are multiples of 400
  - So 1992, 1996, 2000 are leap years… 1900 is not

Implementation

```
Program NextDate1          'Simple version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
'

Output ("Enter today's date in the form MM DD YYYY")
Input (month,day,year)
Case month Of
```

```
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
    If day < 30
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
Case 3: month Is 12: 'December
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = 1
            If year = 2012
                Then Output ("2012 is over")
                Else tomorrow.year = year + 1
    EndIf
```

```
Case 4: month is 2: 'February
    If day < 28
        Then tomorrowDay = day + 1
        Else
            If day = 28
                Then
                    If ((year is a leap year)
                        Then tomorrowDay = 29 'leap year
                        Else          'not a leap year
                            tomorrowDay = 1
                            tomorrowMonth = 3
                    EndIf
                Else    If day = 29
                            Then tomorrowDay = 1
                                 tomorrowMonth = 3
                            Else    Output("Cannot have Feb.", day)
                        EndIf
            EndIf
    EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate
```

Improved Version

```
Program NextDate2          Improved version
'

Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
'

Do
   Output ("Enter today's date in the form MM DD YYYY")
```

```
    Input (month,day,year)
    c1 = (1 <= day) AND (day <= 31)
    c2 = (1 <= month) AND (month <= 12)
    c3 = (1812 <= year) AND (year <= 2012)
    If NOT(c1)
        Then    Output("Value of day not in the range 1..31")
    EndIf
    If NOT(c2)
        Then    Output("Value of month not in the range 1..12")
    EndIf
    If NOT(c3)
        Then    Output("Value of year not in the range 1812..2012")
    EndIf
Until c1 AND c2 AND c2

Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
```

```
Case 2: month Is 4,6,9, Or 11 '30 day months
    If day < 30
        Then tomorrowDay = day + 1
        Else
            If day = 30
                Then    tomorrowDay = 1
                        tomorrowMonth = month + 1
                Else    Output("Invalid Input Date")
            EndIf
    EndIf
Case 3: month Is 12: 'December
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = 1
            If year = 2012
                Then Output ("Invalid Input Date")
                Else tomorrow.year = year + 1
    EndIf
```

```
Case 4: month is 2: 'February
    If day < 28
        Then tomorrowDay = day + 1
        Else
            If day = 28
                Then
                    If (year is a leap year)
                        Then tomorrowDay = 29 'leap day
                        Else          'not a leap year
                            tomorrowDay = 1
                            tomorrowMonth = 3
                    EndIf
                Else
                    If day = 29
                        Then
                            If (year is a leap year)
```

```
                                    Then    tomorrowDay = 1
                                            tomorrowMonth = 3
                                    Else
                                        If day > 29
                                            Then    Output("Invalid Input Date")
                                        EndIf
                                    EndIf
                                EndIf
                        EndIf
                EndIf
        EndCase
        Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
        '
        End NextDate2
```

The commission Problem

- It contains a mix of computation & decision making.
- A rifle salesperson in the former Arizona territory sold rifle lock's, stocks, & barrel's made of a gunsmith in Missouri.
- Locks cost $45, stocks cost $30, Barrel Cost $ 25.
- Sales person has to sell at least 1 complete rifle per month
- Production limitation such that 1 sales man can sell 70 locks, 80 stocks, 90 barrels per month.

- After each town visit salesperson update sale of no of locks, stocks, barrels through a telegram to gunsmith
- At the end of month salesperson sent a shot telegram showing -1 locks sold.
- Gunman knew sales for month are over & compute the commission of sales person
  - 10% on sales up to $1000
  - 15% on the next $800
  - 20% on any sales in excess of $1800

  The commission program produces a monthly sales report that gave total no. of locks, barrels, stocks sold. Sales persons total dollar sale & commission.

Discussion

- This problem separates into 3 distinct pieces
- The input data portion(data validation) ignore here

Sales calculation Commission calculation problem.

Implementation

```
Program Commission (INPUT,OUTPUT)
'
Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks,totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL
'
lockPrice   = 45.0
stockPrice  = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0
'
Input(locks)
While NOT(locks = -1)      'Input device uses -1 to indicate end of data
    Input(stocks, barrels)
    totalLocks = totalLocks + locks
    totalStocks = totalStocks + stocks
    totalBarrels = totalBarrels + barrels
    Input(locks)
EndWhile
'
```

```
Output("Locks sold: ", totalLocks)
Output("Stocks sold: ", totalStocks)
Output("Barrels sold: ", totalBarrels)
'
lockSales = lockPrice*totalLocks
stockSales = stockPrice*totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output("Total sales: ", sales)
'
If (sales > 1800.0)
    Then
        commission = 0.10 * 1000.0
        commission = commission + 0.15 * 800.0
        commission = commission + 0.20*(sales-1800.0)
    Else If (sales > 1000.0)
                Then
                    commission = 0.10 * 1000.0
                    commission = commission + 0.15*(sales-1000.0)
                Else commission = 0.10 * sales
            EndIf
EndIf
Output("Commission is $",commission)
'
End Commission
```

The SATM System

- To better discuss the issues of integration & system testing

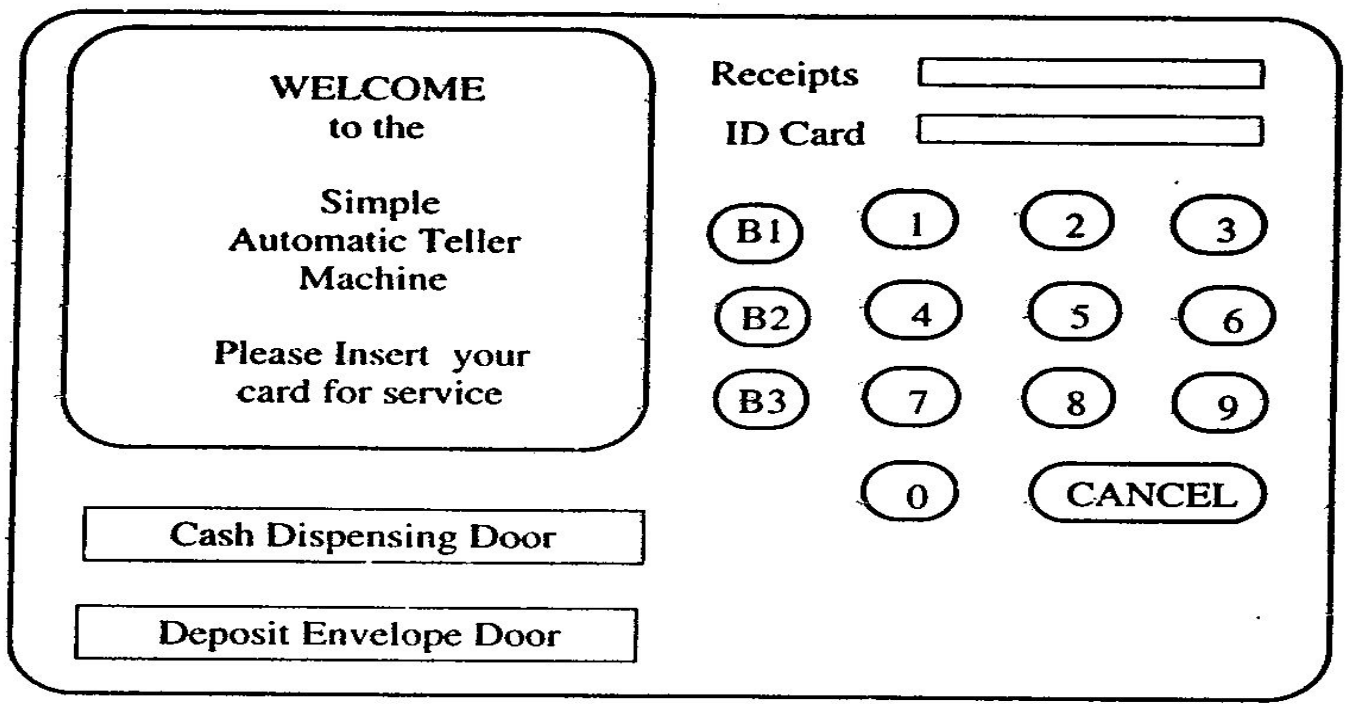


Figure 2.3 The SATM terminal.

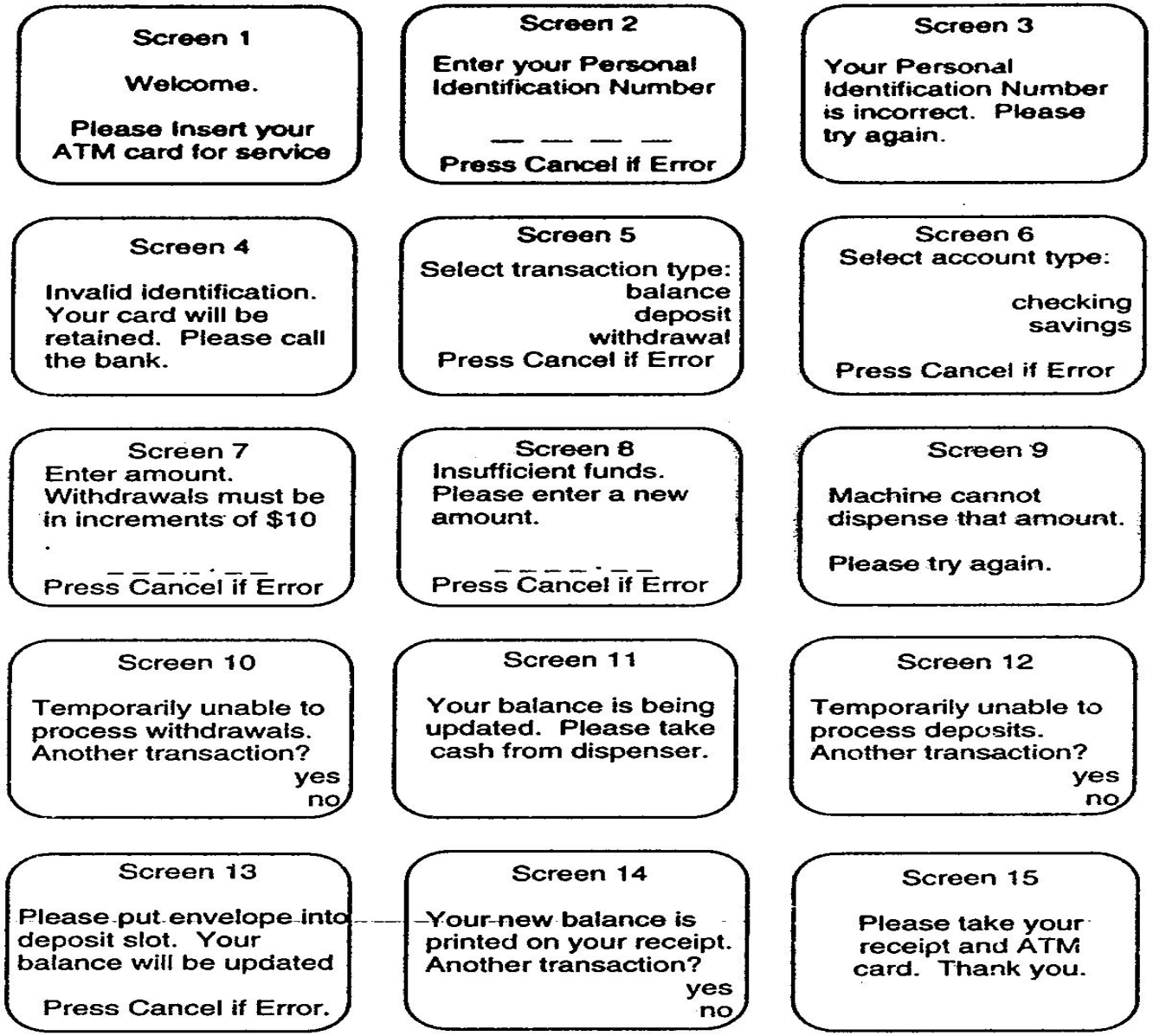


Figure 2.4 SATM screens.

The currency converter

- ***Another event driven program that emphasizes code associated with a GUI***
- ***A sample GUI built with visual basic is shown.***

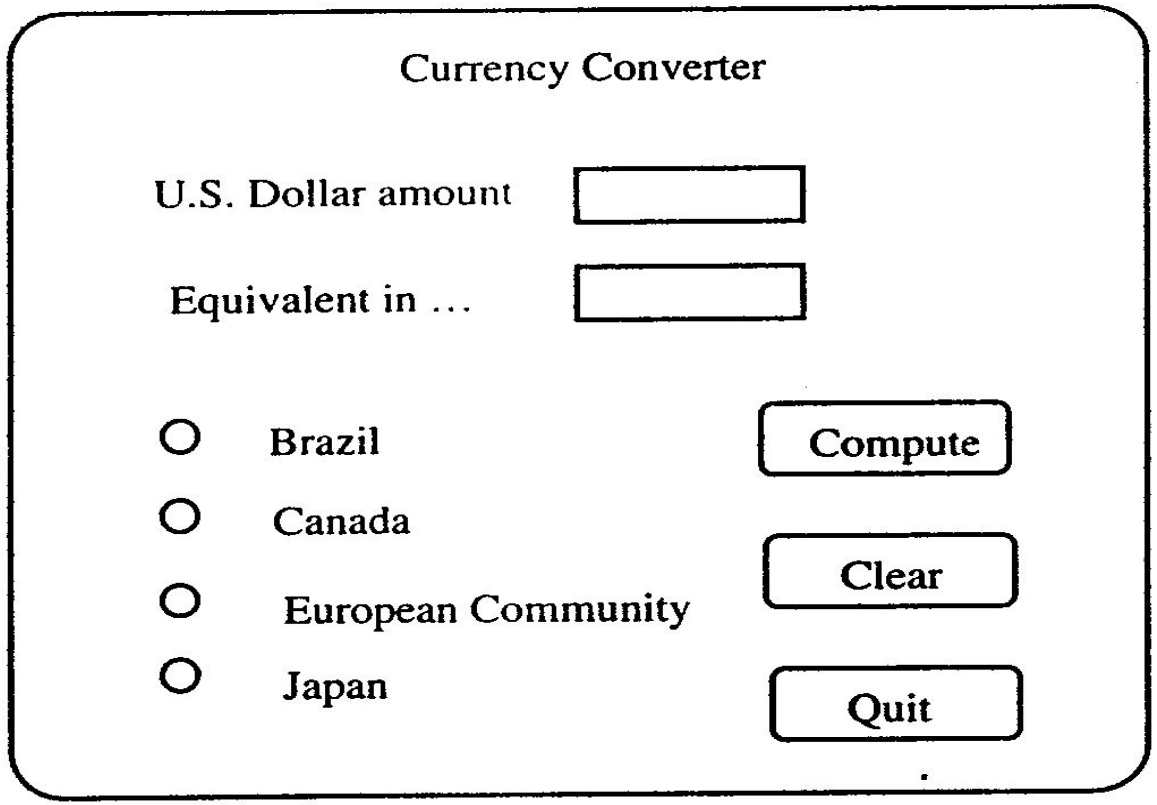


Figure 2.5 Currency converter GUI.

Saturn Windshield Wiper Controller

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper	0	4	6	12	30	60


MIST
OFF
INT
LO
HI
FRONT
INT
OFF
INT
ON
REAR
Byerly Nissan
(502) 448-8222