Maharaja Education Trust (R), Mysuru

# Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

**Approved by AICTE, New Delhi,**
**Affiliated to VTU, Belagavi & Recognized by Government of Karnataka**

# Lecture Notes on

# SOFTWARE TESTING (17IS63)

**Prepared by**

**Prof. Ramya S & Prof. SmithaShree K P**

**Department of Information Science and Engineering**

## Vision

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

## Mission

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.
- To advance extensive research in science, engineering and management disciplines.
- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.
- To instill the need to uphold ethics in every aspect.
- To mould holistic individuals capable of contributing to the advancement of the society.

## VISION OF THE DEPARTMENT

To be recognized as the best centre for technical education and research in the field of information science and engineering.

## MISSION OF THE DEPARTMENT

- To facilitate adequate transformation in students through a proficient teaching learning process with the guidance of mentors and all-inclusive professional activities.
- To infuse students with professional, ethical and leadership attributes through industry collaboration and alumni affiliation.
- To enhance research and entrepreneurship in associated domains and to facilitate real time problem solving.
-

## PROGRAM EDUCATIONAL OBJECTIVES:

- Proficiency in being an IT professional, capable of providing genuine solutions to information science problems.
- Capable of using basic concepts and skills of science and IT disciplines to pursue greater competencies through higher education.
- Exhibit relevant professional skills and learned involvement to match the requirements of technological trends.

## PROGRAM SPECIFIC OUTCOME:

Student will be able to

- **PSO1:** Apply the principles of theoretical foundations, data Organizations, networking concepts and data analytical methods in the evolving technologies.
- **PSO2:**Analyse proficient algorithms to develop software and hardware competence in both professional and industrial areas

# Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

# Course Overview

**SUBJECT: DOT NET FRAMEWORK FOR APPLICATION DEVELOPMENT**

**SUBJECT CODE: 17CS564**

This course provides an introduction to the process of validating and verifying the software during the life cycle of the software. Software testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools. Software testing techniques will be thought.

By this course student will learn verifying and validating softwares by using different software testing techniques. Student will be able to analyze the problem and apply the proper testing techniques.

At the end of the course students will be able to apply the different testing techniques to create the real time applications and also plan and monitor the process and documenting analysis and test,design specification ,test and analysis report.

# Course Objectives

1. Explain the need for planning and monitoring a process
2. Differentiate the various testing techniques
3. Analyze the problem and derive suitable test cases
4. Apply suitable technique for designing of flow graph

# Course Outcomes

| CO's | DESCRIPTION OF THE OUTCOMES |
|---|---|
| 17IS63.1 | Apply the basic concepts of testing to derive different test cases. |
| 17IS63.2 | Apply the different testing techniques to validate the software product. |
| 17IS63.3 | Analyze the testing techniques and tools to classifying the problems into suitable testing model. |
| 17IS63.4 | Analyze the different way of creating the documentation for software artifact. |

# Maharaja Institute of Technology Mysore

## Department of Information Science and Engineering

## Syllabus

**SUBJECT: SOFTWARE TESTING**
**SUBJECT CODE: 17IS63**

| Topics Covered as per Syllabus | Teaching Hours |
|---|---|
| **MODULE-1:** | |
| Basics of Software Testing: Basic definitions, Software Quality , Requirements, Behaviour and Correctness, Correctness versus Reliability, Testing and Debugging, Test cases, Insights from a Venn diagram, Identifying test cases, Test-generation Strategies, Test Metrics, Error and fault taxonomies , Levels of testing, Testing and Verification, Static Testing. Problem Statements: Generalized pseudocode, the triangle problem, the NextDate function, the commission problem, the SATM (Simple Automatic Teller Machine) problem, the currency converter, Saturn windshield wiper. | **10 Hours** |
| **MODULE-2:** | |
| Functional Testing: Boundary value analysis, Robustness testing, Worst-case testing, Robust Worst testing for triangle problem, Nextdate problem and commission problem, Equivalence classes, Equivalence test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations, Decision tables, Test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations. Fault Based Testing: Overview, Assumptions in fault based testing, Mutation analysis, Fault-based adequacy criteria, Variations on mutation analysis. | **10 Hours** |
| **MODULE -3:** | |
| Structural Testing: Overview, Statement testing, Branch testing, Condition testing , Path testing: DD paths, Test coverage metrics, Basis path testing, guidelines and observations, Data –Flow testing: Definition-Use testing, Slicebased testing, Guidelines and observations. Test Execution: Overview of test execution, from test case specification to test cases, Scaffolding, Generic versus specific scaffolding, Test oracles, Self-checks as oracles, Capture and replay. | **10 Hours** |
| **MODULE-4:** | |
| Process Framework :Basic principles: Sensitivity, redundancy, restriction, partition, visibility, Feedback, the quality process, Planning and monitoring, Quality goals, Dependability properties ,Analysis Testing, Improving the process, Organizational factors. Planning and Monitoring the Process: Quality and process, Test and analysis strategies and plans, Risk planning, monitoring the process, Improving the process, the quality team Documenting Analysis and Test: Organizing documents, Test strategy document, Analysis and test plan, Test design specifications documents, Test and analysis reports. | **10 Hours** |
| **MODULE-5:** | |
| Integration and Component-Based Software Testing: Overview, Integration testing strategies, Testing components and assemblies. System, Acceptance and Regression Testing: Overview, System testing, Acceptance testing, Usability, Regression testing, Regression test selection techniques, Test case prioritization and selective execution. Levels of Testing, Integration Testing: Traditional view of testing levels, Alternative life-cycle models, The SATM system, Separating integration and system testing, A closer look at the SATM system, Decomposition-based, call graph-based, Path-based integrations. | **10 Hours** |
| **List of Text Books** | |
| 1.Paul C. Jorgensen: Software Testing, A Craftsman's Approach, 3rd Edition, Auerbach Publications, 2008. (Listed topics only from Chapters 1, 2, 5, 6, 7, 9, 10, 12, 13)<br>2.Mauro Pezze, Michal Young: Software Testing and Analysis – Process, Principles and Techniques, Wiley India, 2009. (Listed topics only from Chapters 3, 4, 16, 17, 20,21, 22,24)<br>1. 3. Aditya P Mathur: Foundations of Software Testing, Pearson Education, 2008.( Listed topics only from Section 1.2 , 1.3, 1.4 ,1.5, 1.8,1.12,6. 2.1,6. 2.4 ) | |
| **List of Reference Books** | |

1. Software testing Principles and Practices – Gopalaswamy Ramesh, Srinivasan Desikan, 2 nd Edition, Pearson, 2007.

**List of URLs, Text Books, Notes, Multimedia Content, etc**

1. https://www.tutorialspoint.com/software_testing/index.htm
2. https://nptel.ac.in/courses/106/105/106105150/
3. https://www.pdfdrive.com/software-testing-books.html

# Index

**SUBJECT: SOFTWARE TESTING**
**SUBJECT CODE: 17IS63**


| Module-1 | Pg no |
|---|---|
| **Basics of Software Testing:** Basic definitions, Software Quality | *1,27-29* |
| Requirements Behavior and Correctness, Correctness versus Reliability | *29-31* |
| Testing and Debugging, Test cases, Insights from a Venn diagram | *2-3,31-37* |
| Identifying test cases, Test-generation Strategies, | *4* |
| Test Metrics, Error and fault taxonomies , Levels of testing | *6-8* |
| **Problem Statements:** Generalized pseudo code, the triangle problem | *9-16* |
| The Next Date function, the commission problem, | *16-22* |
| SATM (Simple Automatic Teller Machine) problem | *22-26* |
| The currency converter, Saturn windshield wiper | *26* |
| Module-2 | Pg no |
| **Functional Testing:** Boundary value analysis, Robustness testing | *39-43* |
| Worst-case testing, Robust Worst testing for triangle problem | *43-47* |
| Next date problem and commission problem | *47-55* |
| Equivalence classes, Equivalence test cases for the triangle problem | *56-61* |
| Next Date function, and the commission problem | *62* |
| Guidelines and observations, Decision tables, Test cases for the triangle problem, | *63* |
| Next Date function, and the commission problem, Guidelines and observations | *64-69* |
| **Fault Based Testing:** Overview, Assumptions in fault based testing, | *70-71* |
| Mutation analysis Fault-based adequacy criteria | *72-77* |
| Variations on mutation analysis. | *78-79* |
| **Module 3** | **Pg no** |
| **Structural Testing:** Overview, Statement testing, Branch testing Condition | *86-87* |
| Path testing: DD paths, | *80-84* |
| Test coverage metrics, Basis path testing, | *84-100* |
| Guidelines and observations | *100-101* |
| Data –Flow testing, Definition-Use testing, Slice based testing | *102-113* |

| | |
|---|---|
| **Test Execution:** Overview of test execution, | ***114*** |
| from test case specification to test cases, Scaffolding | ***115*** |
| Test oracles, Self-checks as oracles | ***116*** |
| Capture and replay | ***117*** |
| Module 4 | **Pg no** |
| **Process Framework :**Basic principles: Sensitivity, redundancy, restriction partition, visibility, Feedback, the quality process, Planning and monitoring. | ***118*** |
| **Planning and Monitoring the Process:** Quality and process, Test and analysis strategies and plans | ***118-124*** |
| Risk planning, monitoring the process, Improving the process, the quality team | ***125-135*** |
| **Documenting Analysis and Test:** Organizing documents | ***136*** |
| Test strategy document | ***138*** |
| Analysis and test plan | ***138*** |
| Test design specifications documents | ***139-140*** |
| Test and analysis reports | ***141-142*** |
| Module 5 | **Pg no** |
| **Integration and Component-Based Software Testing:** Overview, Integration testing strategies | ***143-146*** |
| Testing components and assemblies. System, Acceptance | ***146-149*** |
| Regression Testing: Overview, System testing, Acceptance testing, Usability | ***150-153*** |
| **Levels of Testing, Integration Testing:** Traditional view of testing levels,, Alternative life-cycle models | ***143*** |
| Separating integration and system testing, | ***150*** |
| Regression testing, Regression test selection techniques, | ***154-157*** |
| Test case prioritization and selective execution | ***158*** |

# Software Testing

# Module 1

## Basic Definitions

**Error**—People make errors. A good synonym is *mistake*. When people make mistakes while coding, we call these mistakes *bugs*.

**Fault**—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, another name for fault is defects.**Fault of commission** occurs when we enter something into a representation that is incorrect. An error of omission results in a fault in which something is missing that should be present in the representation.**Faults of omission** occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

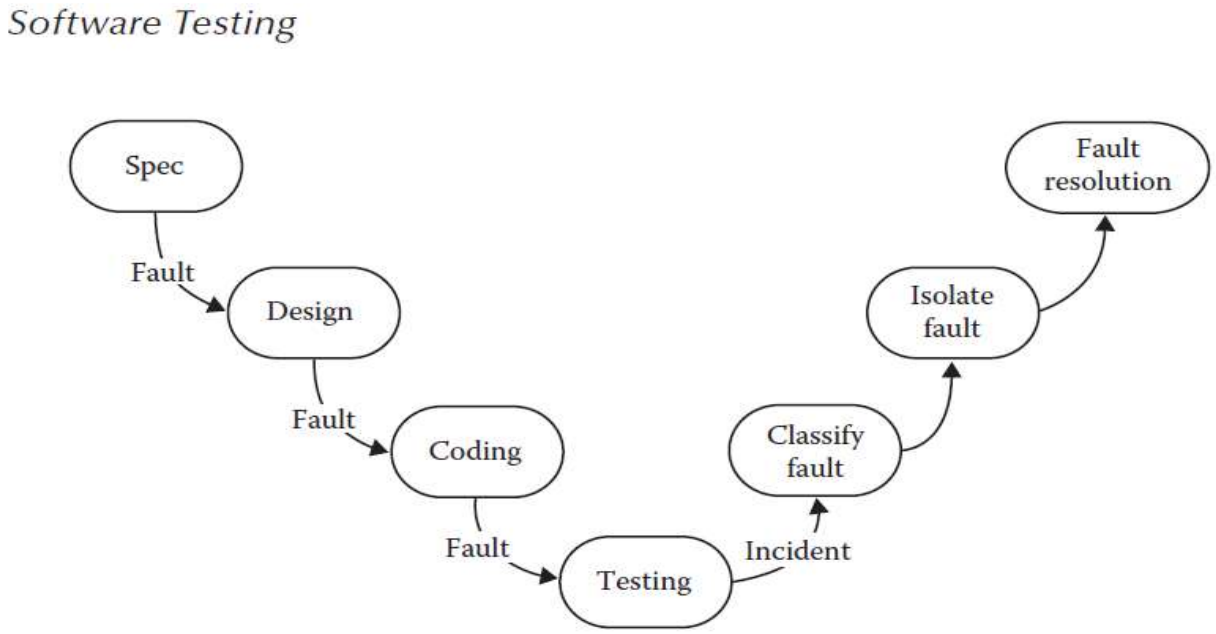
**Failure**—A failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition relates failures only to faults of commission.

**Incident:** An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

**Test**—Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.

**Test case—**A test case has an identity and is associated with a program behavior. It also has a set of inputs and expected outputs.

## Testing life cycle




The development phases, three opportunities arise for errors to be made, resulting in faults that may propagate through the remainder of the development process. The fault resolution step is another opportunity for errors (and new faults). When a fix causes formerly correct software to misbehave, the fix is deficient. We will revisit this when we discuss regression testing. From this sequence of terms, we see that test cases occupy a central position in testing. The process of testing can be subdivided into separate steps: test planning, test case development, running test cases, and evaluating test results.

## Test case

The essence of software testing is to determine a set of test cases for the item to be tested. A complete test case will contain a test case identifier, a brief statement of purpose (e.g., a business rule), a description of preconditions, the actual test case inputs, the expected outputs, a description of expected postconditions, and an execution history. The execution history is primarily for test management use—it may contain the date when the test was run, the person who ran it, the version on which it was run, and the pass/fail result.

Test case execution entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, comparing these with the expected outputs, and then ensuring that the expected postconditions exist to determine whether the test passed. From all of this, it becomes clear that test cases are valuable—at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

## Insights from a Venn Diagram

Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code-based view common to software (and system) developers.In this section, we develop a simple Venn diagram that clarifies several questions about testing.


**Figure 1.2 Specified and implemented program behaviors.**

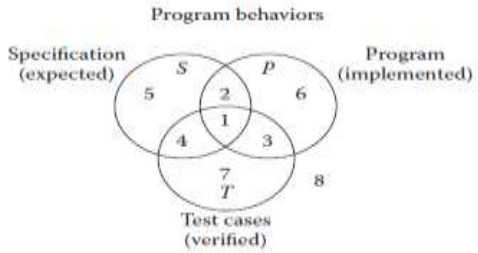



**Figure 1.3 Specified, implemented, and tested behaviors.**

Consider a universe of program behaviors.Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. Figure 1.2 shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S and all those behaviors actually programmed are in P. What if certain specified behaviors have not been programmed? In our earlier terminology, these are **faults of omission.** Similarly, what if certain programmed (implemented) behaviors have not

been specified? These correspond to **faults of commission** and to errors that occurred after the specification was complete.

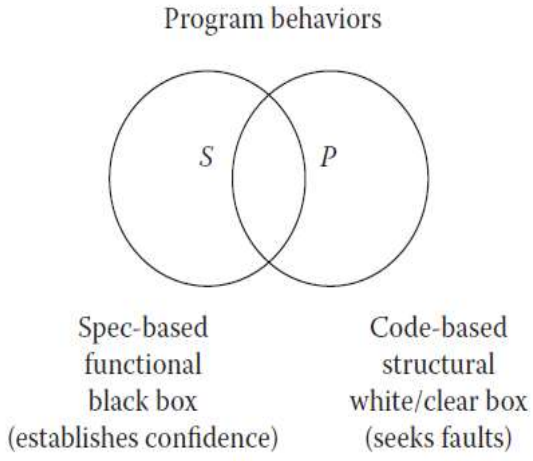
**The intersection of S and P (the football-shaped region) is the "correct" portion, that is, behaviors that are both specified and implemented.**

The new circle in Figure 1.3 is for test cases. Now, consider the relationships among sets S, P, and T.

- There may be **specified behaviors** that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).
- There may be **programmed behaviors** that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7).

## Identifying Test Cases

Two fundamental approaches are used to identify test cases; traditionally, these have been calledfunctional and structural testing. Specification-based and code-based are more descriptive names,and they will be used here.



***Specification-Based Testing***

The reason that specification-based testing was originally called "**functional testing**" is that any program can be considered to be a function that maps values from its input domain to values in its output range. It also called **Black-Box** testing in which the content (implementation) of the black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs.

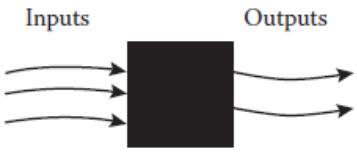



**Figure 1.4 Engineer's black box.**

**Advantages:**

(1) They are independent of how the software is implemented, so if the implementation changes, the test cases are still useful;

(2) Test case development can occur in parallel with the implementation, thereby reducing the overall project development interval.

**Disadvantage:**

(1) Significant redundancies may exist among test cases.

(2) Compounded by the possibility of gaps of untested software.

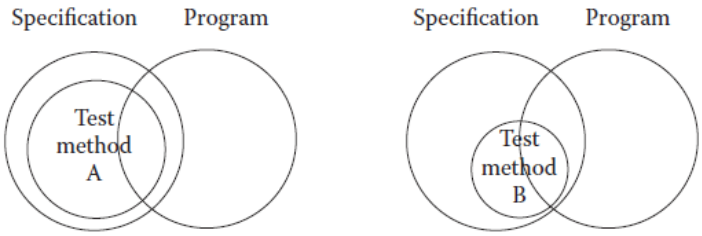



**Figure 1.5 Comparing specification-based test case identification methods.**

Figure 1.5 shows the results of test cases identified by two specification-based methods. Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because specification based

methods are based on the specified behavior, **it is hard to imagine these methods identifying behaviors that are not specified.**

***Code-Based Testing***

Code-based testing is the other fundamental approach to test case identification. It is also called **White-Box testing,** it main focus on implementation and also ability to "see inside" the black box allows the tester to identify test cases on the basis of how the function is actually implemented.

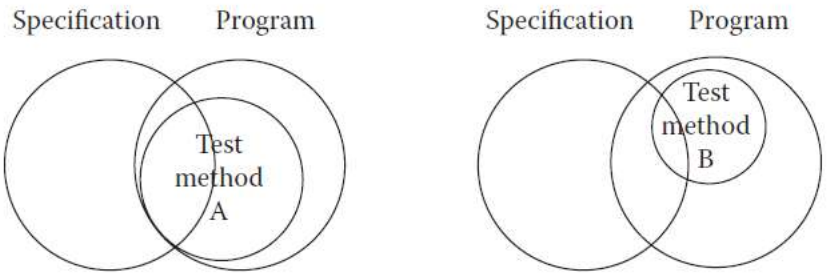



**Figure 1.6 Comparing code-based test case identification methods.**

Figure 1.6 shows the results of test cases identified by two code-based methods. Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of programmed behavior. Because code-based methods are based on the program, it is hard to imagine these methods identifying behaviors that are not programmed.

## Fault Taxonomies

Our definitions of error and fault hinge on the distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance (SQA) meet is that SQA typically tries to improve the product by improving the process.Some of the more useful anomalies are given in Tables 1.1 through 1.5; most of these are from the IEEE standard, as follows

**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 |

**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 ≤) |

**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 |

**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 |

## Levels of Testing

A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels.

- Specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing—system, integration, and unit testing.
- A practical relationship exists between levels of testing versus specification-based and codebasedtesting. Most practitioners agree that code-based testing is most appropriate at the unitlevel, whereas specification-based testing is most appropriate at the system level.
- The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing.

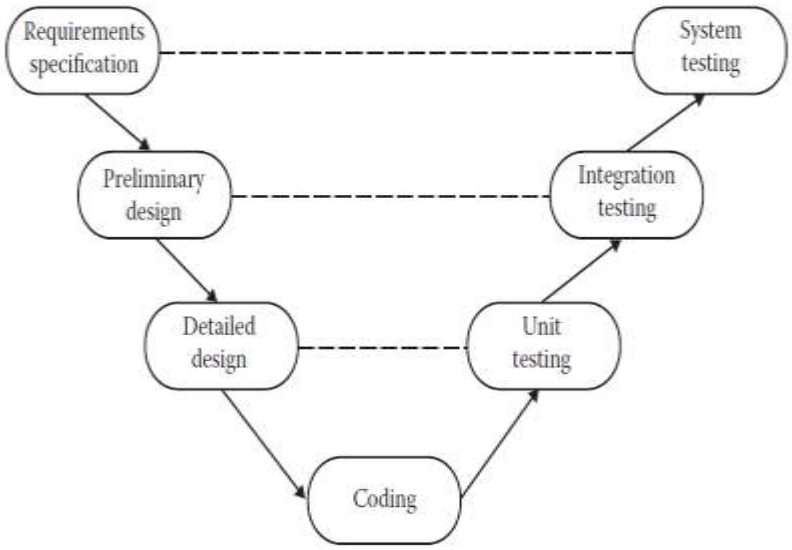



Figure 1.8 Levels of abstraction and testing in waterfall model.

**Examples**

## Generalized Pseudocode

Pseudocode provides a language-neutral way to express program source code. This version is loosely based on Visual Basic and has constructs at two levels: unit and program components. Units can be interpreted either as traditional components (procedures and functions) or as object-oriented components (classes and objects). We illustrate this by allowing natural language phrases in place of more formal, complex conditions ( Table 2.1).

**Table 2.1 Generalized Pseudocode**

| *Language Element* | *Generalized Pseudocode Construct* |
|---|---|
| Comment | ' <text> |
| Data structure declaration | Type <type name><br><list of field descriptions><br>End <type name> |
| Data declaration | Dim <variable> As <type> |
| Assignment statement | <variable> = <expression> |
| Input | Input (<variable list>) |
| Output | Output (<variable list>) |
| Condition | <expression> <relational operator> <expression> |
| Compound condition | <Condition> <logical connective><br><Condition> |
| Sequence | statements in sequential order |
| Simple selection | If <condition> Then<br><then clause><br>EndIf |
| Selection | If <condition><br>Then <then clause><br>Else <else clause><br>EndIf |
| Multiple selection | Case <variable> Of<br>Case 1: <predicate><br><Case clause><br>...<br>Case *n*: <predicate><br><Case clause><br>EndCase |
| Counter-controlled repetition | For <counter> = <start> To <end><br><loop body><br>EndFor |
| Pretest repetition | While <condition><br><loop body><br>EndWhile |

(continued)

**Table 2.1 Generalized Pseudocode (Continued)**

| *Language Element* | *Generalized Pseudocode Construct* |
|---|---|
| Posttest repetition | Do<br><loop body><br>Until <condition> |
| Procedure definition (similarly for functions and o–o methods) | <procedure name> (Input: <list of variables>;<br>Output: <list of variables>)<br><body><br>End <procedure name> |
| Interunit communication | Call <procedure name> (<list of variables>;<br><list of variables>) |
| Class/Object definition | <name> (<attribute list>; <method list>, <body>)<br>End <name> |
| Interunit communication | msg <destination object name>.<method name><br>(<list of variables>) |
| Object creation | Instantiate <class name>.<object name> (list of attribute values) |
| Object destruction | Delete <class name>.<object name> |
| Program | Program <program name><br><unit list><br>End<program name> |

## The Triangle Problem

**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 Not A Triangle.

**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:

c1. $1 \le a \le 200$

c2. $1 \le b \le 200$

c3. $1 \le c \le 200$

c4. $a < b + c$

c5. $b < a + c$

c6. $c < a + b$

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 Not A Triangle.

***Traditional Implementation***

The traditional implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. Figure 2.2 is a flowchart for the improved version. The variable "match" is used to record equality among pairs of the sides. A classic intricacy of the FORTRAN style is connected with the variable "match": notice that all three tests for the triangle inequality do not occur. If two sides are equal, say a and c, it is only necessary to compare a + c with b.

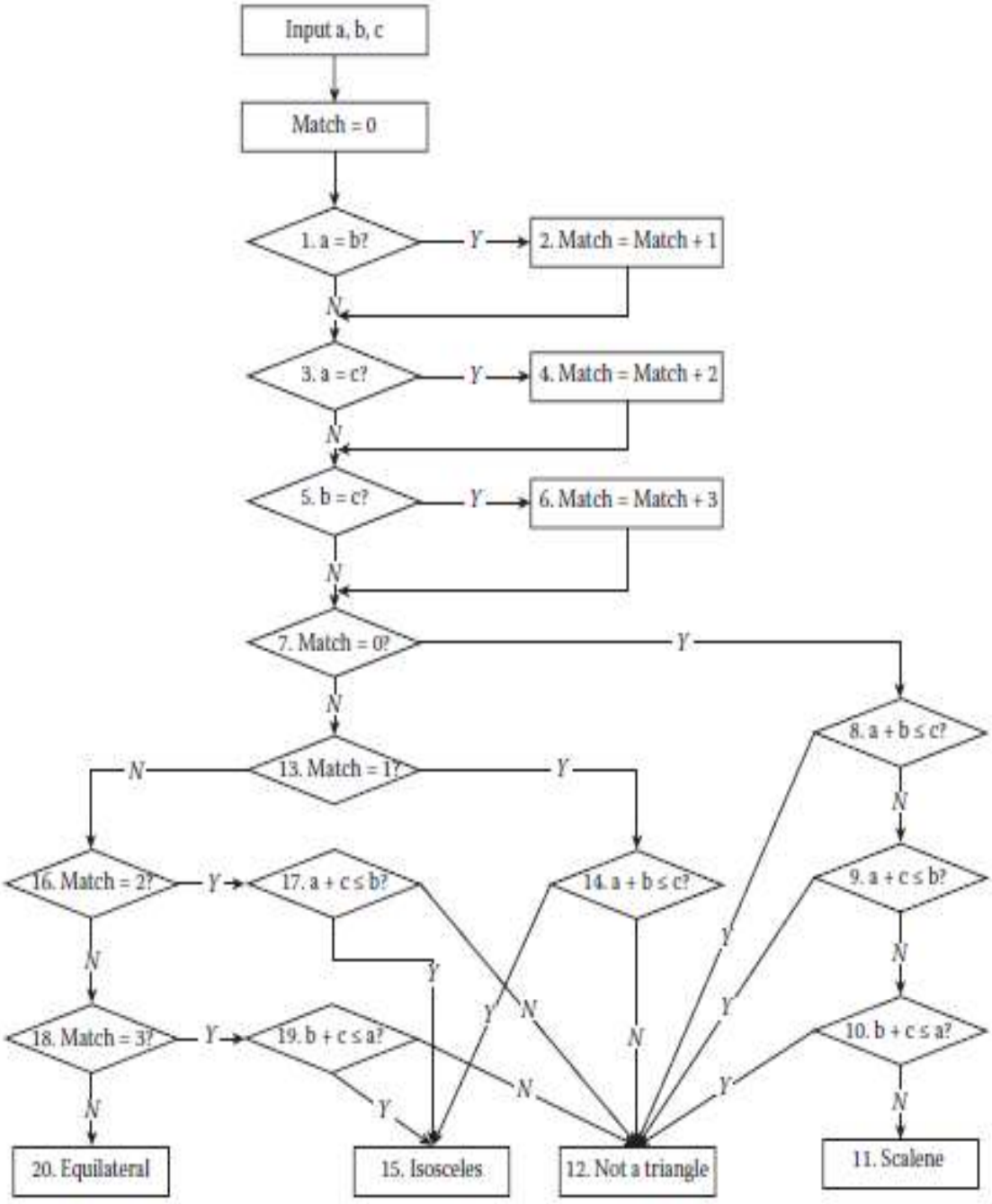



**Figure 2.1 Flowchart for traditional triangle program implementation.**

The pseudocode for this is given next.

Program triangle1 'Fortran-like version

Dim a, b, c, match As INTEGER

```
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
```

***Structured Implementations***

***Simple version***

Program triangle2 'Structured programming version of simpler specification

Dim a,b,c As Integer

Dim IsATriangleAs 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
```

**Improved Version**

Program triangle3'
Dim a, b, c As Integer
Dim c1, c2, c3, IsATriangleAs 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 ≤ 300)
    c2 = (1 ≤ b) AND (b ≤ 300)
    c3 = (1 ≤ c) AND (c ≤ 300)
    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)
         ThenOutput("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
```

## The NextDate Function

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions (the year range ending in 2012 is arbitrary, and is from the first edition):

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

If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, "Value of month not in the range 1...12." Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: "Invalid Input Date."

**Simple version**

Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is 365.2422 days long; therefore, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, a slight error would occur. Another complexity validating December month.

**Improved version**

If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, "Value of month not in the range 1...12." Because numerous invalid day–month

```
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 If ((year is a leap year)
                  Then tomorrowDay = 1
```

```
                        tomorrowMonth = 3
                    Else 'not a leap year
                      Output("Cannot have Feb.", day)
                  EndIf
            EndIf
      EndIf
 EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
End NextDate

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
 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

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to sell at least one lock, one stock, and one barrel (but not necessarily one complete rifle) per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. The gunsmith then knew the sales for the month were complete and computed the salesperson's commission as follows: 10% on sales up to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800.

```
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

The ATM described here is minimal, yet it contains an interesting variety of functionality and interactions that typify the client side of client–server systems. we need an example with larger scope(Figure2.3).

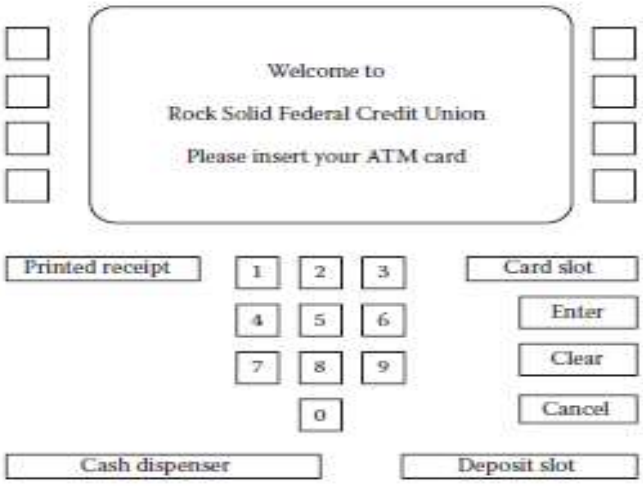



**Figure 2.3 SATM terminal.**

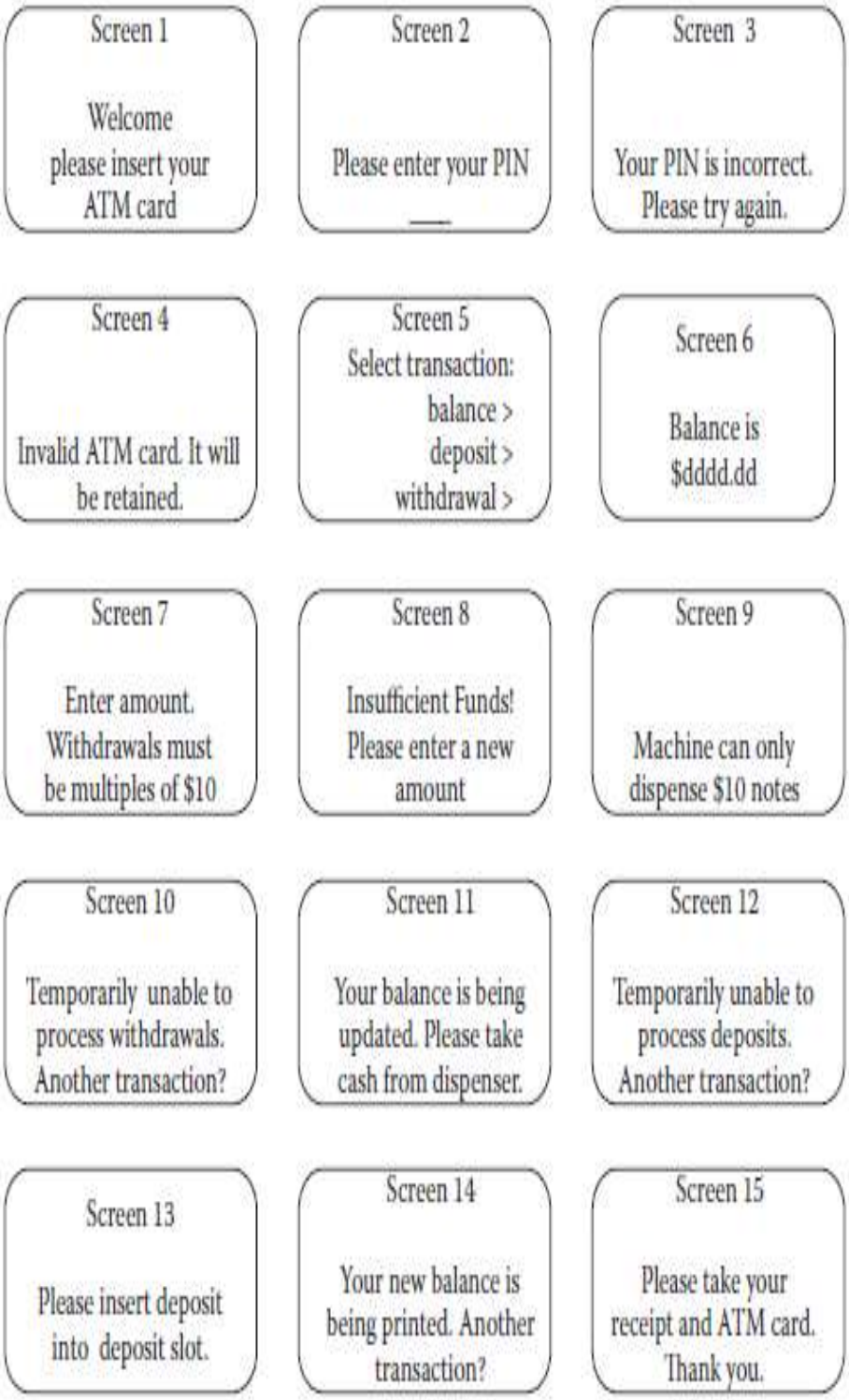



**Figure 2.4 SATM screens.**

The SATM system communicates with bank customers via the 15 screens shown in Figure 2.4. SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account.

- When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information.
- If the customer's PAN matches the information in the customeraccount file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept.
- At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.
- On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount.
- If a problem occurs with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13,accepts the deposit envelope, and processes the deposit. The system then displays screen 14.
- If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount.
- Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed.
- If the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14.

- When the "No" button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from the card slot, screen 1 is displayed. When the "Yes" button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

## The Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a GUI. A sample GUI is shown in Figure 2.5.

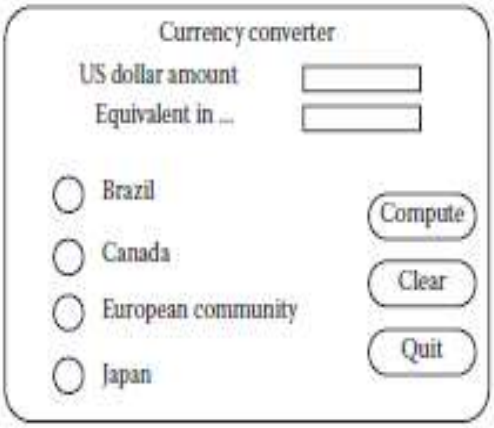



Figure 2.5 Currency converter graphical user interface.

The application converts US dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label;

Either before or after currency selection, the user inputs an amount in US dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, or the Quit button. Clicking on the Compute button results in the conversion of the US dollar amount to the equivalent amount in the selected currency.

## Saturn Windshield Wiper Controller

The windshield wiper on some Saturn automobiles is controlled by a lever with a dial. The lever has four positions: OFF, INT (for intermittent), LOW, and HIGH; and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

| 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 |

## SOFTWARE QUALITY

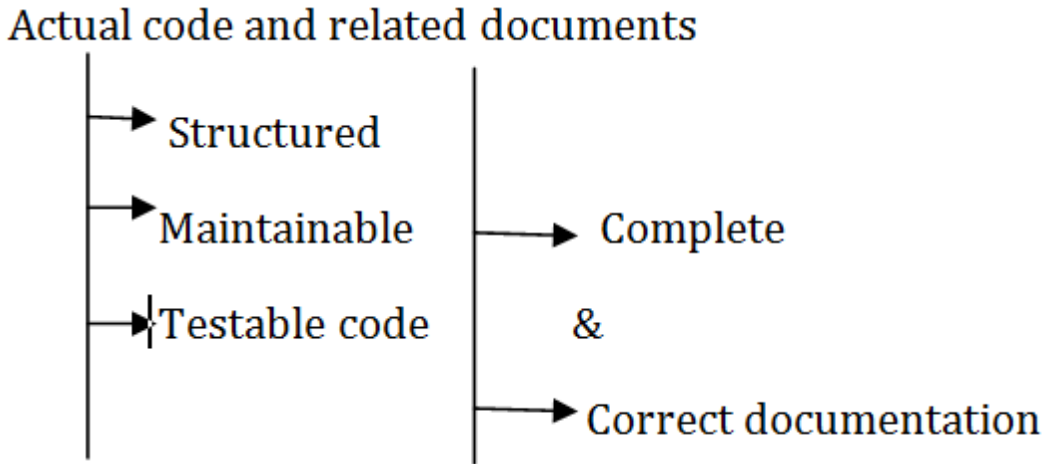
Software quality is a multidimensional quantity and is measurable.

**Quality Attributes**

These can be divided to static and dynamic quality attributes

**Static quality attributes**

It refers to the actual code and related documents.




Example: A poorly documented piece of code will be harder to understand and hence difficult to modify. A poorly structured code might be harder to modify and difficult to test.

**Dynamic quality Attributes:**

- Reliability
- Correctness
- Completeness
- Consistency
- Usability
- performance

**Reliability:**

It refers to the probability of failure free operation.

**Correctness:**

- Refers to the correct operation and is always with reference to some artefact.
- For a Tester, correctness is w.r.t to the requirements
- For a user correctness is w.r.t the user manual

**Completeness:**

- Refers to the availability of all the features listed in the requirements or in the user manual.
- An incomplete software is one that does not fuly implement all features required.

**Consistency:**

- Refers to adherence to a common set of conventions and assumptions.
- Ex: All buttons in the user interface might follow a common-color coding convention.

**Usability:**

- Refer to ease with which an application can be used. This is an area in itself and there exist techniques for usability testing.
- Psychology plays an important role in the design of techniques for usability testing.
- Usability testing is a testing done by its potential users.

- The development organization invites a selected set of potential users and asks them to test the product.
- Users in turn test for ease of use, functionality as expected, performance, safety and security.
- Users thus serve as an important source of tests that developers or testers within the organization might not have conceived.
- Usability testing is sometimes referred to as user-centric testing.

**Performance:**

Refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement.

**Reliability:**

Software reliability can vary from one operational profile to another. An implication is that one might say "this program is lousy" while another might sing praises for the same program. Software reliability is the probability of failure free operation of software in its intended environments.

## Requirements, Behaviour and Correctness:

During the development of the product, the requirement might have changed from what was stated originally. Regardless of any change, the expected behaviour of the product is determined by the tester's understanding of the requirements during testing.

**Requirement 1: It is required to write a program that inputs and outputs the maximum of these. Requirement 2: It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.**

- Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness requirements 1.

- The second requirement in (the above example is ambiguous. It is not clear from this requirement whether the input sequence is to be sorted in ascending or descending order.

## Correctness versus reliability:

- To establish correctness via testing would imply testing a program on all elements in the input domain, which is impossible to accomplish in most cases that are encountered in practice. Thus, correctness is established via **mathematical proofs of programs**.

- While correctness attempts to establish that the program is error-free, testing attempts to find if there are any errors in it. Thus, completeness of testing does not necessarily demonstrate that a program is error-free.

- Testing, debugging and the error-removal process together increase confidence in the correct functioning of the program under test.

Example:

```
Integer x, y
Input x, y
If(x<y)   ←this condition should be x≤ y
{
        Print f(x, y)
}
Else(x
{
        Print g(x, y)
}
```

- When the error is removed by changing the condition x<y to x≤*y*, the program fails again when the input values are the same. The latter failure is due to the error in function f. In this program, when the error in f is also removed, the program will be correct assuming that all other code is correct.

**Operational Profile(for sorting program)**

An operational profile is a numerical description of how a program is used. In accordance with the above definition, a program might have several operational profiles depending on its users.

| Operational profile 1 | |
|---|---|
| Sequence | probability |
| Numbers only | 0.9 |
| Alphanumeric strings | 0.1 |

| Operational profile 2 | |
|---|---|
| Sequence | probability |
| Numbers only | 0.1 |
| Alphanumeric strings | 0.9 |

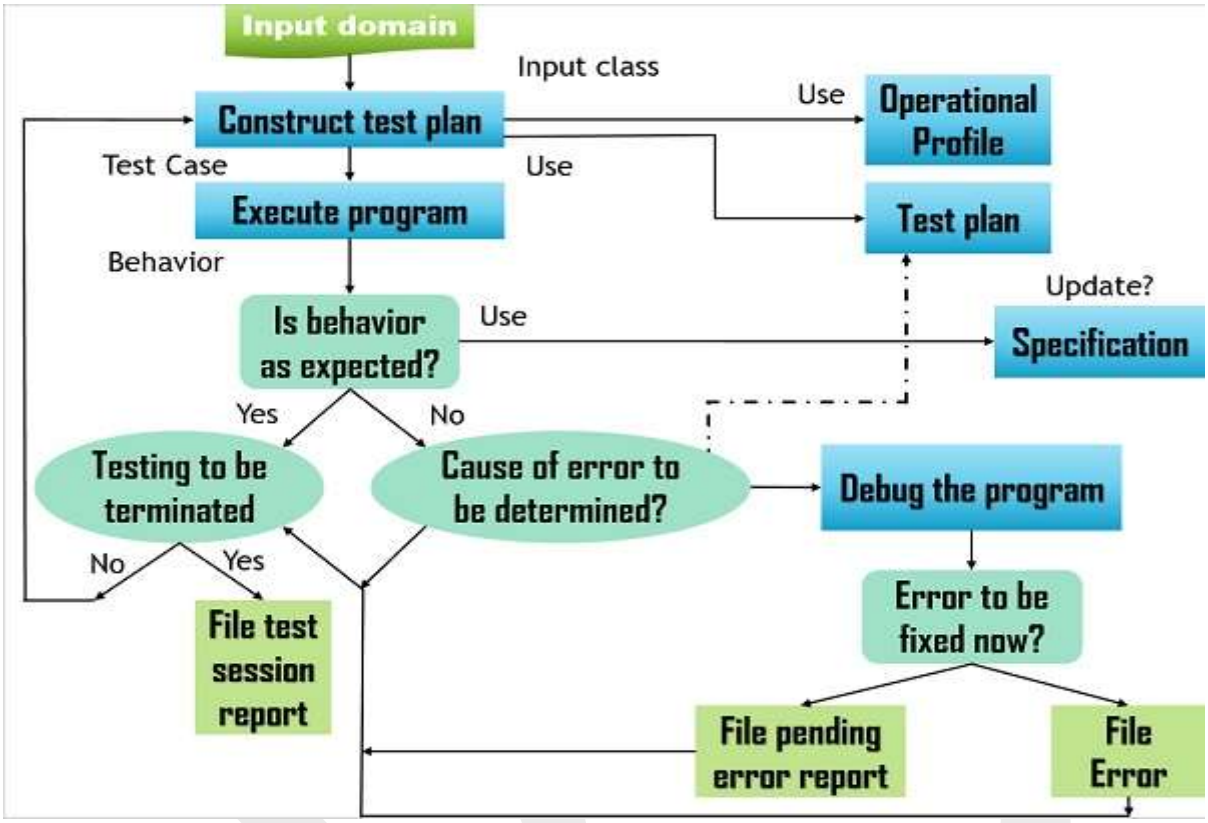
## Testing and Debugging

**Testing:** Testing is the process of determining if a program behaves as expected. In the process one may discover errors in the program under test. However, the testing reveals an error.

**Debugging**: The process used to determine the cause of this error and to remove it is known as debugging.

As illustrated in figure, testing and debugging are often used as two related activities in a cyclic manner. Steps are

1. Preparing a test plan
2. Constructing test data
3. Executing the program
4. Specifying program behaviour
5. Assessing the correctness of program behaviour
6. Construction of oracle




**Preparing a test plan:**

Example test plan: Consider following items such as the method used for testing, method for evaluating the adequacy of test cases, and method to determine if a program has failed or not. ***Test plan for sort:*** The sort program is to be tested to meet the requirements given in example

1. Execute the program on at least two input sequence one with "A" and the other with "D" as request characters.

2. Execute the program on an empty input sequence

3. Test the program for robustness against erroneous input such as "R" typed in as the request character.

4. All failures of the test program should be recorded in a suitable file using the company failure report form.

**Constructing Test Data:**

- A test case is a pair consisting of test data to be input to the program and the expected output.

- The test data is a set of values, one for each input variable.
- A test set is a collection of zero or ore cases.

Test case 1:

| | |
|---|---|
| Test data: | <"A" 12 -29 32 > |
| Expected output: | -29 12 32 |

Test case 2:

| | |
|---|---|
| Test data: | <"D" 12 -29 32.> |
| Expected output: | 32 12 -29 |

Test case 3:

| | |
|---|---|
| Test data: | <"A".> |
| Expected output: | No input to be sorted in ascending order |

Test case 4:

| | |
|---|---|
| Test data: | <"D".> |
| Expected output: | No input to be sorted in descending order |

Test case 5:

| | |
|---|---|
| Test data: | <"R" 3 17.> |
| Expected output: | Invalid request character; |

*valid characters: "A" and "D"*

Test case 6:

| | |
|---|---|
| Test data: | <"A" c 17.> |
| Expected output: | *Invalid number* |

**Executing the program:**

Execution of a program under test is the next significant step in the testing. Execution of this step for the sort program is most likely a trivial exercise. The complexity of actual program execution is dependent on the program itself. The output generated by the program may be saved in a file for subsequent examination by a tester.

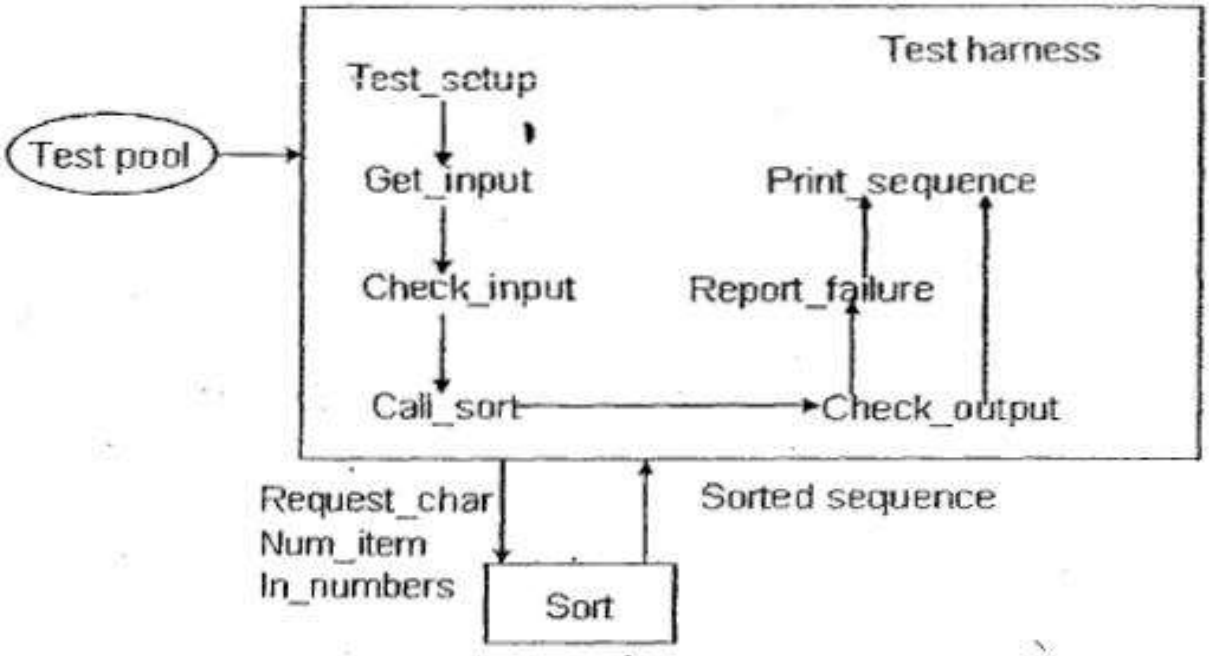



Figure: A simple test harness to test the *sort* program.

In preparing this test harness assume that:

(a) Sort is coded as a procedure

(b) The get-input procedure reads the request character & the sequence to be sorted into variables request_char, num_items and in_number, test_setup procedure-invoked first to set up the test includes identifying and opening the file containing tests.

- Check_outputprocedure serve as the oracle that checks if the program under test behaves correctly.
- Report_failure: output from sort is incorrect. May be reported via a message(or)saved in a file.
- Print_sequence: prints the sequence generated by the sort program. This also can be saved in file for subsequent examination.

**Specifying program behavior:**

State→ can be used to define program behaviour.

State transmission diagram, (or)
Simply state diagram
} → specify program behaviour

**State vector:** collecting the current values of program variables into a vector known as the state vector. An indication of where the control of execution is at any instant of time can be given by using an identifier associated with the next program statement.

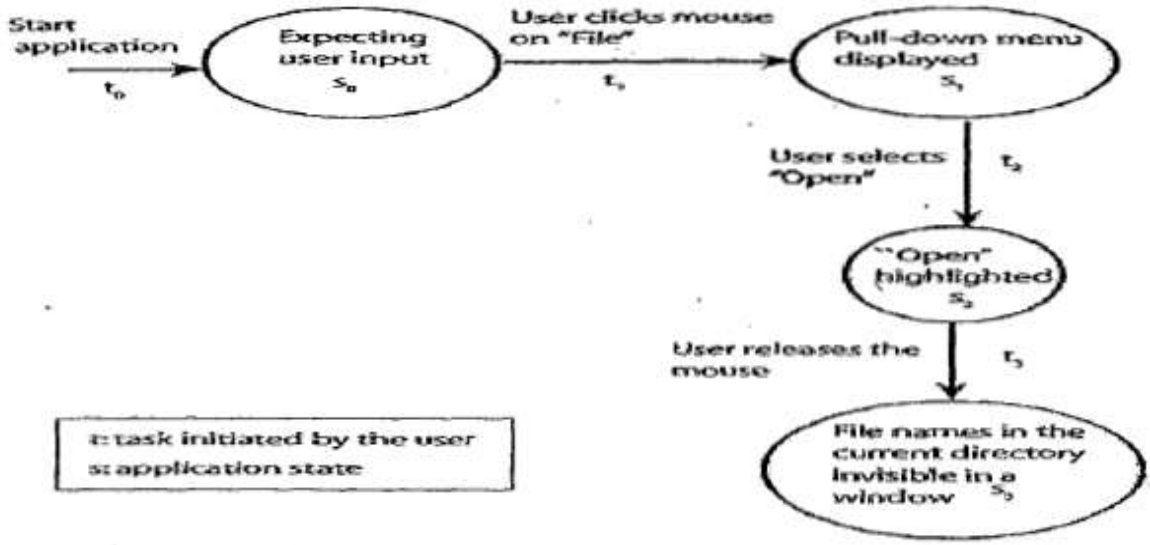


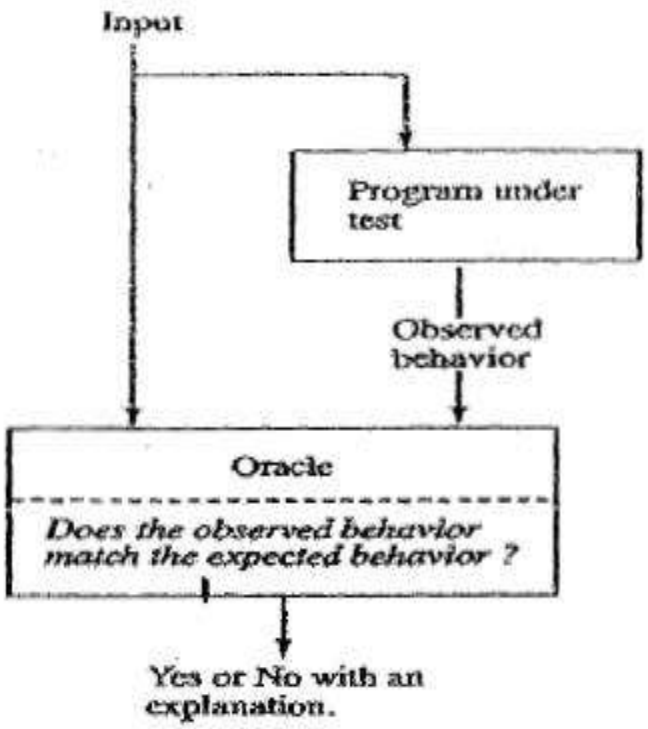

Figure: A state sequence for *myapp* showing how the application is expected to behave when the user selects the `open` option under the `file` menu

**Assessing the correctness of program**

Behaviour: It has two steps:

1. Observes the behaviour
2. Analyzes the observed behaviour.

Above task, extremely complex for large distributed system the entity that performs the task of checking the correctness of the observed behaviour is known as an oracle.

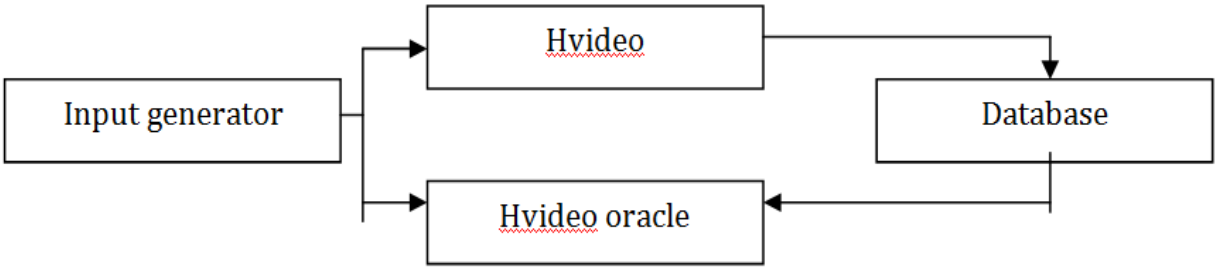



- But human oracle is the best available oracle.
- Oracle can also be programs designed to check the behaviour of other programs.

**Construction of oracles:**

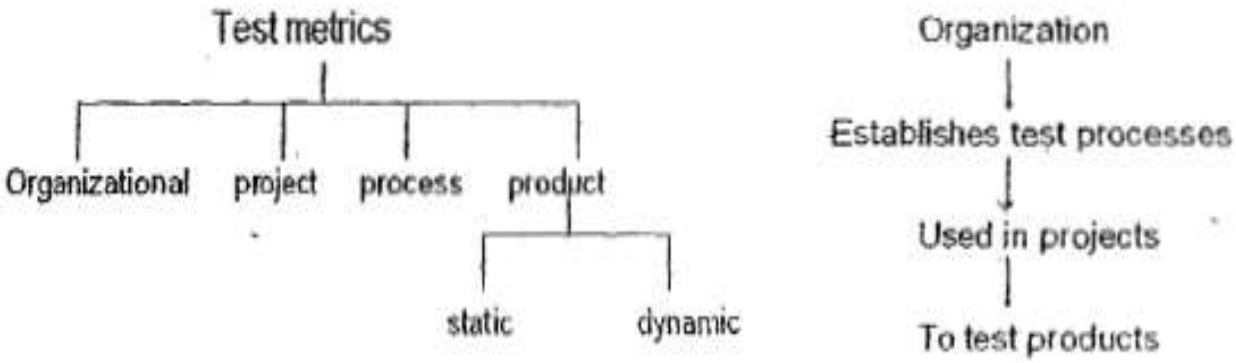
Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, Requires determination of I/O relationship.

Example: Consider a program named Hvideo that allows one to keep track of home videos. In the data entry mode, it displays a screen in which the user types in information about a DVD. In search mode, the program displays a screen into which a user can type some attribute of the video being searched for and set up a search criterion. To test Hvideo we need to create an oracle that checks whether the program function correctly in data entry and search nodes. The input generator generates a data entry request. The input generaor now requests the oracle to test if Hvideo performed its task correctly on the input given for data entry.




## TEST METRICS

The term metric refers to a standard of measurement. In software testing, there exist a variety of metrics.




There are four general core areas that assist in the design of metrics  schedule, quality, resources and size.

***Schedule related metrics:*** Measure actual completion times of various activities and compare these with estimated time to completion.

***Quality related metrics:*** Measure quality of a product or a process

***Resource related metrics:*** Measure items such as cost in dollars, man power and test executed.

***Size-related metrics:*** Measure size of various objects such as the source code and number of tests in a test suite

***Organizational metrics:*** Metrics at the level of an organization are useful in overall project planning and management. Ex: the number of defects reported after product release, averaged over

a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.

***Project metrics:***

Project metrics relate to a specific project, for example the I/O device testing project or a compiler project. These are useful in the monitoring and control of a specific project.

1. Actual/planned system test effort is one project metrics. Test effort could be measured in terms of the tester_man_months.

2. **Project metric=***no.of successfulteststotal /numberof testsinthesystemphase*

***Process metrics:***

Test process consists of several phases like unit test, integration test, system test, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the consttier it is to fix.

# Module 2

# Boundary Value Testing

Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. Input domain testing (also called "boundary value testing") is the best-known specification-based testing technique.There are two independent considerations that apply to input domain testing.

- The first asks whether or not we are concerned with invalid values of variables.
- The second consideration is whether we make the "single fault" assumption common to reliability theory.

Taken together, the two considerations yield four variations of boundary value testing:

1. Normal boundary value testing
2. Robust boundary value testing
3. Worst-case boundary value testing
4. Robust worst-case boundary value testing

Function, F, of two variables x1 and x2. When the function F is implemented as a program, the input variables x1 and x2 will have some (possibly unstated) boundaries:

$$a \leq x_1 \leq b$$

$$c \leq x_2 \leq d$$

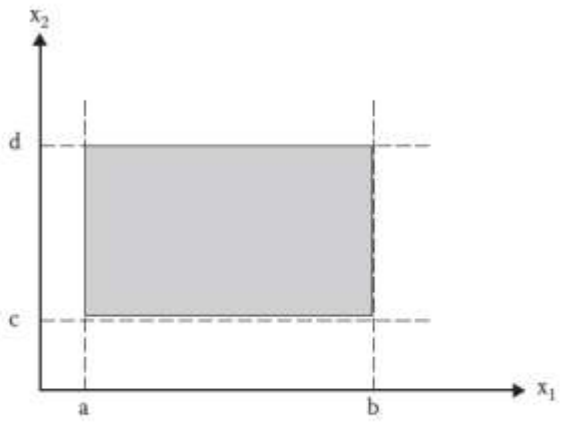



Figure 5.1 Input domain of a function of two variables.

The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle and including the boundaries is a legitimate input to the function F.the intervals [a, b] and [c, d] are referred to as the ranges of x1 and x2, so right away we have an overloaded term.

## Normal Boundary Value Testing

The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. A commercially available testing tool (originally named T) generates such test cases for a properly specified program. The T tool refers to these values as **min, min+, nom, max–, and max**. The robust forms add two values, min– and max+.

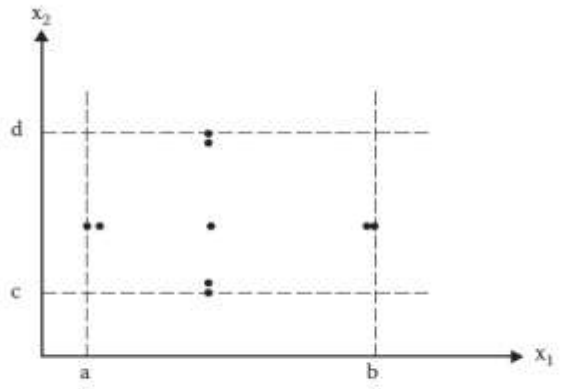



Figure 5.2 Boundary value analysis test cases for a function of two variables.

The normal boundary value analysis test cases for our function F of two variables (illustrated in Figure 5.2) are

$\{<X_{1nom}, X_{2min}>, <X_{1nom}, X_{2min+}>, <X_{1nom}, X_{2nom}>, <X_{1nom}, X_{2max-}>, <X_{1nom}, X_{2max}>, <X_{1min}, X_{2nom}>, <X_{1min+}, X_{2nom}>, <X_{1max-}, X_{2nom}>, <X_{1max}, X_{2nom}>\}$

**Generalizing Boundary Value Analysis**

The basic boundary value analysis technique can be generalized in two ways: by the number of variables and by the kinds of ranges.

- Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max–, and max values, repeating this for each variable.Thus, for a function of n variables, boundary value analysis yields **4n + 1** unique test cases.
- Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year.
    1. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on.
    2. We could define the variable month as an enumerated type {Jan., Feb., …, Dec.}.
    3. Either way, the values for min, min+, nom, max–, and max are clear from the context.
    4. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max–, and max are also easily determined.
    5. When no explicit bounds are present,The lower bound of side lengths is clearly 1 and we might impose an arbitrary upper limit such as 200 or 2000.
    6. Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, but no clear choice is available for the remaining three.
    7. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's PIN is a logical variable.

### *Limitations of Boundary Value Analysis*

- Boundary value analysis works well when the program to be tested is a function of several **independent variables** that represent bounded **physical quantities**. Mathematically, the variables need to be described by a true **ordering relation**, in which, for every pair <a, b> of values of a variable, it is possible to say that $a \leq b$ or $b \leq a$.
- The **physical quantity** criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important.
- As an example of **logical variables,** we might look at PINs or telephone numbers. It is hard to imagine what faults might be revealed by testing PIN values of 0000, 0001, 5000, 9998, and 9999. Sets of car colors, for example, or football teams, do not support an ordering relation; thus, no form of boundary value testing is appropriate for such variables.

## Robust Boundary Value Testing

Robust boundary value testing 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 extrema are exceeded with a value slightly greater than the **maximum (max+)** and a value slightly less than the **minimum (min–).** Robust boundary value test cases for our continuing example are shown in Figure 5.3.

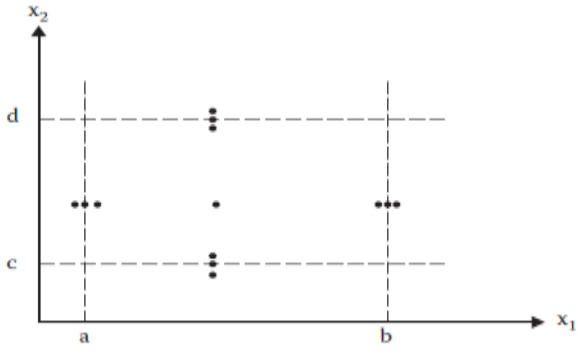



**Figure 5.3 Robustness test cases for a function of two variables.**

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs but with the expected outputs.

What happens when a physical quantity exceeds its maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it is the load capacity of a public elevator, we hope nothing special would happen. If it is a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling.

With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution.

## Worst-Case Boundary Value Testing

Both forms of boundary value testing, make the single fault assumption of reliability theory, Rejecting single-fault assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called "worst-case analysis" we use that idea here to generate worstcase test cases.

For each variable, we start with the five-element set that contains the **min, min+, nom, max–, and max values.** We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 5.4.

- Worst-case boundary value testing is requires worst-case testing for a function of n variables generates **$5^n$test cases.**

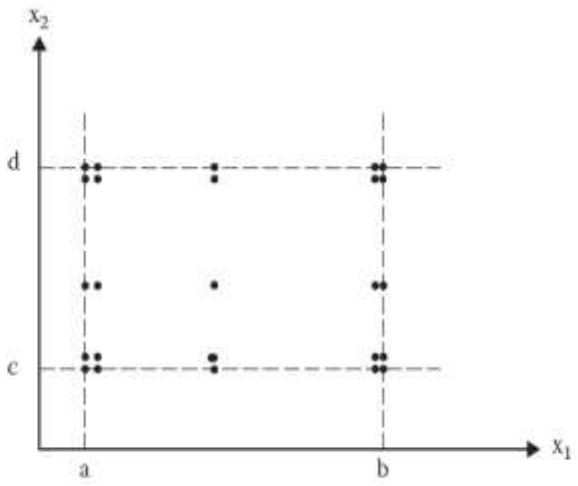



Figure 5.4 Worst-case test cases for a function of two variables.

- We could go to robust worst-case testing. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in **$7^n$ test cases**. Figure 5.5 shows the robust worst-case test cases for our two-variable function.

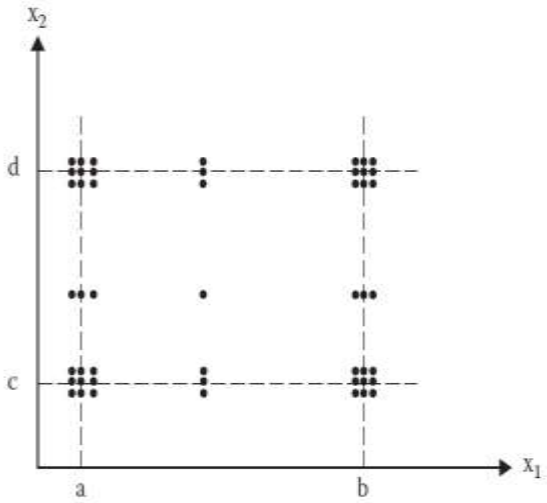



Figure 5.5 Robust worst-case test cases for a function of two variables.

## Robust worst testing the Triangle Problem

For triangle problem the lower bounds of the ranges are all 1 and arbitrarily take 200 as an upper bound. For each side, the test values are **{1, 2, 100, 199, 200}**. Robust boundary value test cases will add **{0, 201}.** Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted.

**Table 5.1 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 | 100 | 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 |

The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table 5.2 only lists the first 25 worst-case boundary value test cases for the triangle problem.

**Table 5.2 (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 a 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 |

## Test Cases for the NextDate Function

All 125 worst-case test cases for NextDate are listed in Table 5.3. Take some time to examine it for gaps of untested functionality and for redundant testing.

**Table 5.3 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 |

(continued)

**Table 5.3 Worst-Case Test Cases (Continued)**

| Case | Month | Day | Year | Expected Output |
|---|---|---|---|---|
| 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 |

(continued)

**Table 5.3 Worst-Case Test Cases (Continued)**

| *Case* | *Month* | *Day* | *Year* | *Expected Output* |
|---|---|---|---|---|
| 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 |
| 76 | 11 | 1 | 1812 | 11, 2, 1812 |
| 77 | 11 | 1 | 1813 | 11, 2, 1813 |
| 78 | 11 | 1 | 1912 | 11, 2, 1912 |
| 79 | 11 | 1 | 2011 | 11, 2, 2011 |
| 80 | 11 | 1 | 2012 | 11, 2, 2012 |
| 81 | 11 | 2 | 1812 | 11, 3, 1812 |
| 82 | 11 | 2 | 1813 | 11, 3, 1813 |
| 83 | 11 | 2 | 1912 | 11, 3, 1912 |
| 84 | 11 | 2 | 2011 | 11, 3, 2011 |
| 85 | 11 | 2 | 2012 | 11, 3, 2012 |
| 86 | 11 | 15 | 1812 | 11, 16, 1812 |

*(continued)*

**Table 5.3 Worst-Case Test Cases (Continued)**

| *Case* | *Month* | *Day* | *Year* | *Expected Output* |
|---|---|---|---|---|
| 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 |

*(continued)*

**Table 5.3 Worst-Case Test Cases (Continued)**

| Case | Month | Day | Year | Expected Output |
|---|---|---|---|---|
| 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 |

## Test Cases for the Commission Problem

Instead of going through 125 test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values derived from the output range, especially near the threshold points of $1000 and $1800 where the commission percentage changes. The output space of the commission is shown in Figure 5.6.

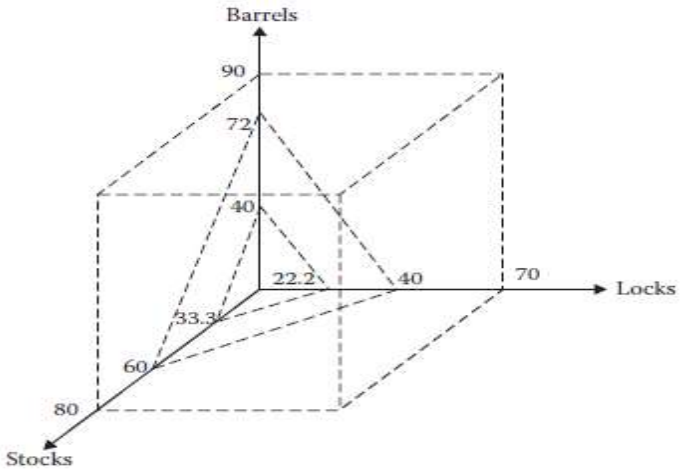



**Figure 5.6 Input space of the commission problem.**

**Table 5.4 Output Boundary Value Analysis Test Cases**

| *Case* | *Locks* | *Stocks* | *Barrels* | *Sales* | *Comm* | *Comment* |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 100 | 10 | Output minimum |
| 2 | 1 | 1 | 2 | 125 | 12.5 | Output minimum + |
| 3 | 1 | 2 | 1 | 130 | 13 | Output minimum + |
| 4 | 2 | 1 | 1 | 145 | 14.5 | Output minimum + |
| 5 | 5 | 5 | 5 | 500 | 50 | Midpoint |
| 6 | 10 | 10 | 9 | 975 | 97.5 | Border point – |
| 7 | 10 | 9 | 10 | 970 | 97 | Border point – |
| 8 | 9 | 10 | 10 | 955 | 95.5 | Border point – |
| 9 | 10 | 10 | 10 | 1000 | 100 | Border point |
| 10 | 10 | 10 | 11 | 1025 | 103.75 | Border point + |
| 11 | 10 | 11 | 10 | 1030 | 104.5 | Border point + |
| 12 | 11 | 10 | 10 | 1045 | 106.75 | Border point + |
| 13 | 14 | 14 | 14 | 1400 | 160 | Midpoint |
| 14 | 18 | 18 | 17 | 1775 | 216.25 | Border point – |
| 15 | 18 | 17 | 18 | 1770 | 215.5 | Border point – |
| 16 | 17 | 18 | 18 | 1755 | 213.25 | Border point – |
| 17 | 18 | 18 | 18 | 1800 | 220 | Border point |
| 18 | 18 | 18 | 19 | 1825 | 225 | Border point + |
| 19 | 18 | 19 | 18 | 1830 | 226 | Border point + |
| 20 | 19 | 18 | 18 | 1845 | 229 | Border point + |
| 21 | 48 | 48 | 48 | 4800 | 820 | Midpoint |
| 22 | 70 | 80 | 89 | 7775 | 1415 | Output maximum – |
| 23 | 70 | 79 | 90 | 7770 | 1414 | Output maximum – |
| 24 | 69 | 80 | 90 | 7755 | 1411 | Output maximum – |
| 25 | 70 | 80 | 90 | 7800 | 1420 | Output maximum |

# Module-2

# Equivalence Class Testing

The use of equivalence classes as the basis **for functional testing has two motivations**:

1) we would like to have a sense of **complete testing**,
2) We would hope to avoid redundancy.

Boundary value testing—looking at the tables of test cases, it is easy to see **massive redundancy,**and looking more closely, serious **gaps exist**. Equivalence class testing echoes the two deciding factors of boundary value testing, **robustness and the single/multiple fault assumption.**

Equivalence class testing has four distinct forms based on the two assumptions.

- The single versus multiple fault assumption yields the weak/strong distinction.
- second distinction: the focus on invalid data yieldsi.enormal versus robust.

Taken together, these two assumptions result in

1) Weak Normal Equivalence Class testing.
2) Strong Normal Equivalence Class testing.
3) Weak Robust Equivalence Class testing.
4) Strong Robust Equivalence Class testing.

**Two problems** occur with robust forms.The first is that, very often, the specification does not define what the expected output for an invalid input should be.The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs.

## Equivalence Classes

Equivalence classes are that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets, the union of which is the entire set.

- This has two important implications for testing—the fact that the entire set isrepresented provides a form of **completeness.**
- Disjointedness ensures a form of **nonredundancy.**

The idea of equivalence class testing is to identify test cases by **using one element from each equivalence class.** If the equivalence classes are chosen wisely, this greatly**reduces the potential redundancy among test cases.**

**Eg:**In the triangle problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100).Thus, they would be **redundant.**

- The four forms of equivalence class testing all address the problems of gaps and redundancies that are common to the four forms of boundary value testing.
- There will be one point of overlap—this occurs when equivalence classes are defined by bounded variables.
- In such cases, a hybrid of boundary value and equivalence class testing is appropriate. The International Software Testing Qualifications Board (ISTQB) syllabi refer to this as **"edge testing."**

## Traditional Equivalence Class Testing

Traditional equivalence class testing is nearly identical to weak robust equivalence class testing. This traditional form focuses on **invalid data values.**In the early years, it was the program user's responsibility **provides valid data**. There was no guarantee about results based on invalid data. The term soon became known as **GIGO("Garbage In, Garbage Out").**The usual response to GIGO was extensive **input validation** sections of a program.

In modern programming languages, especially those that feature strong data typing, and then to **graphical user interfaces (GUIs**) obviated much of the need for input data validation. Indeed, good use of user interface devices such **as drop-down lists and slider bars reduces** the likelihood **of bad input data.**

Traditional equivalence class testing echoes the process of boundary value testing. Figure 6.1 shows test cases for a function F of two variables x1 and x2,The extension to more realistic cases of n variables proceeds as follows:

1) Test F for valid values of all variables.
2) If step 1 is successful, then test F for invalid values of x1 with valid values of the remainingvariables.
3) Repeat step 2 for the remaining variables.

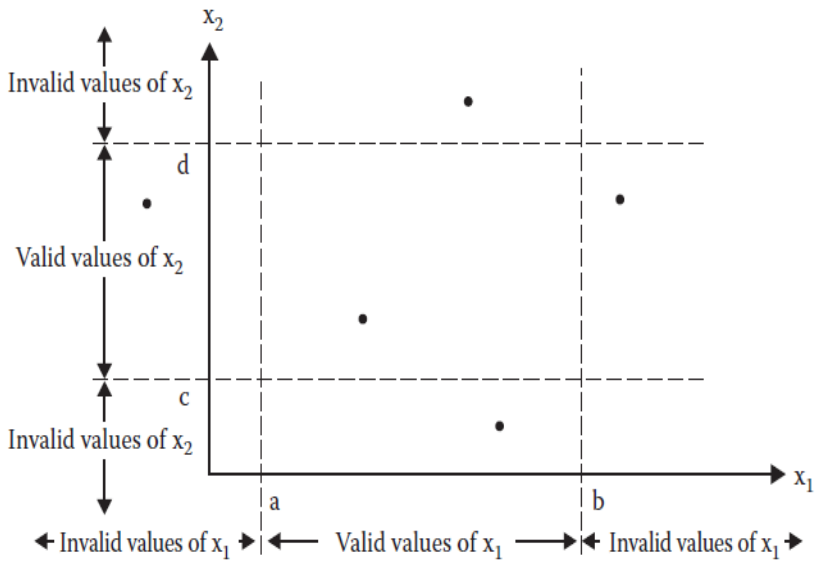



Figure 6.1 Traditional equivalence class test cases.

One clear advantage of this process is that it focuses on finding faults due to invalid data. Since the GIGO concern was on invalid data, the kinds of combinations that we saw in the worst-case variations of boundary value testing were ignored.Figure 6.1 shows the **five test cases for this process for our continuing function F of two variables.**

## Improved Equivalence Class Testing

When F is implemented as a program, the input variables x1 and x2 will have the following boundaries, and intervals within the boundaries:

$$a \le x_1 \le d, \text{ with intervals } [a, b), [b, c), [c, d]$$
$$e \le x_2 \le g, \text{ with intervals } [e, f), [f, g]$$

Where square brackets and parentheses denote, respectively, closed and open interval endpoints.These ranges are equivalence classes. Invalid values of x1 and x2 are **x1 <a, x1> d,** and **x2 <e, x2> g.** The equivalence classes of valid values are,

V1 = {x1: a ≤ x1 < b}, V2 = {x1: b ≤ x1 < c}, V3 = {x1: c ≤ x1 ≤ d},
V4 = {x2: e ≤ x2 <f }, V5 = {x2: f ≤ x2 ≤ g}

The equivalence classes of invalid values are

NV1 = {x1: x1 < a}, NV2 = {x1: d < x1}, NV3 = {x2: x2 < e}, NV4 = {x2: g < x2}

The equivalence classes V1, V2, V3, V4, V5, NV1, NV2, NV3, and NV4 are disjoint, and their union is the entire plane.

### *Weak Normal Equivalence Class Testing*

Weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case.

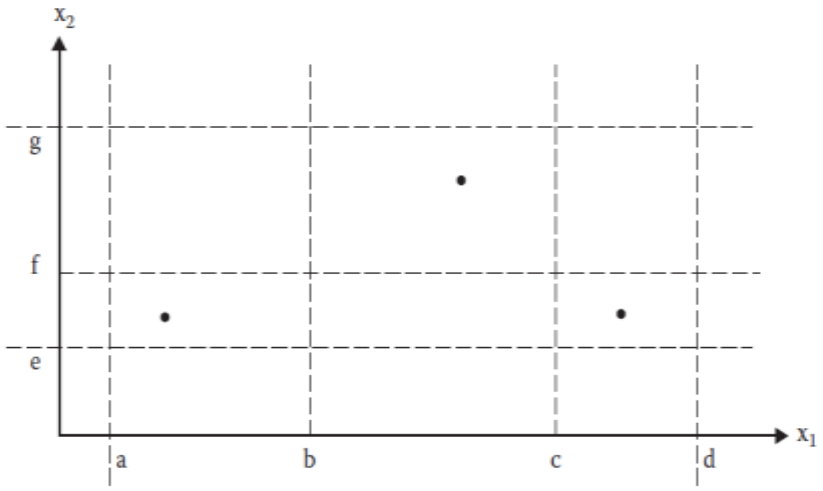



Figure 6.2 Weak normal equivalence class test cases.

For the running example, we would end up with the three weak equivalence class test cases shown in Figure 6.2. These three test cases use one value from each equivalence class.

- The test case in the lower left rectangle corresponds to a value of x1 in the class [a, b), and to a value of x2 in the class [e, f ).
- The test case in the upper center rectangle corresponds to a value of x1 in the class [b, c) and to a value of x2 in the class [f, g].
- The third test case could be in either rectangle on the right side of the valid values.

There could be a problem with x1, or a problem with x2, or maybe an interaction between the two. This ambiguity is the reason for the **"weak" designation**. If the expectation of failure is low, as it is for regression testing, this can be an acceptable choice. When more fault isolation is then, the stronger forms is required.

### ***Strong Normal Equivalence Class Testing***

Strong equivalence class testing is based on the **multiple fault assumption,** so we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 6.3.The Cartesian product guarantees that we have a notion of **"completeness" in two senses**: we **cover all the equivalence classes**, and we have **one of each possible combination of inputs**.

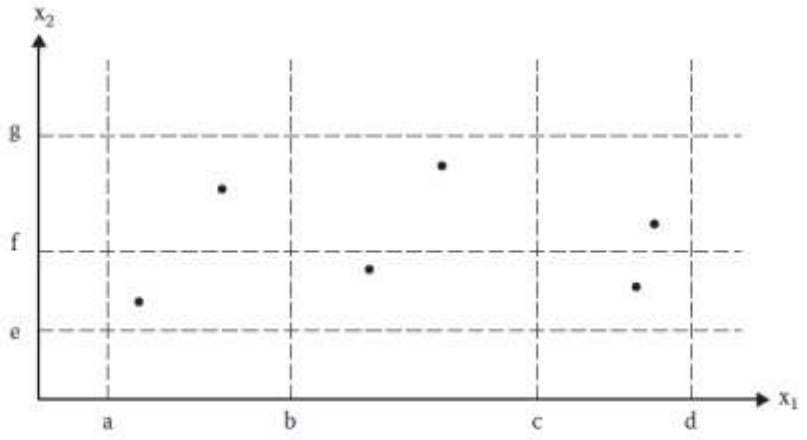



Figure 6.3 Strong normal equivalence class test cases.

## *Weak Robust Equivalence Class Testing*

How can something be both weak and robust? **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.

The two additional test cases cover all four classes of invalid values is shown in figure 6.4. The process is similar to that for boundary value testing:
1. **For valid inputs, use one value from each valid class** (as in what we have called weak normal equivalence class testing). (Note that each input in these test cases will be valid.)
2. **For invalid inputs, a test case will have one invalid value and the remaining values will all be valid.** (Thus, a "single failure" should cause the test case to fail.)

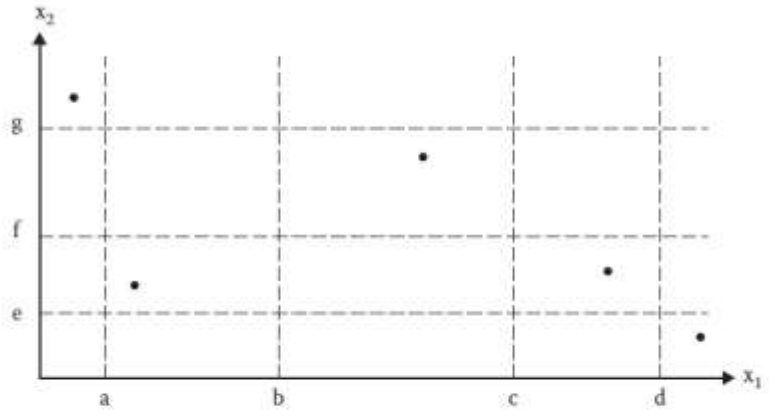



Figure 6.4 Weak robust equivalence class test cases.

There is a potential **problem**with these test cases. **Consider the test cases in the upper left and lower right corners. Each of the test cases represents values from two invalid equivalence classes. Failure of either of these could be due to the interaction of two variables**. Figure 6.5 presents a compromise **between "pure" weak normal equivalence class testing** and its robust extension.

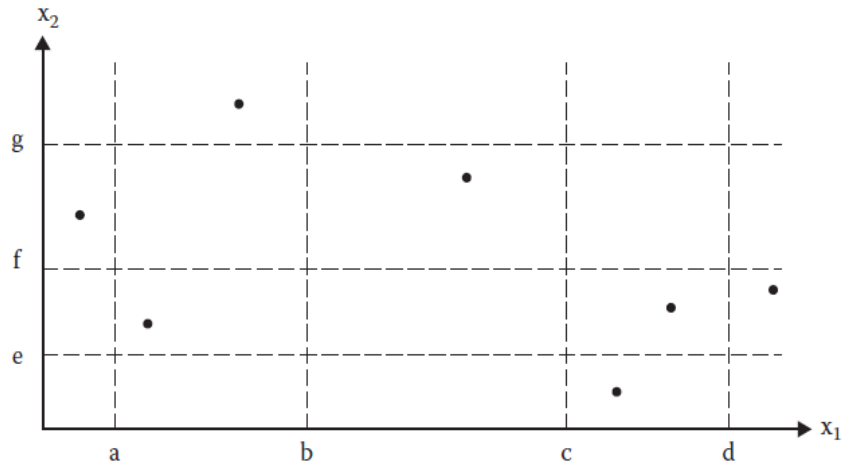



Figure 6.5 Revised weak robust equivalence class test cases.

## ***Strong Robust Equivalence Class Testing***

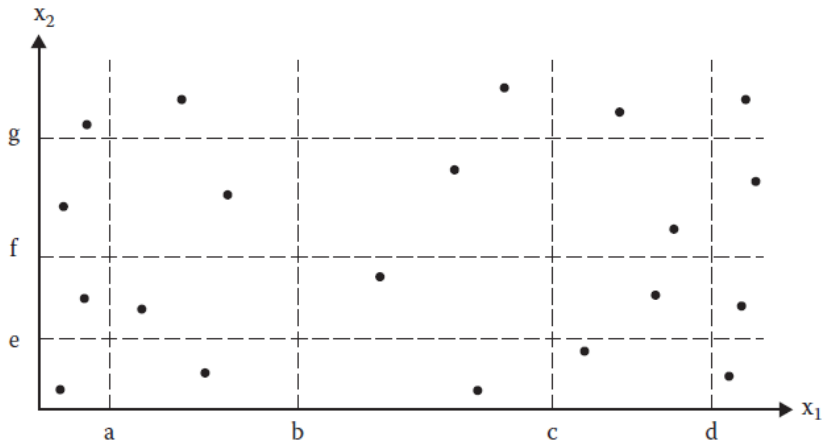



Figure 6.6 Strong robust equivalence class test cases.

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, as shown in Figure 6.6.

## Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: NotATriangle, Scalene, Isosceles, and Equilateral. We can use these 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}

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 |

- Because no valid subintervals of variables a, b, and c exist, the strong normal equivalence classtest cases are identical to the weak normal equivalence class test cases.
- 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 numbergreater 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 |
| WR4 | 201 | 5 | 5 | Value of a is not in the range of permitted values |
| WR5 | 5 | 201 | 5 | Value of b is not in the range of permitted values |
| WR6 | 5 | 5 | 201 | Value of c is not in the range of permitted values |

Here is one "corner" of the cube in three-space of the additional **strong robust equivalence**class test cases:

| *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 | Values of a, b are not in the range of permitted values |
| SR5 | 5 | –1 | –1 | Values of b, c are not in the range of permitted values |
| SR6 | –1 | 5 | –1 | Values of a, c are not in the range of permitted values |
| SR7 | –1 | –1 | –1 | Values of a, b, c are not in the range of permitted values |

Equivalence class testing is clearly sensitive to the equivalence relation used to define classes. Here is another instance of craftsmanship.

$D1 = \{<a, b, c>: a = b = c\}$
$D2 = \{<a, b, c>: a = b, a \neq c\}$
$D3 = \{<a, b, c>: a = c, a \neq b\}$
$D4 = \{<a, b, c>: b = c, a \neq b\}$
$D5 = \{<a, b, c>: a \neq b, a \neq c, b \neq c\}$
Invalid cases for NOTATriangle
$D6 = \{<a, b, c>: a \geq b + c\}$
$D7 = \{<a, b, c>: b \geq a + c\}$
$D8 = \{<a, b, c>: c \geq a + b\}$

## Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation.NextDate is a function of three variables: month, day, and year, and these have intervals of valid values defined as follows:

$M1 = \{month: 1 \leq month \leq 12\}$
$D1 = \{day: 1 \leq day \leq 31\}$
$Y1 = \{year: 1812 \leq year \leq 2012\}$

The invalid equivalence classes are

M2 = {month: month < 1}
M3 = {month: month > 12}
D2 = {day: day < 1}
D3 = {day: day > 31}
Y2 = {year: year < 1812}
Y3 = {year: year > 2012}

Because the **number of valid classes equals the number of independent variables,** only oneweak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case:

| *Case ID* | *Month* | *Day* | *Year* | *Expected Output* |
|---|---|---|---|---|
| WN1, SN1 | 6 | 15 | 1912 | 6/16/1912 |

Here is the full set of **weak robust test cases**:

| *Case ID* | *Month* | *Day* | *Year* | *Expected Output* |
|---|---|---|---|---|
| WR1 | 6 | 15 | 1912 | 6/16/1912 |
| 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 |

As with the triangle problem, here is one “corner” of the cube in three-space of the additional **strong robust equivalence class test cases**:

| 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<br>Value of day not in the range 1 ... 31 |
| SR5 | 6 | –1 | 1811 | Value of day not in the range 1 ... 31<br>Value of year not in the range 1812 ... 2012 |
| SR6 | –1 | 15 | 1811 | Value of month not in the range 1 ... 12<br>Value of year not in the range 1812 ... 2012 |
| SR7 | –1 | –1 | 1811 | Value of month not in the range 1 ... 12<br>Value of day not in the range 1 ... 31<br>Value of year not in the range 1812 ... 2012 |

The problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \leq day \leq 28$}
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}

### *Weak and strong normal Equivalence Class Test Cases*

These classes yield the following weak normal equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

| 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 |

The strong normal equivalence class test cases for the revised classes are as follows:

| 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 | 7/01/2000 |
| SN8 | 6 | 30 | 1996 | 7/01/1996 |
| SN9 | 6 | 30 | 2002 | 7/01/2002 |
| 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 |

| Case ID | Month | Day | Year | Expected Output |
|---|---|---|---|---|
| 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 | 7/30/2002 |
| SN19 | 7 | 30 | 2000 | 7/31/2000 |
| SN20 | 7 | 30 | 1996 | 7/31/1996 |
| SN21 | 7 | 30 | 2002 | 7/31/2002 |
| 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 | 2/15/2002 |
| 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 |

## Equivalence Class Test Cases for the Commission Problem

The input domain of the commission problem is "naturally" partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input; the other two are invalid.

The valid classes of the input variables are

L1 = {locks: $1 \leq locks \leq 70$}
L2 = {locks = –1} (occurs if locks = –1 is used to control input iteration)
S1 = {stocks: $1 \leq stocks \leq 80$}
B1 = {barrels: $1 \leq barrels \leq 90$}

The corresponding invalid classes of the input variables are

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}

- Sales is a function of the number of locks, stocks, and barrels sold:

  **Sales = 45*locks + 30 *stocks + 25 *barrels**

- Equivalence classes of three variables by commission ranges:

  **S1 = {<locks, stocks, barrels>: sales ≤ 1000}**
  **S2 = {<locks, stocks, barrels>: 1000 < sales ≤ 1800}**
  **S3 = {<locks, stocks, barrels>: sales > 1800}**

- The salesperson's commission as follows: 10% on salesup to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800.

The fact that our input is a triplet means that we no longer take test cases from a Cartesian product.

| *Test Case* | *Locks* | *Stocks* | *Barrels* | *Sales* | *Commission* |
|---|---|---|---|---|---|
| OR1 | 5 | 5 | 5 | 500 | 50 |
| OR2 | 15 | 15 | 15 | 1500 | 175 |
| OR3 | 25 | 25 | 25 | 2500 | 360 |

We will have **eight weak robust test cases**.

| 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 stocks not in the range 1 ... 80 |
| WR6 | 35 | 81 | 45 | Value of stocks not in the range 1 ... 80 |
| WR7 | 35 | 40 | –1 | Value of barrels not in the range 1 ... 90 |
| WR8 | 35 | 40 | 91 | Value of barrels not in the range 1 ... 90 |

Here is one "corner" of the cube in 3-space of the additional strong robust equivalence class test cases:

| 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 stocks not in the range 1 ... 80 |
| SR3 | 35 | 40 | –2 | Value of barrels not in the range 1 ... 90 |
| SR4 | –2 | –1 | 45 | Value of locks not in the range 1 ... 70<br>Value of stocks not in the range 1 ... 80 |
| SR5 | –2 | 40 | –1 | Value of locks not in the range 1 ... 70<br>Value of barrels not in the range 1 ... 90 |
| SR6 | 35 | –1 | –1 | Value of stocks not in the range 1 ... 80<br>Value of barrels not in the range 1 ... 90 |
| SR7 | –2 | –1 | –1 | Value of locks not in the range 1 ... 70<br>Value of stocks not in the range 1 ... 80<br>Value of barrels not in the range 1 ... 90 |

# Module-2

# Fault-Based Testing

## Overview

Experience with common software faults sometimes leads to improvements in design methods and programming languages.The basic concept of fault-based testing is to **select test cases that would distinguish the program under test from alternative programs that contain hypothetical faults**. This is usually approached by modifying the program under test to actually produce the hypothetical faulty programs.**Fault seeding** can be used to evaluate the thoroughness of a test suite (that is, as an element of a test adequacy criterion), or for selecting test cases to augment a test suite, or to **estimate the number of faults in a program**.

## Assumptions in Fault-Based Testing

- The effectiveness of fault-based testing depends on the **quality of the fault model** and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice, the **seeded faults are small syntactic changes,** like replacing one variable reference by another in an expression, or changing a comparison from < to <=.
- The program from small variantsi.e the **competent programmer hypothesis**, anassumption that the program under test is "close to" (in the sense of textual difference)a correct program.
- Some program faults are indeed simple typographical errors, and others that involvean **error of logic** will result in much more complex differences inprogram text. This may not invalidate fault-based testing with a simpler fault model,provided test cases sufficient for detecting the simpler faults are sufficient also **for complex fault**. This is known as the **coupling effect.**
- Fault-based testing can guarantee fault detection only if the competent programmerhypothesis and the coupling effect hypothesis hold.

## Fault-Based Testing: Terminology

**Original program:** The program unit (e.g., C function or Java class) to be tested.

**Program location:** A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and Boolean expressions, and procedure calls.

**Alternate expression:** Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

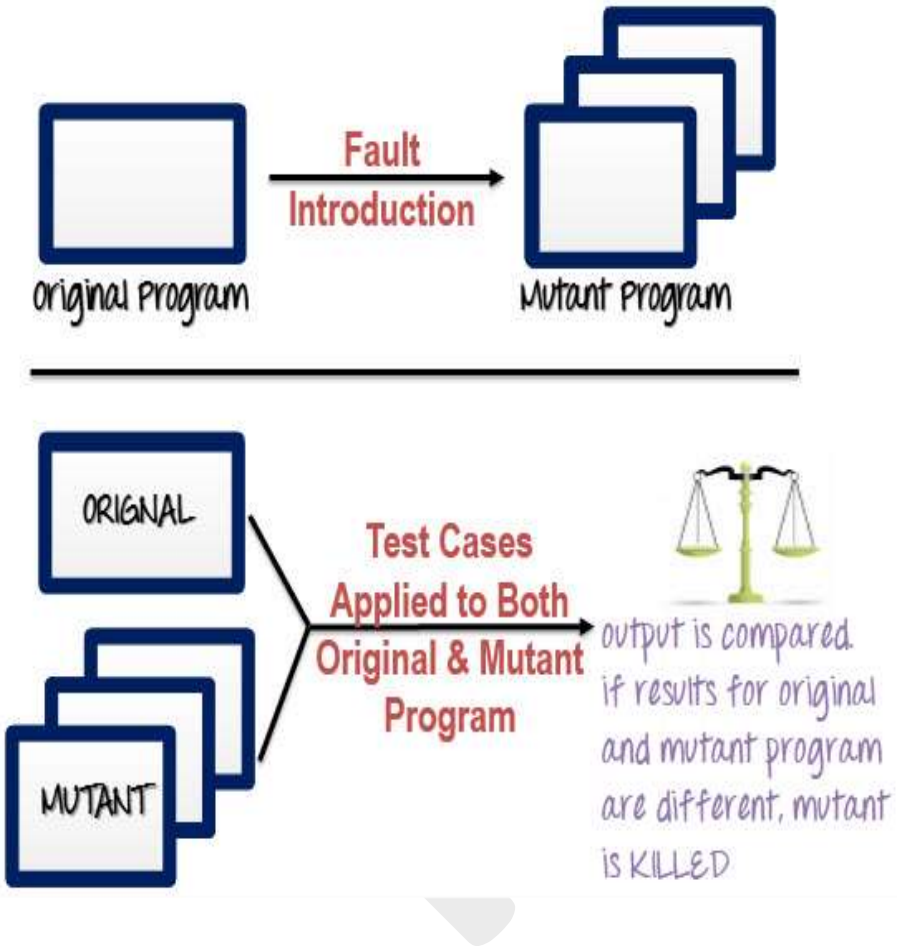
**Alternate program:** A program obtained from the original program by substituting an alternate expression for the text at some program location.

**Distinct behavior of an alternate program $R$ for a test $t$:** The behavior of an alternate program $R$ is distinct from the behavior of the original program $P$ for a test $t$, if $R$ and $P$ produce a different result for $t$, or if the output of $R$ is not defined for $t$.

**Distinguished set of alternate programs for a test suite $T$:** A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in $T$.

## Mutation Analysis

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. **Variants are created by "seeding" faults,** that is, by making a small change to the program under test following a pattern in the fault model. The patterns for changing program text are called **mutation operators**, and each variant program is operator called a **mutant.**



## Mutation Analysis: Terminology

**Original program under test:** The program or procedure (function) to be tested.

**Mutant:** A program that differs from the original program for one syntactic element (e.g., a statement, a condition, a variable, a label).

**Distinguished mutant:** A mutant that can be distinguished for the original program by executing at least one test case.

**Equivalent mutant:** A mutant that cannot be distinguished from the original program.

**Mutation operator:** A rule for producing a mutant program by syntactically modifying the original program.

- We say a **mutant is valid** if it is syntactically correct. A mutant obtained from the program of Figure 16.1 by substituting while for switch in the statement at line 13 would not be valid, since it would result in a compile-time error.
- We say a **mutant is D useful mutant** useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases. A mutant obtained by substituting 0 for 1000 in the statement at line 4 would be valid, but not useful, since the mutant would be distinguished from the program under test by all inputs and thus would not give any useful information on the effectiveness of a test suite.
- Defining **mutation operators that produce valid and useful mutations** is a nontrivial task.

```
1
2  /** Convert each line from standard input */
3  void transduce() {
4      #define BUFLEN 1000
5      char buf[BUFLEN];   /* Accumulate line into this buffer   */
6      int  pos = 0;       /* Index for next character in buffer */
7
8      char inChar; /* Next character from input */
9
10     int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12     while ((inChar = getchar()) != EOF ) {
13       switch (inChar) {
14       case LF:
15         if (atCR) {   /* Optional DOS LF */
16           atCR = 0;
17         } else {      /* Encountered CR within line */
18           emit(buf, pos);
19           pos = 0;
20         }
21         break;
22       case CR:
23         emit(buf, pos);
24         pos = 0;
25         atCR = 1;
26         break;
27       default:
28         if (pos >= BUFLEN-2) fail("Buffer overflow");
29         buf[pos++] = inChar;
30       } /* switch */
31     }
32     if (pos > 0) {
33       emit(buf, pos);
34     }
35 }
```

*Figure 16.1: Program* transduce *converts line endings among Unix, DOS, and Macintosh conventions. The main procedure, which selects the output line end convention, and the output procedure* emit *are not shown.*

| ID | Operator | Description | Constraint |
|---|---|---|---|
| *Operand Modifications* | | | |
| crp | constant for constant replacement | replace constant $C1$ with constant $C2$ | $C1 \neq C2$ |
| scr | scalar for constant replacement | replace constant $C$ with scalar variable $X$ | $C \neq X$ |
| acr | array for constant replacement | replace constant $C$ with array reference $A[I]$ | $C \neq A[I]$ |
| scr | struct for constant replacement | replace constant $C$ with struct field $S$ | $C \neq S$ |
| svr | scalar variable replacement | replace scalar variable $X$ with a scalar variable $Y$ | $X \neq Y$ |
| csr | constant for scalar variable replacement | replace scalar variable $X$ with a constant $C$ | $X \neq C$ |
| asr | array for scalar variable replacement | replace scalar variable $X$ with an array reference $A[I]$ | $X \neq A[I]$ |
| ssr | struct for scalar replacement | replace scalar variable $X$ with struct field $S$ | $X \neq S$ |
| vie | scalar variable initialization elimination | remove initialization of a scalar variable | |
| car | constant for array replacement | replace array reference $A[I]$ with constant $C$ | $A[I] \neq C$ |
| sar | scalar for array replacement | replace array reference $A[I]$ with scalar variable $X$ | $A[I] \neq X$ |
| cnr | comparable array replacement | replace array reference with a comparable array reference | |
| sar | struct for array reference replacement | replace array reference $A[I]$ with a struct field $S$ | $A[I] \neq S$ |
| *Expression Modifications* | | | |
| abs | absolute value insertion | replace $e$ by abs($e$) | $e < 0$ |
| aor | arithmetic operator replacement | replace arithmetic operator $\psi$ with arithmetic operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| lcr | logical connector replacement | replace logical connector $\psi$ with logical connector $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| ror | relational operator replacement | replace relational operator $\psi$ with relational operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| uoi | unary operator insertion | insert unary operator | |
| cpr | constant for predicate replacement | replace predicate with a constant value | |
| *Statement Modifications* | | | |
| sdl | statement deletion | delete a statement | |
| sca | switch case replacement | replace the label of one case with another | |
| ses | end block shift | move } one statement earlier and later | |

*Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.*

## Fault-Based Adequacy Criteria

Given a program and a test suite T, mutation analysis consists of the following steps:

1. **Select mutation operators**: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.
2. **Generate mutants**: Mutants are generated mechanically by applying mutation operators to the original program.
3. **Distinguish mutants:** Execute the original program and each generated mutant with the test cases in T. A mutant is killed when it can be distinguished from the original program.

| ID | Operator | line | Original/ Mutant | 1U | 1D | 2U | 2D | 2M | End | Long | Mixed |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $M_i$ | ror | 28 | (pos >= BUFLEN−2)<br>(pos == BUFLEN−2) | - | - | - | - | - | - | - | - |
| $M_j$ | ror | 32 | (pos > 0)<br>(pos >= 0) | - | x | x | x | x | - | - | - |
| $M_k$ | sdl | 16 | atCR = 0<br>*nothing* | - | - | - | - | - | - | - | - |
| $M_l$ | ssr | 16 | atCR = 0<br>pos = 0 | - | - | - | - | - | - | - | x |

| Test case | Description |
|---|---|
| 1U | One line, Unix line-end |
| 1D | One line, DOS line-end |
| 2U | Two lines, Unix line-end |
| 2D | Two lines, DOS line-end |

| Test case | Description |
|---|---|
| 2M | Two lines, Mac line-end |
| End | Last line not terminated with line-end sequence |
| Long | Very long line (greater than buffer length) |
| Mixed | Mix of DOS and Unix line ends in the same file |

*Figure 16.3: A sample set of mutants for program* Transduce *generated with mutation operators from Figure 16.2.* x *indicates the mutant is killed by the test case in the column head.*

Figure 16.3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in Figure 16.2. Test suite TS

TS = {1U,1D,2U,2D,2M,End,Long}

**kills Mj,** which can be distinguished from the original program by test cases 1D, 2U, 2D, and 2M. **Mutants Mi, Mk, and Ml are not distinguished from the original program**by any test in TS. We say that **mutants not killed** by a test suite are live.

A mutant can remain live for two reasons:

- The mutant can be distinguished from the original program, but the test suite T does not contain a test case that distinguishes them (i.e., the test suite is not adequate with respect to the mutant).
- The mutant cannot be distinguished from the original program by any test case (i.e., the mutant is equivalent to the original program).

The adequacy of the test suite TS evaluated with respect to the four mutants of Figure 16.3 is 25%. However, we can easily observe that mutant Mi is equivalent to the original program (i.e., no input would distinguish it). Conversely, mutants Mk and Ml seem to be nonequivalent to the original program: There should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of TS, measured after eliminating the equivalent mutant Mi, is 33%. Mutant Ml is killed by test case Mixed, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines.

**Case Study (Differentiate between Mutation and Structural Testing)**

### Mutation Analysis vs. Structural Testing

For typical sets of syntactic mutants, a mutation-adequate test suite will also be adequate with respect to simple structural criteria such as statement or branch coverage. Mutation adequacy can simulate and subsume a structural coverage criterion if the set of mutants can be killed only by satisfying the corresponding test coverage obligations.

Statement coverage can be simulated by applying the mutation operator sdl (statement deletion) to each statement of a program. To kill a mutant whose only difference from the program under test is the absence of statement *S* requires executing the mutant and the program under test with a test case that executes *S* in the original program. Thus to kill all mutants generated by applying the operator sdl to statements of the program under test, we need a test suite that causes the execution of each statement in the original program.

Branch coverage can be simulated by applying the operator cpr (constant for predicate replacement) to all predicates of the program under test with constants *True* and *False*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *False*, we need to execute the mutant and the program under test with a test case that causes the execution of the *True* branch of *P*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *True*, we need to execute the mutant and the program under test with a test case that causes the execution of the *False* branch of *P*.

A test suite that satisfies a structural test adequacy criterion may or may not kill all the corresponding mutants. For example, a test suite that satisfies the statement coverage adequacy criterion might not kill an sdl mutant if the value computed at the statement does not affect the behavior of the program on some possible executions.

## Variations on Mutation Analysis

1. **Strong mutation:** the output of the program is compared at the end of the program execution.

   The mutation analysis process described in the preceding sections, which kills mutants based on the outputs produced by execution of test cases, is known as strong mutation.

   **Disadvantage:** The time and space required for compiling all mutants and for executing all test cases for each mutant maybe impractical.
2. **Weak mutation:** the state of the program is compared after every execution of the component.

   Weak mutation analysis **decreases the number of tests** to be executed **by killing mutants** whenthey produce a different intermediate state**, rather than waiting for a difference in the final result** or observable program behavior.
3. **Meta mutant:program is divided into segments** containing original and mutated source code,with a mechanism to select which segments to execute. **Two copies of the meta-mutant are executed in parallel**, one with only original program code selected and the otherwith a set of live mutants selected. **Execution is paused after each segment** to comparethe program state of the two versions.
   - **If the state is equivalent**, execution resumes withthe next segment of original and mutated code.
   - **If the state differs,** the mutant is markedas dead, and execution of original and mutated code is restarted with a new selectionof live mutants.
4. **Statistical mutation:** To **estimate the number of faults remaining in a program**. Usually **we know only the number of faults that have been detected, and** not the number that remains. However, again to the extent that the**fault model is a valid statistical model of actual fault occurrence,** we can estimate thatthe ratio of actual faults found to those still remaining should be similar to the ratio ofseeded faults found to those still remaining.

**Case study 2:**

## Hardware Fault-based Testing

Fault-based testing is widely used for semiconductor and hardware system validation and evaluation both for evaluating the quality of test suites and for evaluating fault tolerance.

Semiconductor testing has conventionally been aimed at detecting random errors in fabrication, rather than design faults. Relatively simple fault models have been developed for testing semiconductor memory devices, the prototypical faults being "stuck-at-0" and "stuck-at-1" (a gate, cell, or pin that produces the same logical value regardless of inputs). A number of more complex fault models have been developed for particular kinds of semiconductor devices (e.g., failures of simultaneous access in dual-port memories). A test vector (analogous to a test suite for software) can be judged by the number of hypothetical faults it can detect, as a fraction of all possible faults under the model.

Fabrication of a semiconductor device, or assembly of a hardware system, is more analogous to copying disk images than to programming. The closest analog of software is not the hardware device itself, but its design — in fact, a high-level design of a semiconductor device is essentially a program in a language that is compiled into silicon. Test and analysis of logic device designs faces the same problems as test and analysis of software, including the challenge of devising fault models. Hardware design verification also faces the added problem that it is much more expensive to replace faulty devices that have been delivered to customers than to deliver software patches.

In evaluation of fault tolerance in hardware, the usual approach is to modify the state or behavior rather than the system under test. Due to a difference in terminology between hardware and software testing, the corruption of state or modification of behavior is called a "fault," and artificially introducing it is called "fault injection." Pin-level fault injection consists of forcing a stuck-at-0, a stuck-at-1, or an intermediate voltage level (a level that is neither a logical 0 nor a logical 1) on a pin of a semiconductor device. Heavy ion radiation is also used to inject random faults in a running system. A third approach, growing in importance as hardware complexity increases, uses software to modify the state of a running system or to simulate faults in a running simulation of hardware logic design.

# MODULE-3

# (STRUCTURAL TESTING)

# Path Testing

## Program Graphs

### Definition

Given a program written in an imperative programming language, its **program graph is a directed graph** in which nodes are statement fragments, and edges represent flow of control.If i and j are nodes in the program graph, an edge exists from node i to node j if and only if the statement fragment corresponding to node j can be executed immediately after the statement fragment corresponding to node i.

### Style Choices for Program Graphs

Program graph from a given program is an easy process. It is illustrated here with four of the basic structured programming constructs (Figure 8.1), and also with our pseudocodeimplementation of the triangle program.

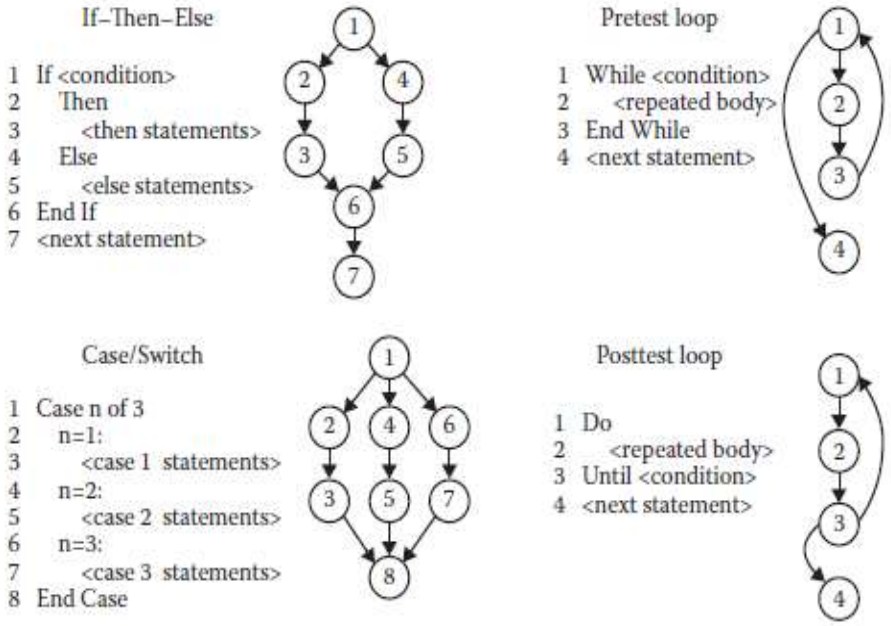



**Figure 8.1 Program graphs of four structured programming constructs.**

A program graph of the second version of the triangle problem is given in Figure 8.2.

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10   Then IsATriangle = True
11   Else IsATriangle = False
12 EndIf
13 If IsATriangle
14   Then  If (a = b) AND (b = c)
15           Then Output ("Equilateral")
16           Else  If (a≠b) AND (a≠c) AND (b≠c)
17                    Then Output ("Scalene")
18                    Else Output ("Isosceles")
19                EndIf
20           EndIf
21   Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

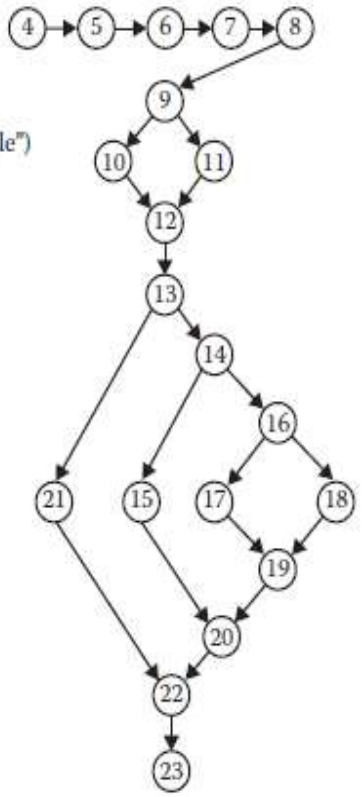



**Figure 8.2 Program graph of triangle program.**

- Nodes 4 through 8 are a sequence, nodes 9 through 12 are an if–then–else construct, and nodes 13 through 22 are nested if–then–else constructs. Nodes 4 and 23 are the program source and sink nodes, corresponding to the single entry, single-exit criteria. No loops exist, so this is a directed acyclic graph.
- The importance of the program graph is that program executions correspondto paths from the source to the sink nodes. Because test cases force the execution of somesuch program path, we now have a very explicit description of the relationship between a test caseand the part of the program it exercises.
- There are detractors of path-based testing. Figure 8.3 is a graph of a simple (but unstructured!)program;Inthis program, five paths lead from node B to node F in the interior

of the loop. If the loop mayhave up to 18 repetitions, some 4.77 trillion distinct program execution paths exist.

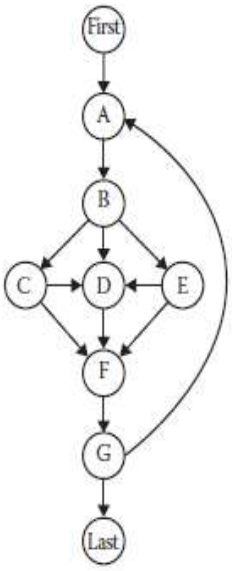



Figure 8.3 Trillions of paths.

## DD-Paths

The best-known form of code-based testing is based on a construct known as a **decision-to decision path** (DD-path) (Miller, 1977).The name refers to a sequence of statements that, in Miller's words, **begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement**. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall.

We will define DD-paths in terms of paths of nodes in a program graph. In graph theory, these paths are called chains, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1.

Notice that the initial node is 2-connected to every other node in the chain, and no instances of 1- or 3-connected nodes occur, as shown in Figure 8.4. The length (number of edges)of the chain in Figure 8.4 is 6.

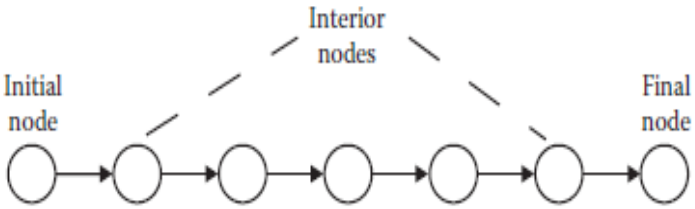



Figure 8.4 Chain of nodes in a directed graph.

A DD-path is a sequence of nodes in a program graph such that

**Case 1: It consists of a single node with indeg = 0.**

**Case 2: It consists of a single node with outdeg = 0.**

**Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.**

**Case 4: It consists of a single node with indeg = 1 and outdeg = 1.**

**Case 5: It is a maximal chain of length ≥ 1.**

- Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-paths.
- Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path.
- Case 4 is needed for "short branches"; it also preserves the one-fragment, one DD-path principle.
- Case 5 is the "normal case," in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to **determine the final node of a normal (nontrivial) chain.**

**Definition**

Given a program written in an imperative language, its DD-path graph is the **directed graph** in which **nodes are** DD-paths of its program graph, and **edges represent** control flow between successor DD-paths.

| Figure 8.2 Nodes | DD-Path | Case of definition |
|---|---|---|
| 4 | First | 1 |
| 5-8 | A | 5 |
| 9 | B | 3 |
| 10 | C | 4 |
| 11 | D | 4 |
| 12 | E | 3 |
| 13 | F | 3 |
| 14 | H | 3 |
| 15 | I | 4 |
| 16 | J | 3 |
| 17 | K | 4 |
| 18 | L | 4 |
| 19 | M | 3 |
| 20 | N | 3 |
| 21 | G | 4 |
| 22 | O | 3 |
| 23 | Last | 2 |

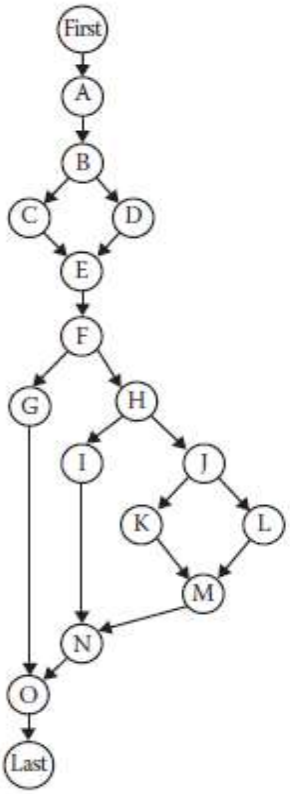



**Figure 8.5 DD-path graph for triangle program.**

- Node 4 is a case 1 DD-path; we will call it "first." Similarly, node 23 is a case 2 DD-path, and we will call it "last."
- Nodes 5 through 8 are case 5 DD-paths. We know that node 8 is the last node in this DD-path because it is the last node that preserves the 2-connectedness property of the chain. If we go beyond node 8 to include node 9, we violate the indegree = outdegree = 1 criterion of a chain.
- If we stop at node 7, we violate the "maximal" criterion. Nodes 10, 11, 15, 17, 18, and 21 are case 4 DD-paths. Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are case 3 DD-paths. Finally, node 23 is a case 2 DD-path. All this is summarized in Figure 8.5.

## Test Coverage Metrics

Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

## Program Graph–Based Coverage Metrics

Given a program graph, we can define the following set of test coverage metrics. We will use them to relate to other published sets of coverage metrics.

**Definition-1**

Given a set of test cases for a program, they constitute **node coverage** if, when executed on the program, **every node in the program graph is traversed**. Denote this level of coverage as Gnode, where the G stands for program graph.

**Definition-2**

Given a set of test cases for a program, they constitute **edge coverage** if, when executed on the program, every **edge in the program graph is traversed.** Denote this level of coverage as Gedge.

**Definition-3**

Given a set of test cases for a program, they constitute **chain coverage** if, when executed on the program, **every chain of length greater than or equal to 2 in the program graph is traversed**. Denote this level of coverage as Gchain.

**Definition-4**

Given a set of test cases for a program, they constitute **path coverage** if, when executed on the program, every **path from the source node to the sink node in the program graph is traversed.** Denote this level of coverage as Gpath.

## E.F. Miller's Coverage Metrics

Several widely accepted test coverage metrics are used; most of those in Table 8.1 are due to the early work of Miller (1977).

**Table 8.1 Miller's Test Coverage Metrics**

| *Metric* | *Description of Coverage* |
|---|---|
| $C_0$ | Every statement |
| $C_1$ | Every DD-path |
| $C_{1p}$ | Every predicate to each outcome |
| $C_2$ | $C_1$ coverage + loop coverage |
| $C_d$ | $C_1$ coverage + every dependent pair of DD-paths |
| $C_{MCC}$ | Multiple condition coverage |
| $C_{ik}$ | Every program path that contains up to $k$ repetitions of a loop (usually $k = 2$) |
| $C_{stat}$ | "Statistically significant" fraction of paths |
| $C_\infty$ | All possible execution paths |

**Statement Testing**

Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C0 metric is subsumed by our Gnode metric.If **some statements have not been**executed by the set of test cases, there is clearly a **severe gap** in the test coverage. Although less adequate than DD-path coverage, the statement coverage metric (C0) is still widely accepted.

**DD-Path Testing**

When every DD-path is traversed (the *C*1 metric), we know that each predicate outcome hasbeen executed; this amounts to traversing every edge in the DD-path graph (or program graph).Therefore, the *C*1 metric is exactly our *G*chainmetric. For if–then and if–then–else statements, this means that **both the true and the false branches are covered** (*C*1p coverage).

**Simple Loop Coverage**

The C2 metric requires DD-path coverage (the C1 metric) plus loop testing.The simple view of loop testing is that every loop involves a decision, and we need to test bothoutcomes of the decision**: one is to traverse the loop, and the other is to exit (or not enter) the loop.**Notice that this is equivalent to the Gedge test coverage.

**Complex Loop Coverage**

Miller's Cik metric extends the loop coverage metric to include full paths from source to sink nodes that contain loops.

**Concatenated loops** are simply a sequence of disjoint loops, while **nested loops** are such that one is contained inside another. **Knotted** (Beizer calls them "horrible") loops cannot occur when the structured programming precepts are followed, but they can occur in languages like Java with try/catch.

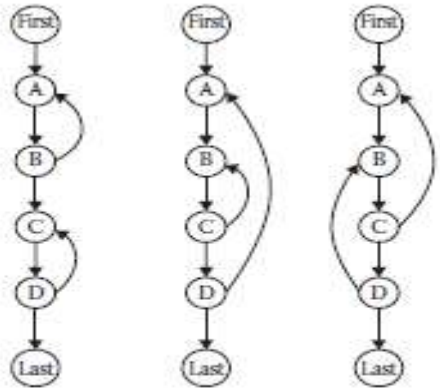



Figure 8.6 Concatenated, nested, and knotted loops.

**Multiple Condition Coverage**

Miller's CMCC metric addresses the question of testing decisions made by compound conditions.

- One possibility is to make a **decision table**; a compound condition of three simpleconditions will have eight rules (see Table 8.2), yielding eight test cases.
- Another possibility is toreprogram compound predicates into nested simple if–then–else logic, which will result in moreDD-paths to cover.

**"Statistically Significant" Coverage**

The Cstat metric is awkward—what constitutes a statistically significant set of full program paths?Maybe this refers to a comfort level on the part of the customer/user.

**All Possible Paths Coverage**

The subscript in Miller's C∞ metric says it all—this can be enormous for programs with loops

## A Closer Look at Compound Conditions

**Boolean Expression (per Chilenski)**

"A Boolean expression evaluates to one of two possible (Boolean) outcomes traditionally known **as False and True."**A Boolean expression may be a simple Boolean variable, or a compound expression containing one or more Boolean operators. Chilenski clarifies Boolean operators into four categories:

| *Operator Type* | *Boolean Operators* |
|---|---|
| Unary (single operand) | NOT(–), |
| Binary (two operands) | AND(∧), OR(∨), XOR(⊕) |
| Short circuit operators | AND (AND–THEN), OR (OR–ELSE) |
| Relational operators | =, ≠, <, ≤, >, ≥ |

In mathematical logic, Boolean expressions are known as logical expressions, where a logical expression can be

1. A simple proposition that contains no logical connective
2. A compound proposition that contains at least one logical connective

**Condition (per Chilenski)**

"A condition is an operand of a Boolean operator. Generally this refers to the lowest level conditions, which are normally the **leaves of an expression tree**. Note that a condition is a Boolean (sub)expression."

In mathematical logic, Chilenski's conditions are known **as simple, or atomic, propositions**. Propositions can be simple or compound, where a compound proposition contains at least one logical connective.

**Coupled Conditions (per Chilenski):** Two (or more) conditions are coupled if changing one also changes the other(s).When conditions are coupled, **it may not be possible to vary individual conditions, because the coupled condition(s) might also change.**

Chelinski notes that conditions can be strongly or weakly coupled.

- In a **strongly coupled pair**, changing one condition always changes the other.
- In a **weakly coupled triplet**, changing one condition may change one other coupled condition, but not the third one.

Chelinski offers these examples:

In (((x = 0) AND A) OR ((x ≠ 0) AND B)), the conditions (x = 0) and (x ≠ 0) are strongly coupled.
In ((x = 1) OR (x = 2) OR (x = 3)), the three conditions are weakly coupled.

**Masking Conditions (per Chilenski)**

"The process masking conditions involves of setting the one operand of an operator to a value such that changing the other operand of that operator does not change the value of the operator.

1. For an AND operator, masking of one operand can be achieved by holding the other operand False.

   (X AND False = False AND X = False no matter what the value of X is.)

2. For an OR operator, masking of one operand can be achieved by holding the other operand True.

   (X OR True = True OR X = True no matter what the value of X is.)."

**Modified Condition Decision Coverage**

MCDC has three variations: Masking MCDC, Unique-Cause MCDC, and Unique-Cause + Masking MCDC.

**Definition**

MCDC requires

1. Every statement must be executed at least once.

2. Every program entry point and exit point must be invoked at least once.
3. All possible outcomes of every control statement are taken at least once.
4. Every nonconstant Boolean expression has been evaluated to both true and false outcomes.
5. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes.
6. Every nonconstant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

**Definition (per Chilenski)**

**"Unique-Cause MCDC** [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions."

**Definition (per Chilenski)**

"**Unique-Cause + Masking MCDC** [requires] a unique cause (toggle a single condition and change the expression result) for all possible **(uncoupled)** conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only, i.e., all other (uncoupled) conditions will remain fixed."

**Definition (per Chilenski)**

"**Masking MCDC** allows masking for all conditions, **coupled and uncoupled** (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of **strongly coupled** conditions, masking [is allowed] for that condition only (i.e., all other (uncoupled) conditions will remain fixed)."

## ***Examples***

Condition with Two Simple Conditions

Consider the program fragment in Figure 8.7. It is deceptively simple, with a cyclomatic complexity of 2.

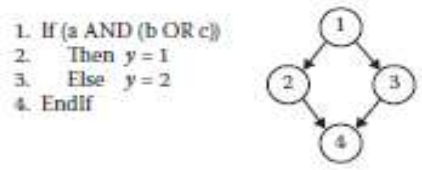



**Figure 8.7 Compound condition and its program graph.**

**Table 8.2 Decision Table for Example Program Fragment**

| *Conditions* | *Rule 1* | *Rule 2* | *Rule 3* | *Rule 4* | *Rule 5* | *Rule 6* | *Rule 7* | *Rule 8* |
|---|---|---|---|---|---|---|---|---|
| a | T | T | T | T | F | F | F | F |
| b | T | T | F | F | T | T | F | F |
| c | T | F | T | F | T | F | T | F |
| a AND (b OR c) | True | True | True | False | False | False | False | False |
| **Actions** | | | | | | | | |
| $y = 1$ | x | x | x | — | — | — | — | — |
| $y = 2$ | — | — | — | x | x | x | x | x |

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 5 toggle condition a; rules 2 and 4 toggle condition b; and rules 3 and 4 toggle condition c.

In the Chelinski (2001) paper (p. 9), it happens that the Boolean expression used **is**

**(a AND (b OR c))**

In its expanded form**, (a AND b) OR (a AND c),** the Boolean variable a cannot be subjected to unique cause MCDC testing because it appears in both AND expressions.

**Compound Condition from NextDate**

In our continuing NextDate problem, suppose we have some code checking for valid inputs of the day, month, and year variables. A code fragment for this and its program graph are in Figure 8.8.

Table 8.3 is a decision table for the NextDate code fragment. Since the day, month, and year variables are all independent, each can be either true or false. The cyclomatic complexity of the program graph in Figure 8.8 is 5.

```
1 NextDate Fragment
2 Dim day, month, year As Integer
3 Dim dayOK, monthOK, yearOK As Boolean
4 Do
5     Input(day, month, year)
6     If 0 < day < 32
7         Then dayOK = True
8         Else dayOK = False
9     EndIf
10    If 0 < month < 13
11        Then monthOK = True
12        Else  monthOK = False
13    EndIf
14    If 1811 < year < 2013
15        Then yearOK = True
16        Else  yearOK = False
17    EndIf
18 Until (dayOK AND monthOK AND yearOK)
19 End Fragment
```

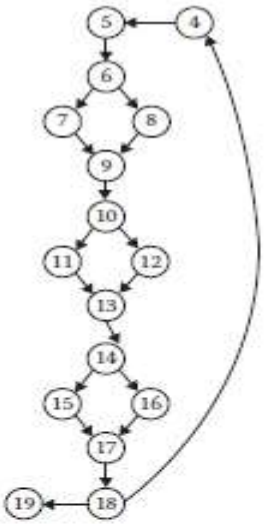



**Figure 8.8 NextDate fragment and its program graph.**

**Table 8.3 Decision Table for NextDate Fragment**

| *Conditions* | *Rule 1* | *Rule 2* | *Rule 3* | *Rule 4* | *Rule 5* | *Rule 6* | *Rule 7* | *Rule 8* |
|---|---|---|---|---|---|---|---|---|
| dayOK | T | T | T | T | F | F | F | F |
| monthOK | T | T | F | F | T | T | F | F |
| YearOK | T | F | T | F | T | F | T | F |
| The Until condition | True | False | False | False | False | False | False | False |
| **Actions** | | | | | | | | |
| Leave the loop | x | — | — | — | — | — | — | — |
| Repeat the loop | — | x | x | x | x | x | x | x |

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rule 1 and any one of rules 2–8 provide decision coverage.

Multiple condition coverage requires exercising a set of rules such that each condition is evaluated to both True and False. The eight test cases corresponding to all eight rules are necessary to provide decision coverage.To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 2 toggle condition yearOK; rules 1 and 3 toggle condition monthOK, and rules 1 and 5 toggle condition dayOK.

**Compound Condition from the Triangle Program**

This example is included to show important differences between it and the first two examples. The code fragment in Figure 8.9 is the part of the triangle program that checks to see if the values of sides a, b, and c constitute a triangle.

```
1. If (a < b + c) AND (a < b + c) AND (a < b + c)
2.     Then IsA Triangle = True
3.     Else IsA Triangle = False
4. EndIf
```

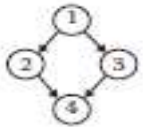



**Figure 8.9 Triangle program fragment and its program graph.**

The dependence among a, b, and c is the cause of the four impossible rules in the decision table for the fragment in Table 8.4; this is proved next.

**Table 8.4 Decision Table for Triangle Program Fragment**

| *Conditions* | *Rule 1* | *Rule 2* | *Rule 3* | *Rule 4* | *Rule 5* | *Rule 6* | *Rule 7* | *Rule 8* |
|---|---|---|---|---|---|---|---|---|
| (a < b + c) | T | T | T | T | F | F | F | F |
| (b < a + c) | T | T | F | F | T | T | F | F |
| (c < a + b) | T | F | T | F | T | F | T | F |
| IsATriangle = True | x | — | — | — | — | — | — | — |
| IsATriangle = False | — | x | x | — | x | — | — | — |
| Impossible | — | — | — | x | | x | x | x |

This code fragment avoids the numerically impossible combinations of a, b, and c. There are four distinct paths through its program graph, and these correspond to rules 1, 2, 3, and 5 in the decision table.

```
1.1    If (a < b + c)
1.2          Then If (b < a + c)
1.3                Then If (c < a + b)
2                        Then IsATriangle = True
3.1                      Else IsATriangle = False
3.2                End If
3.3              Else IsATriangle = False
3.4           End If
3.5         Else IsATriangle = False
4      EndIf
```

## Basis Path Testing

Mathematicians usually define a basis in terms of a structure called a "vector space," which is a set of elements (called vectors) as well as operations that correspond to multiplication and addition defined for the vectors.

## McCabe's Basis Path Method

Figure 8.10 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program.The program does have a single entry (A) and a single exit (G). McCabe based his view of testing on a major result from graph theory, which states that the **cyclomatic number** (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph.

We can always create a **strongly connected graph** by adding an edge from the (every) sink node to the (every) source node.The right side of Figure 8.10 shows the result of doing this; it also contains edge labels that are used in the discussion that follows. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 8.10 is

$$V(G) = e - n + 2p$$
$$= 10 - 7 + 2(1) = 5$$

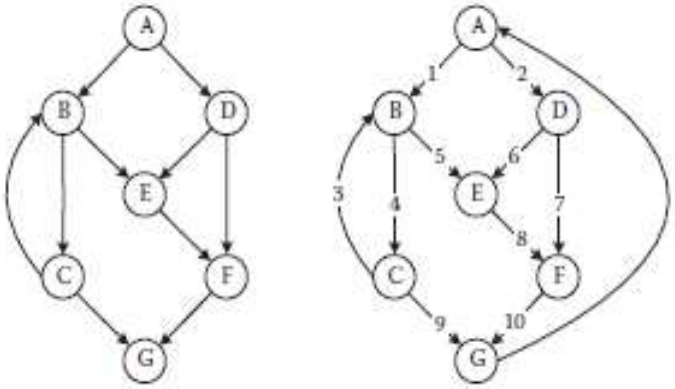



Figure 8.10 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 8.10 is

$$V(G) = e - n + p$$
$$= 11 - 7 + 1 = 5$$

The cyclomatic complexity of the strongly connected graph in Figure 8.10 is 5; thus, there arefive linearly independent circuits. If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here, we identify paths as sequences of nodes:

p1: A, B, C, G
p2: A, B, C, B, C, G
p3: A, B, E, F, G
p4: A, D, E, F, G
p5: A, D, F, G

Table 8.5 shows the edges traversed by each path, and also the number of times an edge is traversed. We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication:

**Table 8.5 Path/Edge Traversal**

| *Path/Edges Traversed* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|---|
| p1: A, B, C, G | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| p2: A, B, C, B, C, G | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| p3: A, B, E, F, G | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| p4: A, D, E, F, G | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| p5: A, D, F, G | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| ex1: A, B, C, B, E, F, G | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| ex2: A, B, C, B, C, B, C, G | 1 | 0 | 2 | 3 | 0 | 0 | 0 | 0 | 1 | 0 |

- Path addition is simply one path followed by another path,
- Multiplication corresponds to repetitions of a path.

- His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum p2 + p3.
- The path A, B, C, B, C, B, C, G is the linear combination2p2 – p1.

McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths.

- The method begins with the selection of a baseline path, which should correspond to some "normal case" program execution.
- McCabe advises choosing a path with as many decision nodes as possible.
- Next, the baseline path is retraced, and in turn each decision is "flipped"; that is, when a node of outdegree ≥ 2 is reached, a different edge must be taken.
- Here we follow McCabe's example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline.

## Observations on McCabe's Basis Path Method

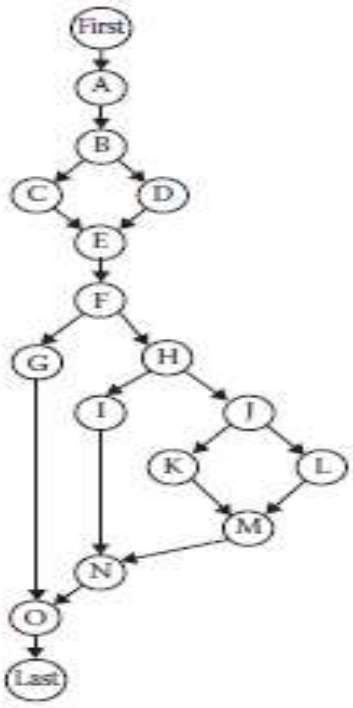



**Table 8.6 Basis Paths in Figure 8.5**

| | | |
|---|---|---|
| Original | p1: A–B–C–E–F–H–J–K–M–N–O–Last | Scalene |
| Flip p1 at B | p2: A–B–D–E–F–H–J–K–M–N–O–Last | Infeasible |
| Flip p1 at F | p3: A–B–C–E–F–G–O–Last | Infeasible |
| Flip p1 at H | p4: A–B–C–E–F–H–I–N–O–Last | Equilateral |
| Flip p1 at J | p5: A–B–C–E–F–H–J–L–M–N–O–Last | Isosceles |

Notice that this set of basis paths is distinct from the one in Table 8.6: this isnot problematic because a unique basis is not required.

Time for a reality check: if you follow paths p2 and p3, you find that they are both infeasible. Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G. Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively. Notice that we do not have a basis path for the NotATriangle case.

Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:

- If node C is traversed, then we must traverse node H.
- If node D is traversed, then we must traverse node G.

| | |
|---|---|
| p1: A–B–C–E–F–H–J–K–M–N–O–Last | Scalene |
| p6: A–B–D–E–F–G–O–Last | Not a triangle |
| p4: A–B–C–E–F–H–I–N–O–Last | Equilateral |
| p5: A–B–C–E–F–H–J–L–M–N–O–Last | Isosceles |

The triangle problem is atypical in that no loops occur. The program has only eight topologicallypossible paths; and of these, only the four basis paths listed above are feasible. Thus, for this special case, we arrive at the same test cases as we did with special value testing and output range testing.

## Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing. In this section, we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing.

we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing.we condense around the structured programming constructs, which are repeated as Figure 8.11.

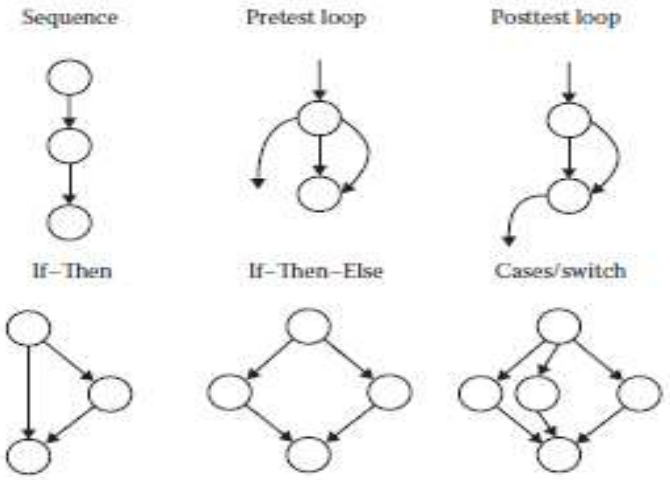



**Figure 8.11 Structured programming constructs.**

McCabe (1976) went on to find elemental "unstructures" that violate the precepts of structured programming. These are shown in Figure 8.13. Each of these violations contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs;

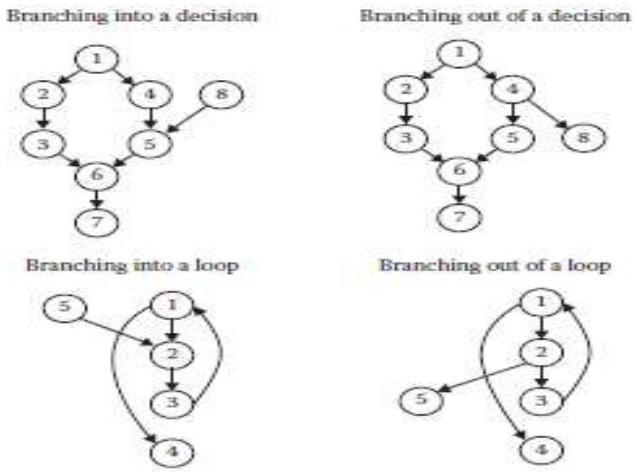



**Figure 8.13 Violations of structured programming constructs.**

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into **a single node, and repeat until no more structured programming constructs can be found.** This process is followed in Figure 8.12, which starts with the DD-path graph of the pseudocode triangle program. The if–then–else construct involving nodes B, C, D, and E is condensed into node

a, and then the three if–then constructs are condensed onto nodes b, c, and d. The remaining if–then–else (which corresponds to the IF IsATriangle statement) is condensed into node e, **resulting in a condensed graph with cyclomatic complexity V(G) = 1**. In general, when a program is well structured, it can always be **reduced to a graph with one path.**

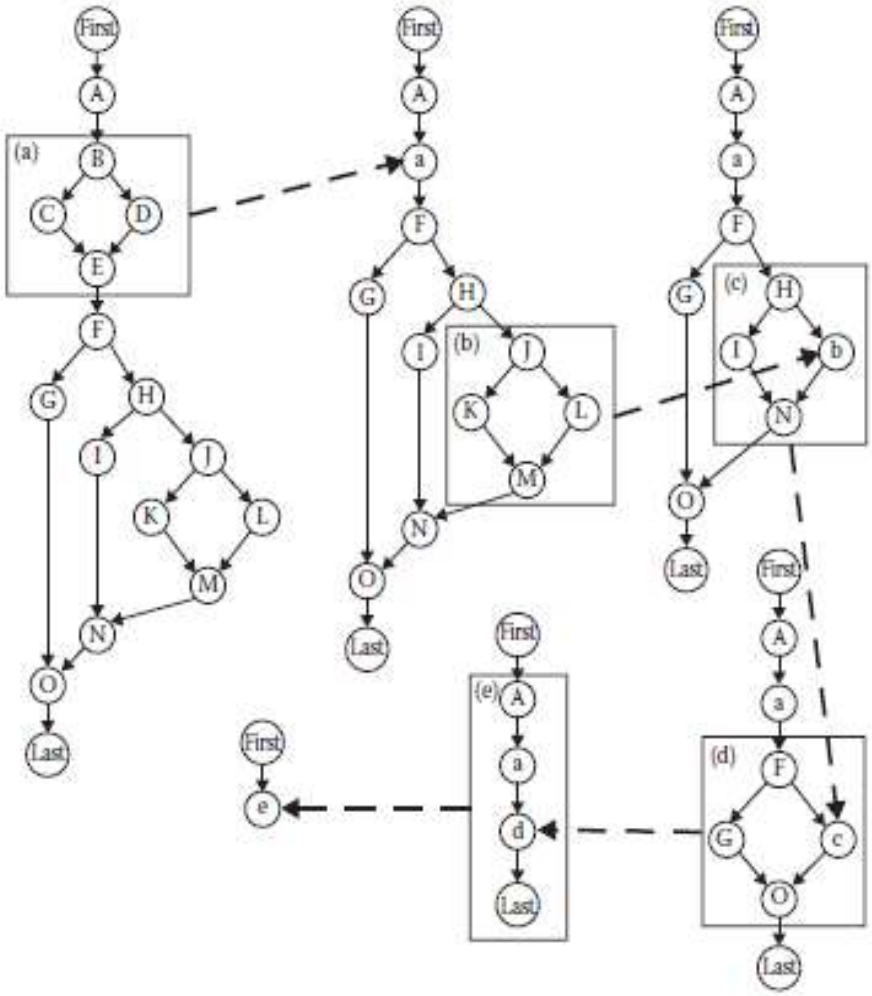



Figure 8.12 Condensing with respect to structured programming constructs.

## Guidelines and Observations

McCabe (1982) was partly right when he observed, "It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases." He

was referring to the DD-path coverage metric and his basis path heuristic based on cyclomaticcomplexity metric. Basis path testing therefore gives us a lower boundary on how much testing is necessary.

Path-based testing also provides us with a set of metrics that act as crosschecks onspecification-based testing. We can use these metrics to resolve the gaps and redundancies question.

# MODULE 3

# Data Flow Testing(Chapter 2)

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).

Early data flow analyses often centered on a set of faults that are now known as define/reference anomalies:

- A variable that is defined but never used (referenced)
- A variable that is used before it is defined
- A variable that is defined twice before it is used

Each of these anomalies can be recognized from the concordance of a program.

## Define/Use Testing

The following definitions refer to a program P that has a program graph G(P) and a set of program variables V. The program graph G(P) is constructed, with statement fragments as nodes and edges that represent node sequences. G(P) has a single-entry node and a single-exit node, Paths, subpaths, and cycles.

**Definition**

Node n ∈G(P) is a defining node of the variable v ∈ V, written as DEF(v, n), if and only if **the value of variable v is defined** as the statement fragment corresponding to node n.

Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the **contents of the memory location(s) associated with the variables are changed**.

**Definition**

Node n ∈G(P) is a usage node of the variable v ∈ V, written as USE(v, n), if and only if **the value of the variable v is used** as the statement fragment corresponding to node n.

Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the **contents of the memory location(s) associated with the variables remain unchanged.**

**Definition**

A usage node USE(v, n) is **a predicate use** (denoted as P-use) if and only if the statement n is a predicate statement; otherwise, USE(v, n) is a **computation use** (denoted C-use). The nodes corresponding to **predicate uses** always have **an outdegree ≥ 2,** and nodes corresponding to **computation uses** always have an **outdegree ≤ 1.**

**Definition**

A definition/use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some v ∈ V, **there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the path.**

**Definition**

A definition-clear path with respect to a variable v (denoted dc-path) is a definition/use path in PATHS(P) with initial and final nodes DEF(v, m) and USE(v, n) **such that no other node in the path is a defining node of v.**

## ***Example***

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 9.1. 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 9.2 shows the decision-to-decision path (DD-path) graph of the program graph in Figure 9.1. More compression exists in this DD-path graph because of the increased computation in the commission problem. Table 9.1 details the statement fragments associated with DD-paths.

Table 9.2 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 9.1 to identify various definition/ use and definition-clear paths.

Tables 9.3 and 9.4 present some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 9.1). The third column in Table 9.3 indicates whether the du-paths are definition clear.

```
1  Program Commission (INPUT,OUTPUT)
2  Dim locks, stocks, barrels As Integer
3  Dim lockPrice, stockPrice, barrelPrice As Real
4  Dim totalLocks, totalStocks, totalBarrels As Integer
5  Dim lockSales, stockSales, barrelSales As Real
6  Dim sales, commission As Real
7  lockPrice = 45.0
8  stockPrice = 30.0
9  barrelPrice = 25.0
10 totalBarrels = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
14 While NOT(locks = –1)  "locks = –1 signals end of data
15  Input(stocks, barrels)
16  totalLocks = totalLocks + locks
17  totalStocks = totalStocks + stocks
18  totalBarrels = totalBarrels + barrels
19  Input(locks)
20 EndWhile
21 Output("Locks sold:," totalLocks)
22 Output("Stocks sold:," totalStocks)
23 Output("Barrels sold:," totalBarrels)
24 lockSales = lockPrice*totalLocks
25 stockSales = stockPrice*totalStocks
26 barrelsSales = barrelPrice * totalBarrels
27 sales = lockSales + stockSales + barrelSales
28 Output("Total sales: ", sales)
29 If (sales > 1800.0)
30  Then
31   commission = 0.10 * 1000.0
32   commission = commission + 0.15 * 800.0
33   commission = commission + 0.20*(sales–1800.0)
34  Else If (sales > 1000.0)
35   Then
36   commission = 0.10 * 1000.0
37   commission = commission + 0.15*(sales–1000.0)
38   Else
39   commission = 0.10 * sales
40  EndIf
41 EndIf
42 Output("Commission is $", commission)
43 End Commission
```

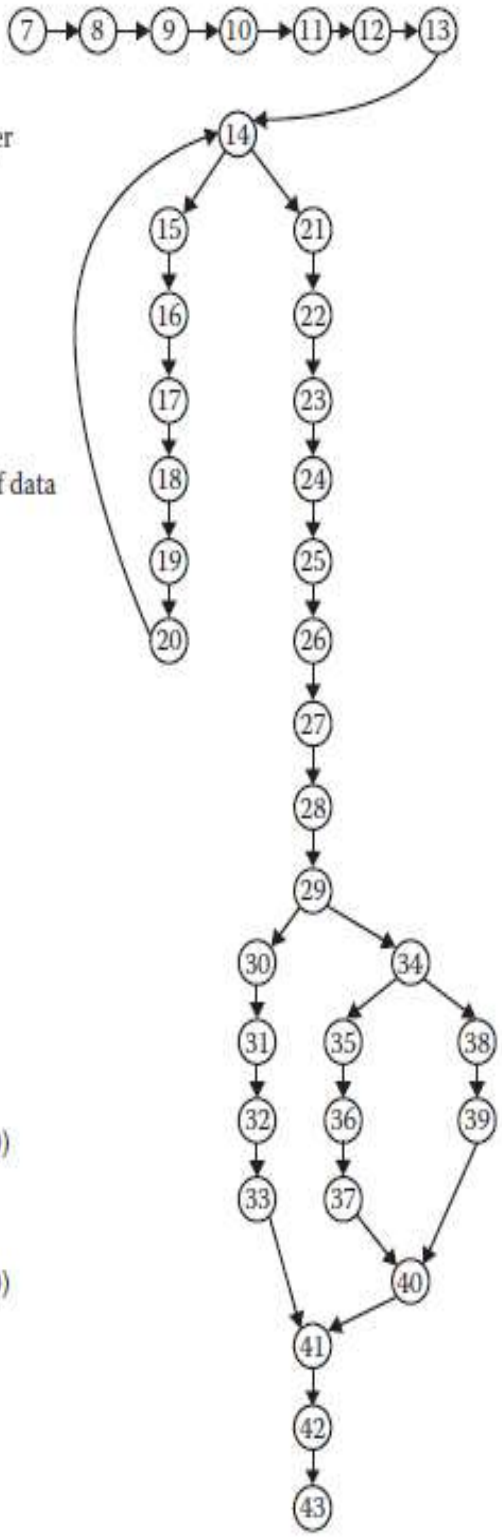



**Figure 9.1 Commission problem and its program graph.**

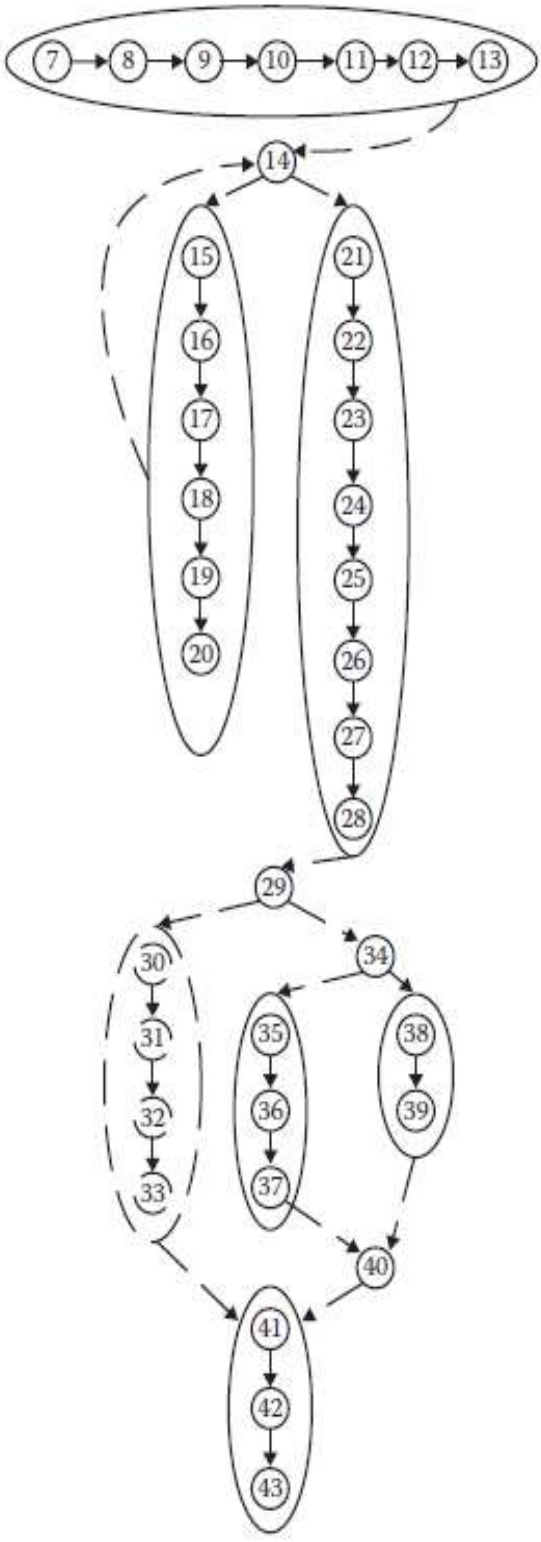



Figure 9.2 DD-path graph of commission problem pseudocode (in Figure 9.1).

**Du-paths for Stocks**

First, let us look at a simple path: the du-path for the variable stocks. We have DEF(stocks, 15) and USE(stocks, 17), so the path <15, 17> is a du-path with respect to stocks. **No other defining nodes are used for stocks; therefore, this path is also definition clear.**

**Du-paths for Locks**

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks, 13), DEF(locks, 19), USE(locks, 14), and USE(locks, 16). These yield four du-paths; they are shown in Figure 9.3.

- p1 = <13, 14>
- p2 = <13, 14, 15, 16>
- p3 = <19, 20, 14>
- p4 = <19, 20, 14, 15, 16>

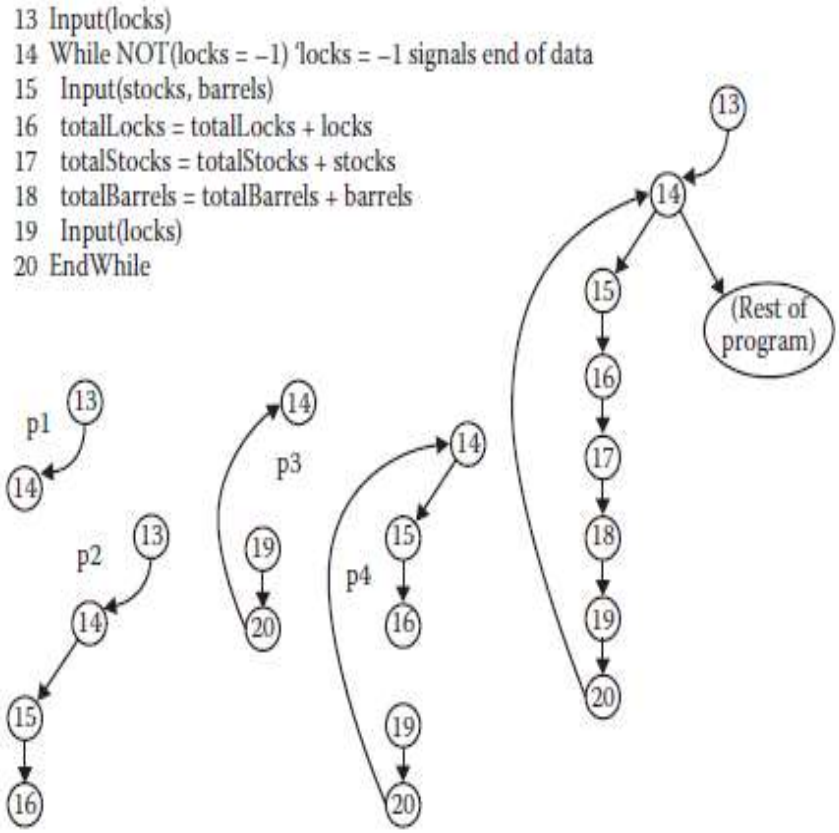



Figure 9.3 Du-paths for locks.

**Table 9.1 DD-paths in Figure 9.1**

| *DD-path* | *Nodes* |
|---|---|
| A | 7, 8, 9, 10, 11, 12, 13 |
| B | 14 |
| C | 15, 16, 17, 18, 19, 20 |
| D | 21, 22, 23, 24, 25, 26, 27, 28 |
| E | 29 |
| F | 30, 31, 32, 33 |
| G | 34 |
| H | 35, 36, 37 |
| I | 38 |
| J | 39 |
| K | 40 |
| L | 41, 42 |

**Table 9.2 Define/Use Nodes for Variables in Commission Problem**

| *Variable* | *Defined at Node* | *Used at Node* |
|---|---|---|
| lockPrice | 7 | 24 |
| stockPrice | 8 | 25 |
| barrelPrice | 9 | 26 |
| totalLocks | 10, 16 | 16, 21, 24 |
| totalStocks | 11, 17 | 17, 22, 25 |
| totalBarrels | 12, 18 | 18, 23, 26 |
| Locks | 13, 19 | 14, 16 |
| Stocks | 15 | 17 |
| Barrels | 15 | 18 |
| lockSales | 24 | 27 |
| stockSales | 25 | 27 |
| barrelSales | 26 | 27 |
| Sales | 27 | 28, 29, 33, 34, 37, 38 |
| Commission | 31, 32, 33, 36, 37, 38 | 32, 33, 37, 41 |

**Table 9.3 Selected Define/Use Paths**

| *Variable* | *Path (Beginning, End) Nodes* | *Definition Clear?* |
|---|---|---|
| lockPrice | 7, 24 | Yes |
| stockPrice | 8, 25 | Yes |
| barrelPrice | 9, 26 | Yes |
| totalStocks | 11, 17 | Yes |
| totalStocks | 11, 22 | No |
| totalStocks | 11, 25 | No |
| totalStocks | 17, 17 | Yes |
| totalStocks | 17, 22 | No |
| totalStocks | 17, 25 | No |
| Locks | 13, 14 | Yes |
| Locks | 13, 16 | Yes |
| Locks | 19, 14 | Yes |
| Locks | 19, 16 | Yes |
| Sales | 27, 28 | Yes |
| Sales | 27, 29 | Yes |
| Sales | 27, 33 | Yes |
| Sales | 27, 34 | Yes |
| Sales | 27, 37 | Yes |
| Sales | 27, 38 | Yes |

**Du-paths for totalLocks**

The du-paths for totalLocks will lead us to typical test cases for computations. With two definingnodes (DEF(totalLocks, 10) and DEF(totalLocks, 16)) and three usage nodes (USE(totalLocks, 16), USE(totalLocks, 21), USE(totalLocks, 24)), we might expect six du-paths. Let us take a closer look.

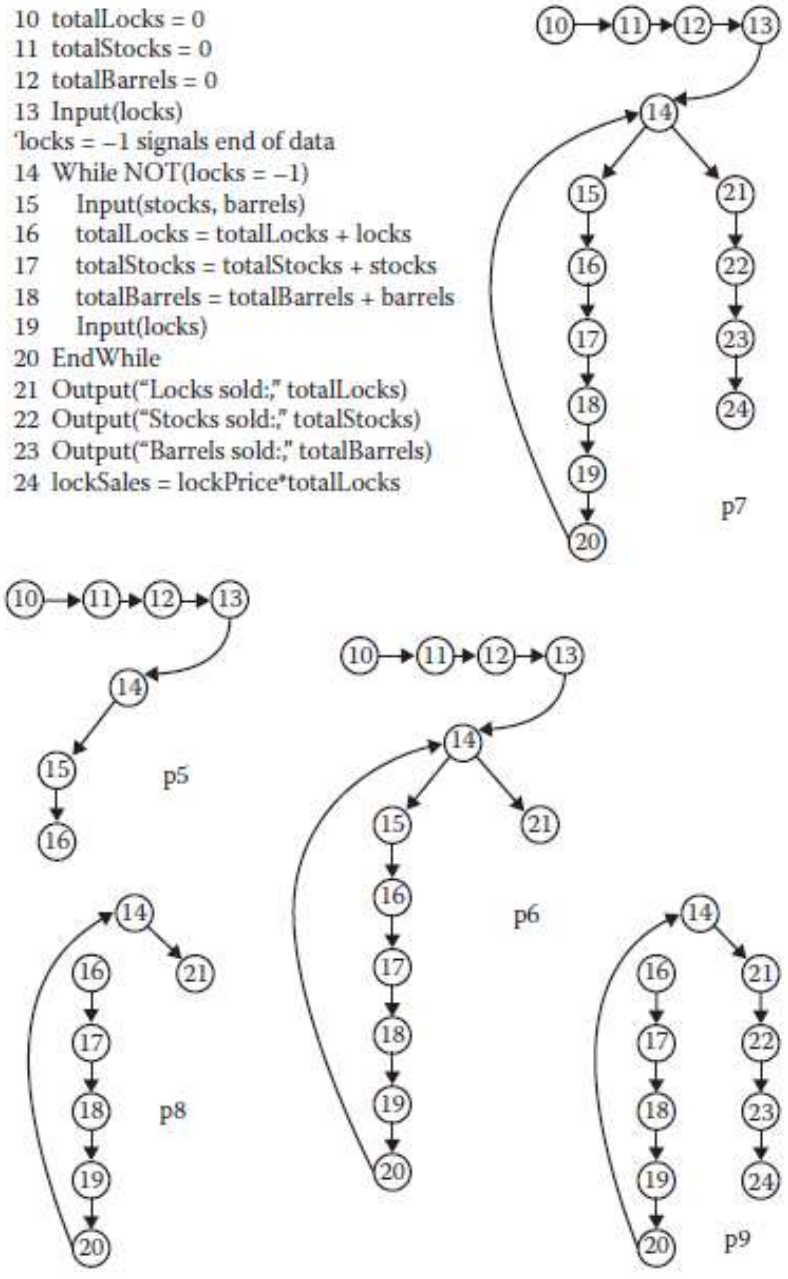



**Figure 9.4 Du-paths for totalLocks.**

## ***Define/Use Test Coverage Metrics***

**Definition**

The set T satisfies the **All-Defs criterion** for the program P if and only if for every variable v ∈ V, T contains definition-clear paths from every defining node of v to a use of v.

**Definition**

The set T satisfies the **All-Uses criterion for the program** P if and only if for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every use of v, and to the successor node of each USE(v, n).

**Definition**

The set T satisfies the **All-P-Uses/Some C-Uses criterion** for the program P if and only if for every variable v ∈ V, T contains definition-clear paths from every defining node of v to every predicate use of v; and **if a definition of v has no P-uses, a definition-clear path leads to at least one computation use.**

**Definition**

The set T satisfies the **All-C-Uses/Some P-Uses criterion** for the program P if and only if for every variable v ∈ V, T contains definition clear paths from every defining node of v to every computation use of v; and **if a definition of v has no C-uses, a definition-clear path leads to at least one predicate use.**

These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable**) All-Paths metric and the generally accepted minimum, All-Edges.**

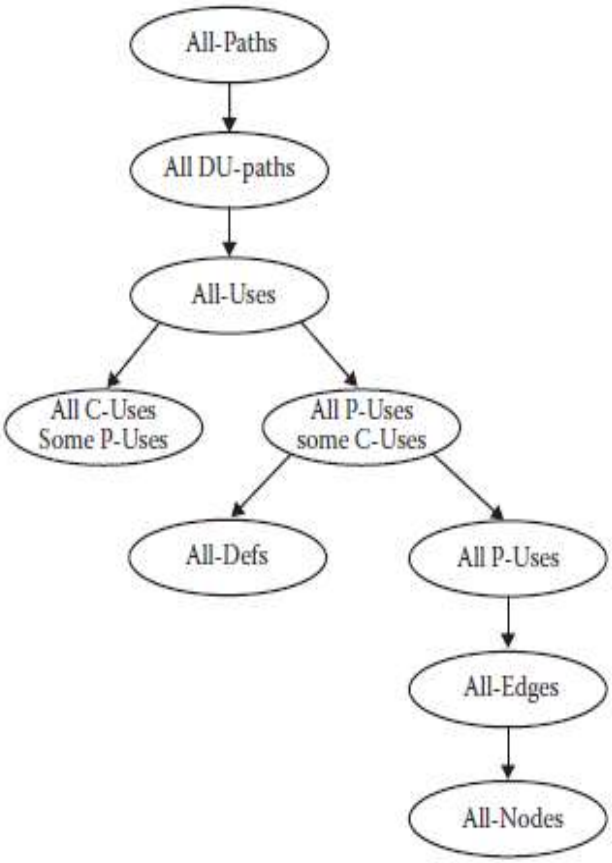



Figure 9.5 Rapps–Weyuker hierarchy of data flow coverage metrics.

# Module 3(Test Execution)

# Chapter 3

## From Test Case Specifications to Test Cases

If the **test case specifications** produced in test design already **include** concrete **input values and expected results**,then producing a complete test case may be as simple as filling a template with those values.

Eg: A more general test case specification (e.g., one that calls for "a sorted sequence, length greater than 2, with items in ascending order with no duplicates") may designate many possible concrete test cases, and it may be desirable to generate just one instance or many.

Test data : <? 12,56,89,04.>
Sorted order:<?A 04,12,56,89.>

Automatic generation of concrete test cases from more abstract test case specifications reduces the impact of small interface changes in the course of development. Corresponding changes to the test suite are still required with each program change, but changes to test case specifications are likely to be smaller and more localized than changes to the concrete test cases.

## Scaffolding

During much of development, only a portion of the **full system is available for testing**. In modern development methodologies, the **partially developed system** is likely to consist of one or more runnable programs and may even be considered a version or prototype of the final system from very early in construction, so it is possible at least to execute each new portion of the software as it is constructed.

**Code developed to facilitate testing is called scaffolding,**Scaffolding may include test drivers (substituting for a main or calling program), test harnesses (substituting for parts of the deployment environment), and stubs (substituting for functionality called or used by the software under test).

The purposes of scaffolding are to provide **controllability** to execute test cases and **observability**to judge the outcome of test execution.

## Generic versus Specific Scaffolding

The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence.

At thevery least one will want to factor out some of the common driver code into reusablemodules. Sometimes it is worthwhile to write more generic test drivers that essentially interpret test case specifications.

At least some level of generic scaffolding support can be used across a fairly wide class of applications. Such support typically includes, in addition to a standard interface for executing a set of test cases, basic support for logging test execution and results. Figure 17.1 illustrates use of generic test scaffolding in the JFlex lexical analyzer generator.

```
1   package JFlex.tests;
2
3   import JFlex.IntCharSet;
4   import JFlex.Interval;
5   import junit.framework.TestCase;
11  ...
12  public class  CharClassesTest extends TestCase {
25  ...
26     public void testAdd1() {
27        IntCharSet set = new IntCharSet(new Interval('a','h'));
28        set.add(new Interval('o','z'));
29        set.add(new Interval('A','Z'));
30        set.add(new Interval('h','o'));
31        assertEquals("{  ['A'-'Z']['a'-'z']  }", set.toString());
32     }
33
34     public void testAdd2() {
35        IntCharSet set = new IntCharSet(new Interval('a','h'));
36        set.add(new Interval('o','z'));
37        set.add(new Interval('A','Z'));
38        set.add(new Interval('i','n'));
39        assertEquals("{  ['A'-'Z']['a'-'z']  }", set.toString());
40     }
99  ...
100 }
```

Figure 17.1: Excerpt of JFlex 1.4.1 source code (a widely used open-source scannergenerator) and accompanying JUnit test cases. JUnit is typical of basic test scaffoldinglibraries, providing support for test execution, logging, and simple result checking

## Test Oracles

It is little use to execute a test suite automatically if execution results must be manually inspected to apply a pass/fail criterion. Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable.Software that applies a pass/fail criterion to a program execution is called a test oracle often shortened to oracle.

Support for comparison-based test oracles is often included in a test harness program or testing framework. A harness typically takes two inputs: (1) the input to the program under test (or can be mechanically transformed to a well-formed input), and (2) the predicted output. Frameworks for writing test cases as program code likewise provide support for comparison-based oracles. The assertEquals method of JUnit, illustrated in Figure 17.1, is a simple example of comparison-based oracle support.

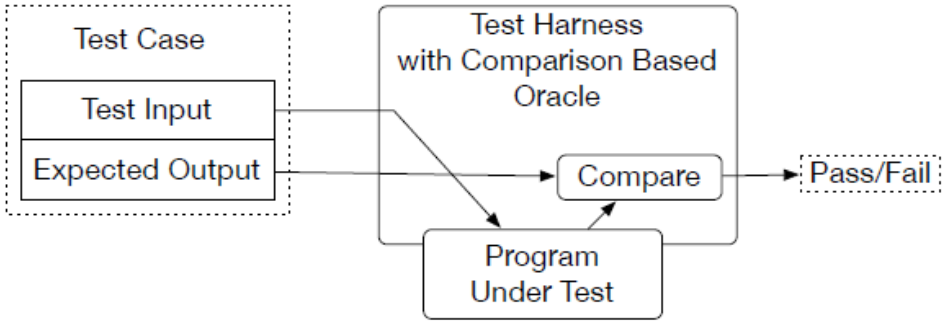



*Figure 17.2: A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.*

## Self-Checks as Oracles

A program or module specification describes all correct program behaviors, so an oracle based on a specification need not be paired with a particular test case. Instead, the oracle can be incorporated into the program under test, so that it checks its own work (see Figure 17.3).

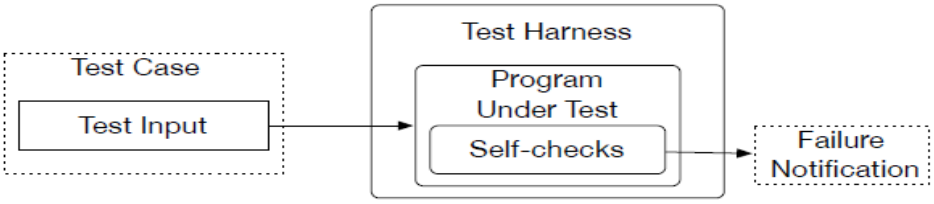



*Figure 17.3: When self-checks are embedded in the program, test cases need not include predicted outputs.*

Self-checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specifications, rather than overall program behavior. Devising program assertions that correspond in a natural way to specifications (formal or informal) poses two main challenges: bridging the gap between concrete execution values and abstractions used in specification, and dealing in a reasonable way with quantification over collections of values.

## Capture and Replay

The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting.

# Module IV (Chapter 1)

# Planning and monitoring the process

## Overview:

Planning involves scheduling activities, allocating resources, and devising observable, unambiguous milestones against which progress and performance can be monitored. Monitoring means answering the question, "How are we doing?"

Quality planning is one aspect of project planning, and quality processes must be closely coordinated with other development processes. Coordination among quality and development tasks may constrain ordering

Formulation of the plan involves risk analysis and contingency planning. Execution of the plan involves monitoring, corrective action, and planning for subsequent releases and projects

## Quality and process:

A software plan involves many intertwined concerns, from schedule to cost to usability and dependability. Despite the intertwining, it is useful to distinguish individual concerns and objectives to lessen the likelihood that they will be neglected, to allocate responsibilities, and to make the overall planning process more manageable

An appropriate quality process follows a form similar to the overall software process in which it is embedded

A general principle, across all software processes, is that the cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults.

The particular verification or validation step at each stage depends on the nature of the intermediate work product and on the anticipated defects

The evolving collection of work products can be viewed as a set of descriptions of different parts and aspects of the software system, at different levels of detail. Portions of the implementation have the useful property of being executable in a conventional sense, and are the traditional subject of testing, but every level of specification and design can be both the subject of verification activities and a source of information for verifying other artifacts. A typical

intermediate artifact - say, a subsystem interface definition or a database schema - will be subject to the following steps:

**Internal consistency check**: Check the artifact for compliance with structuring rules that define "well-formed" artifacts of that type. An important point of leverage is defining the syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations. This is analogous to syntax and strong-typing rules in programming languages, which are not enough to guarantee program correctness but effectively guard against many simple errors.

**External consistency check:** Check the artifact for consistency with related artifacts. Often this means checking for conformance to a "prior" or "higher-level" specification, but consistency checking does not depend on sequential, top-down development - all that is required is that the related information from two or more artifacts be defined precisely enough to support detection of discrepancies

**Generation of correctness conjectures:** Correctness conjectures, which can be test outcomes or other objective criteria, lay the groundwork for external consistency checks of other work products, particularly those that are yet to be developed or revised.

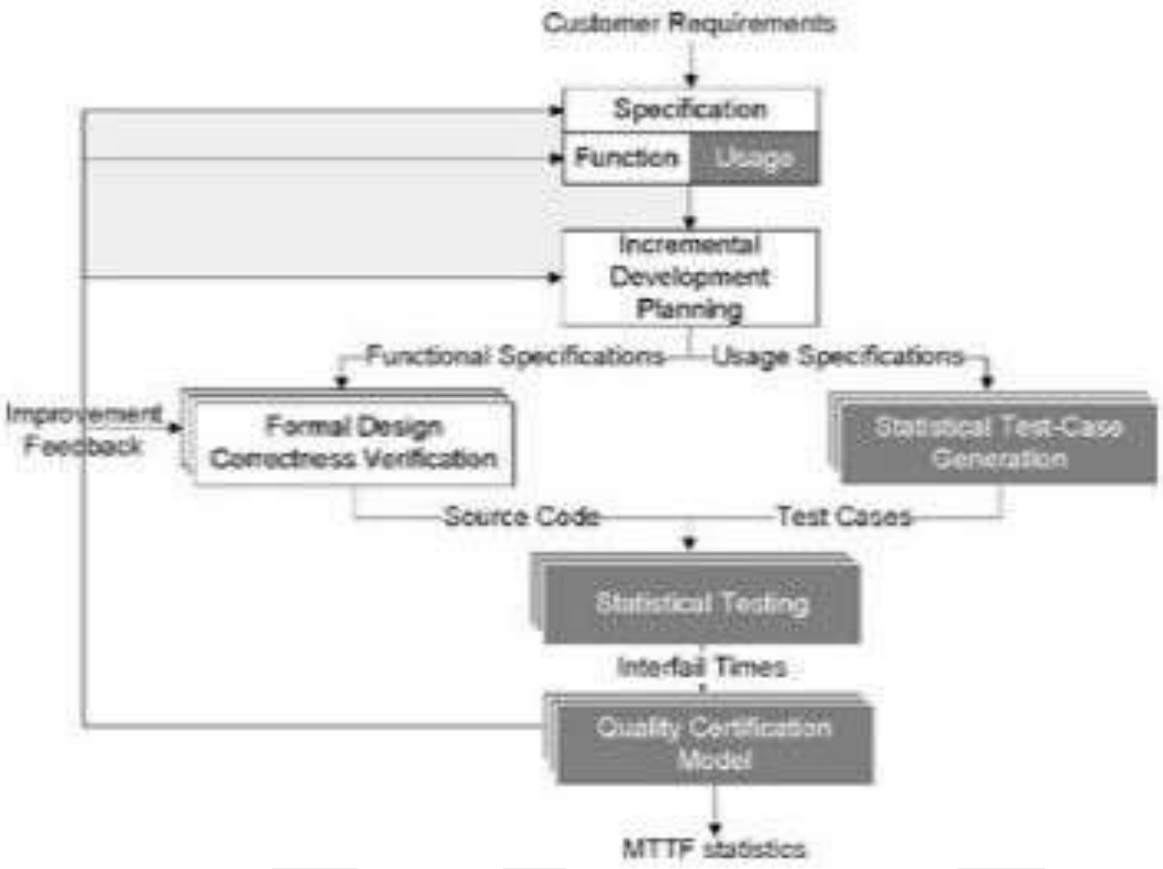
**Test and Analysis Strategies**

A body of explicit knowledge, shared and refined by the group, is more valuable than islands of individual competence. Organizational knowledge in a shared and systematic form is more amenable to improvement and less vulnerable to organizational change, including the loss of key individuals

Capturing the lessons of experience in a consistent and repeatable form is essential for avoiding errors, maintaining consistency of the process, and increasing development efficiency

**Cleanroom**

The Cleanroom process involves two cooperating teams, the development and the quality teams, and five major activities: specification, planning, design and verification, quality certification, and feedback




In the specification activity, the development team defines the required behavior of the system, while the quality team defines usage scenarios that are later used for deriving system test suites. The planning activity identifies incremental development and certification phases

After planning, all activities are iterated to produce incremental releases of the system. Each system increment is fully deployed and certified before the following step. Design and code undergo formal inspection ("Correctness verification") before release. One of the key premises underpinning the Clean room process model is that rigorous design and formal inspection produce "nearly fault-free software.

Another key assumption of the Clean room process model is that usage profiles are sufficiently accurate that statistical testing will provide an accurate measure of quality as perceived by users.[a] Reliability is measured in terms of mean time between failures (MTBF) and is constantly controlled after each release. Failures are reported to the development team for correction, and if reliability falls below an acceptable range, failure data is used for process improvement before the next incremental release.

Test and analysis strategies capture commonalities across projects and provide guidelines for maintaining consistency among quality plans.

The quality strategy is an intellectual asset of an individual organization prescribing a set of

solutions to problems specific to that organization. Among the factors that particularize the strategy are

**Structure and size**

In a smaller organization, or an organization that has devolved responsibility to small, semi-autonomous teams, there is typically less emphasis on formal communication and documents but a greater emphasis on managing and balancing the multiple roles played by each team member

**Overall process**

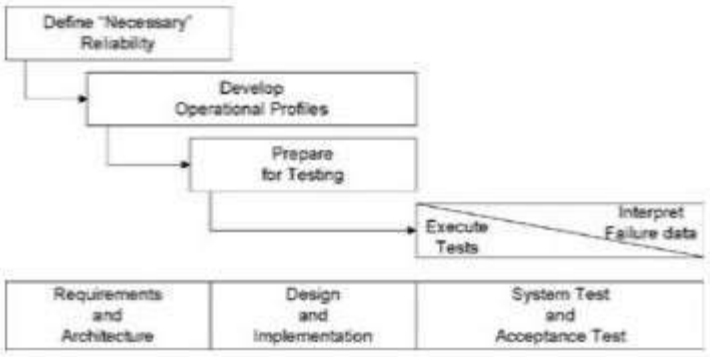
We have already noted the intertwining of quality process with other aspects of an overall software process, and this is of course reflected in the quality strategy. Notations, standard process steps, and even tools can be reflected in the quality strategy to the extent they are consistent from project to project.

**Application domain**

The domain may impose both particular quality objectives (e.g., privacy and security in medical records processing), and in some cases particular steps and documentation required to obtain certification from an external authority.

# SRET

The software reliability engineered testing (SRET) approach, developed at AT&T in the early 1990s, assumes a spiral development process and augments each coil of the spiral with rigorous testing activities. SRET identifies two main types of testing: development testing, used to find and remove faults in software at least partially developed in-house, and certification testing, used to either accept or reject outsourced software.



The five core steps of SRET are:

**Define "Necessary" Reliability** Determine operational models, that is, distinct patterns of system usage that require separate testing, classify failures according to their severity, and engineer the reliability strategy with fault prevention, fault removal, and fault tolerance activities.

**Develop Operational Profiles** Develop both overall profiles that span operational models and operational profiles within single operational models.

**Prepare for Testing** Specify test cases and procedures.

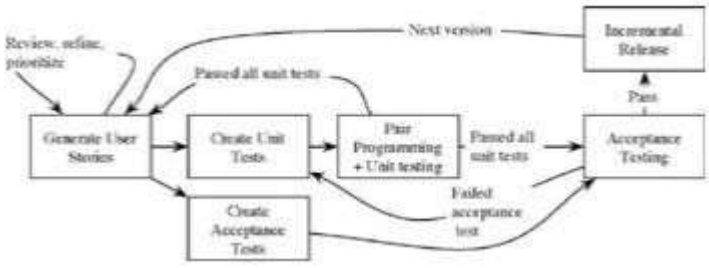
**Execute Tests** executing the programs.

**Interpret Failure Data** Interpretation of failure data depends on the type of testing. In development testing, the goal is to track progress and compare present failure intensities with objectives. In certification testing, the goal is to determine if a software component or system should be accepted or rejected.

# Extreme Programming (XP)

The extreme programming methodology (XP) emphasizes simplicity over generality, global vision and communication over structured organization, frequent changes over big releases, continuous testing and analysis over separation of roles and responsibilities, and continuous feedback over traditional planning.

Customer involvement in an XP project includes requirements analysis (development, refinement, and prioritization of user stories) and acceptance testing of very frequent iterative releases. Planning is based on prioritization of user stories, which are implemented in short iterations. Test cases corresponding to scenarios in user stories serve as partial specifications




Test cases suitable for batch execution are part of the system code base and are implemented prior to the implementation of features they check ("test-first"). Developers work in pairs, incrementally developing and testing a module. Pair programming effectively conflates a review

activity with coding. Each release is checked by running all the tests devised up to that point of development, thus essentially merging unit testing with integration and system testing

## Test and Analysis Plans

An analysis and test plan details the steps to be taken in a particular project. A plan should answer the following questions:

- What quality activities will be carried out?
- What are the dependencies among the quality activities and between quality and development activities?
- What resources are needed and how will they be allocated?
- How will both the process and the evolving product be monitored to maintain an adequate assessment of quality and early warning of quality and schedule problems?

Each of these issues is addressed to some extent in the quality strategy, but must be elaborated and particularized. This is typically the responsibility of a quality manager, who should participate in the initial feasibility study to identify quality goals and estimate the contribution of test and analysis tasks on project cost and schedule

To produce a quality plan that adequately addresses the questions above, the quality manager must identify the items and features to be verified, the resources and activities that are required, the approaches that should be followed, and criteria for evaluating the results

Items and features to be verified circumscribe the target of the quality plan. While there is an obvious correspondence between items to be developed or modified and those to undergo testing, they may differ somewhat in detail. For example, overall evaluation of the user interface may be the purview of a separate human factors group.

Defining quality objectives and process organization in detail requires information that is not all available in the early stages of development. Test items depend on design decisions; detailed approaches to evaluation can be defined only after examining requirements and design specifications; tasks and schedule can be completed only after the design; new risks and contingencies may be introduced by decisions taken during development

After capturing goals as well as possible, the next step in construction of a quality plan is to produce an overall rough list of tasks. The quality strategy and past experience provide a basis for customizing the list to the current project and for scaling tasks appropriately

The manager can start noting dependencies among the quality activities and between them and other activities in the overall project, and exploring arrangements of tasks over time. The main objective at this point is to schedule quality activities so that assessment data are provided continuously throughout the project, without unnecessary delay of other development activities

**Risks:**

While the project plan shows the expected schedule of tasks, the arrangement and ordering of tasks are also driven by risk. The quality plan, like the overall project plan, should include an explicit risk plan that lists major risks and contingencies

A key tactic for controlling the impact of risk in the project schedule is to minimize the likelihood that unexpected delay in one task propagates through the whole schedule and delays project completion

**Critical paths** are chains of activities that must be completed in sequence and that have maximum overall duration. Tasks on the critical path have a high priority for early scheduling, and likewise the tasks on which they depend (which may not themselves be on the critical path) should be scheduled early enough to provide some schedule slack and prevent delay in the inception of the critical tasks

**A critical dependence** occurs when a task on a critical path is scheduled immediately after some other task on the critical path, particularly if the length of the critical path is close to the length of the project. Critical dependence may occur with tasks outside the quality plan part of the overall project plan

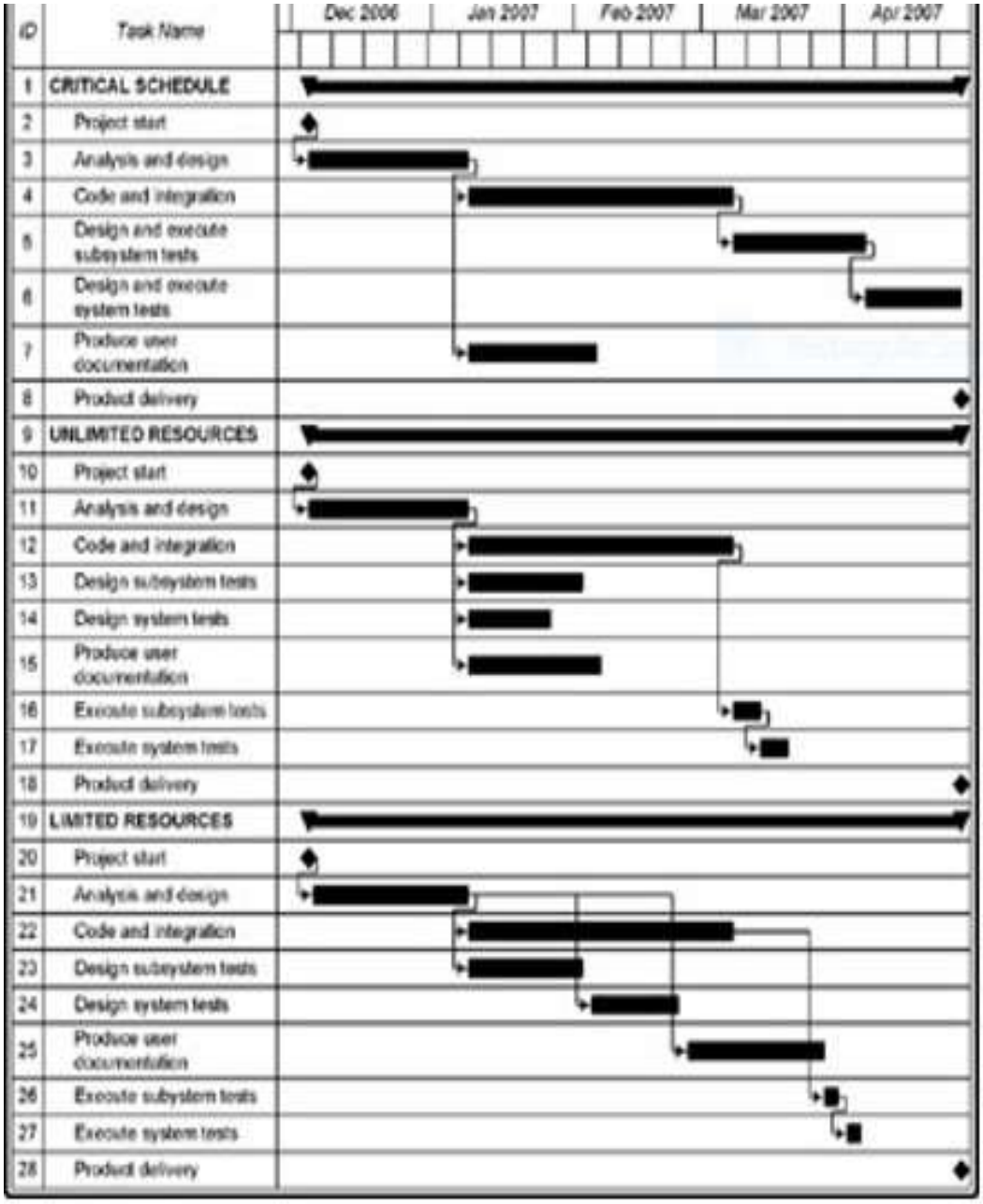
The primary tactic available for reducing the schedule risk of a critical dependence is to decompose a task on the critical path, factoring out subtasks that can be performed earlier

Fig below shows alternative schedules for a simple project that starts at the beginning of January and must be completed by the end of May. In the top schedule, indicated as CRITICAL SCHEDULE, the tasks Analysis and design, Code and Integration, Design and execute subsystem tests, and Design and execute system tests form a critical path that spans the duration of the entire project. A delay in any of the activities will result in late delivery. In this schedule, only the Produce user documentation task does not belong to the critical path, and thus only delays of this task can be tolerated

In the middle schedule, marked as UNLIMITED RESOURCES, the test design and execution activities are separated into distinct tasks. Test design tasks are scheduled early, right after analysis and design, and only test execution is scheduled after Code and integration. In this way the tasks Design subsystem tests and Design system tests are removed from the critical path, which now spans 16 weeks with a tolerance of 5 weeks with respect to the expected termination of the project

The LIMITED RESOURCES schedule at the bottom of Figure 20.1 rearranges tasks to meet resource constraints. In this case we assume that test design and execution, and production of user documentation share the same resources and thus cannot be executed in parallel. We can see that, despite the limited parallelism, decomposing testing activities and scheduling test design

earlier results in a critical path of 17 weeks, 4 weeks earlier than the expected termination of the project. Notice that in the example, the critical path is formed by the tasks Analysis and design, Design subsystem tests, Design system tests, Produce user documentation, Execute subsystem tests, and Execute system tests. In fact, the limited availability of resources results in dependencies among Design subsystem tests, Design system tests and Produce user documentation that last longer than the parallel task Code and integration



# Risk Planning

Risk is an inevitable part of every project, and so risk planning must be a part of every plan. Risks cannot be eliminated, but they can be assessed, controlled, and monitored.

Risk Management in the Quality Plan: Risks Generic to Process Management The quality plan must identify potential risks and define appropriate control tactics. Some risks and control tactics are generic to process management, while others are specific to the quality process. Here we provide a brief overview of some risks generic to process management.

| Technology Risks | Example Control Tactics |
|---|---|
| Many faults are introduced interfacing to an unfamiliar commercial off-the-shelf (COTS) component. | Anticipate and schedule extra time for testing unfamiliar interfaces; invest training time for COTS components and for training with new tools; monitor, document, and publicize common errors and correct idioms; introduce new tools in lower-risk pilot projects or prototyping exercises. |
| Test and analysis automation tools do not meet expectations. | Introduce new tools in lower-risk pilot projects or prototyping exercises; anticipate and schedule time for training with new tools. |
| COTS components do not meet quality expectations. | Include COTS component qualification testing early in project plan; introduce new COTS components in lower-risk pilot projects or prototyping exercises. |

| Personnel Risks | Example Control Tactics |
|---|---|
| A staff member is lost (becomes ill, changes employer, etc.) or is underqualified for task (the project plan assumed a level of skill or familiarity that the assigned member did not have). | Cross train to avoid overdependence on individuals; encourage and schedule continuous education; provide open communication with opportunities for staff self-assessment and identification of skills gaps early in the project; provide competitive compensation and promotion policies and a rewarding work environment to retain staff; include training time in the project schedule. |

| Schedule Risks | Example Control Tactics |
|---|---|
| Inadequate unit testing leads to unanticipated expense and delays in integration testing. | Track and reward quality unit testing as evidenced by low-fault densities in integration. |
| Difficulty of scheduling meetings makes inspection a bottleneck in development. | Set aside times in a weekly schedule in which inspections take precedence over other meetings and other work; try distributed and asynchronous inspection techniques, with a lower frequency of face-to-face inspection meetings. |

| Development Risks | Example Control Tactics |
|---|---|
| | Provide early warning and feedback; schedule |
| Poor quality software delivered to testing group or inadequate unit test and analysis before committing to the code base. | inspection of design, code and test suites; connect development and inspection to the reward system; increase training through inspection; require coverage or other criteria at unit test level. |
| **Executions Risks** | **Example Control Tactics** |
| Execution costs higher than planned; scarce resources available for testing (testing requires expensive or complex machines or systems not easily available.) | Minimize parts that require full system to be executed; inspect architecture to assess and improve testability; increase intermediate feedback; invest in scaffolding. |
| **Requirements Risks** | **Example Control Tactics** |
| High assurance critical requirements. | Compare planned testing effort with former projects with similar criticality level to avoid underestimating testing effort; balance test and analysis; isolate critical parts, concerns and properties. |

# Monitoring the Process

Effective monitoring, naturally, depends on a plan that is realistic, well organized, and sufficiently detailed with clear, unambiguous milestones and criteria

Successful completion of a planned activity must be distinguished from mere termination, as otherwise it is too tempting to meet an impending deadline by omitting some planned work. Skipping planned verification activities or addressing them superficially can seem to accelerate a late project, but the boost is only apparent; the real effect is to postpone detection of more faults to later stages in development, where their detection and removal will be far more threatening to project success.

Monitoring produces a surfeit of detail about individual activities. Managers need to make decisions based on an overall understanding of project status, so raw monitoring information must be aggregated in ways that provide an overall picture.

Accurate classification schemata can improve monitoring and may be used in very large projects, where the amount of detailed information cannot be summarized in overall data. The orthogonal defect classification (ODC) approach has two main steps: (1) fault classification and (2) fault analysis

**Orthogonal defect classification (ODC)**

ODC fault classification is done in two phases: when faults are detected and when they are fixed. At detection time, we record the activity executed when the fault is revealed, the trigger that exposed the fault, and the perceived or actual impact of the fault on the customer

Fault, which can be: missing, that is, the fault is to due to an omission, as in a missing statement; incorrect, as in the use of a wrong parameter;

or extraneous, that is, due to something not relevant or pertinent to the document or code, as in a section of the design document that is not pertinent to the current product and should be removed. The source of the fault indicates the origin of the faulty modules: in-house, library, ported from other platforms, or outsourced code

**Distribution of fault types versus activities** Different quality activities target different classes of faults. For example, algorithmic (that is, local) faults are targeted primarily by unit testing, and we expect a high proportion of faults detected by unit testing to be in this class.

**Distribution of triggers over time during field test** Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late. In both cases, the rate of disclosure of new faults should asymptotically decrease

**Age distribution over target code** Most faults should be located in new and rewritten code, while few faults should be found in base or re-fixed code, since base and re-fixed code has already been tested and corrected

**Distribution of fault classes over time** the proportion of missing code faults should gradually decrease, while the percentage of extraneous faults may slowly increase, because missing functionality should be revealed with use and repaired, while extraneous code or documentation may be produced by updates

## Improving the Process

The occurrence of many such faults can be reduced by modifying the process and environment. For example, resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks Root cause analysis (RCA) is a technique for identifying and eliminating process faults. RCA was first developed in the nuclear power industry and later extended to software analysis. It consists of four main steps to select significant classes of faults and track them back to their original causes: What, When, Why, and How

What are the faults? The goal of this first step is to identify a class of important faults. Faults are categorized by severity and kind. The severity of faults characterizes the impact of the fault on the product

| Level | Description | Example |
|---|---|---|
| Critical | The product is unusable. | The fault causes the program to crash. |
| Severe | Some product features cannot be used, and there is no workaround. | The fault inhibits importing files saved with a previous version of the program, and there is no way to convert files saved in the old format to the new one. |
| Moderate | Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability. | The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but the process is not obvious or documented (loss of usability) and requires extra steps (loss of efficiency). |
| Cosmetic | Minor inconvenience. | The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience. |

**ODC Classification of Triggers Listed by Activity**

## Design Review and Code Inspection

**Design Conformance** A discrepancy between the reviewed artifact and a prior-stage artifact that serves as its specification.

**Logic/Flow** An algorithmic or logic flaw.

**Backward Compatibility** A difference between the current and earlier versions of an artifact that could be perceived by the customer as a failure.

**Internal Document** An internal inconsistency in the artifact (e.g., inconsistency between code and comments).

**Lateral Compatibility** An incompatibility between the artifact and some other system or module with which it should interoperate.

**Concurrency** A fault in interaction of concurrent processes or threads.

**Language Dependency** A violation of language-specific rules, standards, or best practices.

**Side Effects** A potential undesired interaction between the reviewed artifact and some other part of the system.

**Rare Situation** An inappropriate response to a situation that is not anticipated in the artifact.

(Error handling as specified in a prior artifact design conformance, not rare situation.)

## Structural (White-Box) Test

**Simple Path** The fault is detected by a test case derived to cover a single program element.

**Complex Path** The fault is detected by a test case derived to cover a combination of program elements.

## Functional (Black-Box) Test

**Coverage** The fault is detected by a test case derived for testing a single procedure (e.g., C function or Java method), without considering combination of values for possible parameters.

**Variation** The fault is detected by a test case derived to exercise a particular combination of parameters for a single procedure.

**Sequencing** The fault is detected by a test case derived for testing a sequence of procedure calls.

**Interaction** The fault is detected by a test case derived for testing procedure interactions.

## System Test

**Workload/Stress** The fault is detected during workload or stress testing.

**Recovery/Exception** The fault is detected while testing exceptions and recovery procedures.

**Startup/Restart** The fault is detected while testing initialization conditions during start up or after possibly faulty shutdowns.

**Hardware Configuration** The fault is detected while testing specific hardware configurations.

**Software Configuration** The fault is detected while testing specific software configurations.

**Blocked Test** Failure occurred in setting up the test scenario.

## ODC Classification of Customer Impact

**Installability** Ability of the customer to place the software into actual use. (Usability of the installed software is not included.)

**Integrity/Security** Protection of programs and data from either accidental or malicious destruction or alteration, and from unauthorized disclosure.

**Performance** The perceived and actual impact of the software on the time required for the customer and customer end users to complete their tasks.

**Maintenance** The ability to correct, adapt, or enhance the software system quickly and at minimal cost.

**Serviceability** Timely detection and diagnosis of failures, with minimal customer impact.

**Migration** Ease of upgrading to a new system release with minimal disruption to existing customer data and operations.

**Documentation** Degree to which provided documents (in all forms, including electronic) completely and correctly describe the structure and intended uses of the software.

**Usability** The degree to which the software and accompanying documents can be understood and effectively employed by the end user.

**Standards** The degree to which the software complies with applicable standards.

**Reliabilit**y The ability of the software to perform its intended function without unplanned interruption or failure.

**Accessibility** The degree to which persons with disabilities can obtain the full benefit of the software system.

**Capability** The degree to which the software performs its intended functions consistently with documented system requirements.

**Requirements** The degree to which the system, in complying with document requirements, actually meets customer expectations

## ODC Classification of Defect Types for Targets Design and Code

**Assignment/Initialization** A variable was not assigned the correct initial value or was not assigned any initial value.

**Checking** Procedure parameters or variables were not properly validated before use.

**Algorithm/Method** A correctness or efficiency problem that can be fixed by re implementing a single procedure or local data structure, without a design change.

**Function/Class/Object** A change to the documented design is required to conform to product requirements or interface specifications.

**Timing/Synchronization** The implementation omits necessary synchronization of shared resources, or violates the prescribed synchronization protocol.

**Interface/Object-Oriented Messages** Module interfaces are incompatible; this can include syntactically compatible interfaces that differ in semantic interpretation of communicated data.

**Relationship** Potentially problematic interactions among procedures, possibly involving different assumptions but not involving interface incompatibility

**When did faults occur, and when were they found**? It is typical of mature software processes to collect fault data sufficient to determine when each fault was detected (e.g., in integration test or in a design inspection). In addition, for the class of faults identified in the first step, we attempt to determine when those faults were introduced (e.g., was a particular fault introduced in coding, or did it result from an error in architectural design?).

**Why did faults occur?** In this core RCA step, we attempt to trace representative faults back to causes, with the objective of identifying a "root" cause associated with many faults in the class. Analysis proceeds iteratively by attempting to explain the error that led to the fault, then the cause of that error, the cause of that cause, and so on

**How could faults be prevented?** The final step of RCA is improving the process by removing root causes or making early detection likely. The measures taken may have a minor impact on the development process (e.g., adding consideration of exceptional conditions to a design inspection checklist), or may involve a substantial modification of the process ODC and RCA

are two examples of feedback and improvement, which are an important dimension of most good software processes

## The Quality Team

The quality plan must assign roles and responsibilities to people. As with other aspects of planning, assignment of responsibility occurs at a strategic level and a tactical level. The tactical level, represented directly in the project plan, assigns responsibility to individuals in accordance with the general strategy. It involves balancing level of effort across time and carefully managing personal interactions. The strategic level of organization is represented not only in the quality strategy document, but in the structure of the organization itself

The strategy for assigning responsibility may be partly driven by external requirements. For example, independent quality teams may be required by certification agencies or by a client organization

When quality tasks are distributed among groups or organizations, the plan should include specific checks to ensure successful completion of quality activities. For example, when module testing is performed by developers and integration and system testing is performed by an independent quality team, the quality team should check the completeness of module tests performed by developer

Many variations and hybrid models of organization can be designed. Some organizations have obtained a good balance of benefits by rotating responsibilities. For example, a developer may move into a role primarily responsible for quality in one project and move back into a regular development role in the next. In organizations large enough to have a distinct quality or testing group, an appropriate balance between independence and integration typically varies across levels of project organization

The plan must clearly define milestones and delivery for outsourced activities, as well as checks on the quality of delivery in both directions: Test organizations usually perform quick checks to verify the consistency of the software to be tested with respect to some minimal "testability" requirements; clients usually check the completeness and consistency of test results

# CHAPTER 2

## Documenting Analysis and Test

Documentation can be inspected to verify progress against schedule and quality goals and to identify problems, supporting process visibility, monitoring, and replicability.

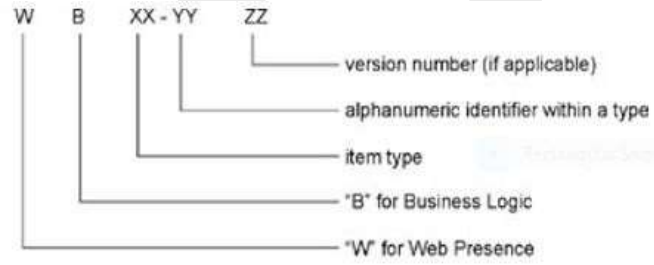
Documentation is an important element of the software development process, including the quality process. Complete and well-structured documents increase the reusability of test suites within and across projects

Documents are divided into three main categories: planning, specification, and reporting. Planning documents describe the organization of the quality process and include strategies and plans for the division or the company, and plans for individual projects. Specification documents describe test suites and test cases. A complete set of analysis and test specification documents include test design specifications, test case specification, checklists, and analysis procedure specifications. Reporting documents include details and summary of analysis and test results

## Organizing Documents

In larger projects, it is common practice to produce and regularly update a global guide for navigating among individual documents.

Naming conventions help in quickly identifying documents. A typical standard for document names would include keywords indicating the general scope of the document, its nature, the specific document, and its version




**analysis and test documentation**

| | |
|---|---|
| WB05-YYZZ | analysis and test strategy |
| WB06-YYZZ | analysis and test plan |
| WB07-YYZZ | test design specifications |
| WB08-YYZZ | test case specification |
| WB09-YYZZ | checklists |
| WB10-YYZZ | analysis and test logs |
| WB11-YYZZ | analysis and test summary reports |
| WB12-YYZZ | other analysis and test documents |

# Chipmunk Document Template Document Title

**Approvals**

issued by name signature date

approved by name signature date

distribution status (internal use only, restricted, …)

distribution list (people to whom the document must be sent)

**History**

Version description

**Table of Contents** List of sections.

**Summary**

Summarize the contents of the document. The summary should clearly explain the relevance of the document to its possible uses.

**Goals of the document** Describe the purpose of this document: Who should read it, and why?

**Required documents and references** Provide a reference to other documents and artifacts needed for understanding and exploiting this document. Provide a rationale for the provided references.

**Glossary** Provide a glossary of terms required to understand this document.

**Section 1**

…

**Section N**

…

# Test Strategy Document

**Overall quality:** Strategy documents indicate common quality requirements across products. Requirements may depend on business conditions. For example, a company that produces safety-critical software may need to satisfy minimum dependability requirements defined by a certification authority, while a department that designs software embedded in hardware products may need to ensure portability across product lines.

**Documentation quality:** The strategy document sets out requirements on other quality documents, typically including an analysis and test plan, test design specifications, test case specifications, test logs, and test summary reports**.**

# Analysis and Test Plan

A typical structure of a test and analysis plan includes information about items to be verified, features to be tested, the testing approach, pass and fail criteria, test deliverables, tasks, responsibilities and resources, and environment constraints

## A Standard Organization of an Analysis and Test Plan

Analysis and test items: The items to be tested or analyzed. The description of each item indicates version and installation procedures that may be required.

**Features to be tested:** The features considered in the plan. Features not to be tested: Features not considered in the current plan.

**Approach**: The overall analysis and test approach, sufficiently detailed to permit identification of the major test and analysis tasks and estimation of time and resources.

**Pass/Fail criteria:** Rules that determine the status of an artifact subjected to analysis and test.

**Suspension and resumption criteria:** Conditions to trigger suspension of test and analysis activities (e.g., an excessive failure rate) and conditions for restarting or resuming an activity.

**Risks and contingencies**: Risks foreseen when designing the plan and a contingency plan for each of the identified risks.

**Deliverables:** A list all A&T artifacts and documents that must be produced.

**Task and schedule:** A complete description of analysis and test tasks, relations among them, and relations between A&T and development tasks, with resource allocation and constraints. A task schedule usually includes GANTT and PERT diagrams.

**Staff and responsibilities:** Staff required for performing analysis and test activities, the required skills, and the allocation of responsibilities among groups and individuals. Allocation of resources to tasks is described in the schedule.

**Environmental needs:** Hardware and software required to perform analysis or testing activities

# Test Design Specification Documents

Design documentation for test suites and test cases serve essentially the same purpose as other software design documentation, guiding further development and preparing for maintenance. Test suite design must include all the information needed for initial selection of test cases and maintenance of the test suite over time, including rationale and anticipated evolution. Specification of individual test cases includes purpose, usage, and anticipated changes.

## Functional Test Design Specification of check configuration

**Test Suite Identifier** WB07-15.01

**Features to Be Tested** Functional test for check configuration, module specification WB02-15.32.[a]

**Approach** Combinatorial functional test of feature parameters, enumerated by category partition method over parameter table on page 3 of this document.[b]

**Procedure** Designed for conditional inclusion in nightly test run. Build target T02 15 32 11 includes JUnit harness and oracles, with test reports directed to standard test log. Test environment includes table MDB 15 32 03 for loading initial test database state.

**Test cases**

WB0715.01.C01 malformed model number

WB0715.01.C02 model number not in DB … …

WB0715.01.C09 valid model number with all legal required slots and some legal optional slots … …

WB0715.01.C19 empty model DB

WB0715.01.C23 model DB with a single element

WB0715.01.C24 empty component DB WB0715.01.C29 component DB with a single element

**Pass/Fail Criterion** Successful completion requires correct execution of all test cases with no violations in test log.

# Test and Analysis Reports

Reports of test and analysis results serve both developers and test designers. They identify open faults for developers and aid in scheduling fixes and revisions. They help test designers assess and refine their approach, for example, noting when some class of faults is escaping early test and analysis and showing up only in subsystem and system testing

## Test Case Specification for check configuration

**Test Case Identifier** WB07-15.01.C09

**Test items** Module check configuration of the Chipmunk Web presence system, business logic subsystem.

**Input specification**

Test Case Specification:

| | |
|---|---|
| Model No. | valid |
| No. of required slots for selected model (#SMRS) | many |
| No. of optional slots for selected model (#SMOS) | many |
| Correspondence of selection with model slots | complete |
| No. of required components with selection ≠ empty | = No. of required slots |
| No. of optional components with select ≠ empty | < No. of optional slots |
| Required component selection | all valid |
| Optional component selection | all valid |
| No. of models in DB | many |
| No. of components in DB | many |

**Test case:**

| | |
|---|---|
| Model number | Chipmunk C20 |
| #SMRS | 5 |
| Screen | 13" |
| Processor | Chipmunk II plus |

| | |
|---|---|
| Hard disk | 30 GB |
| RAM | 512 MB |
| OS | RodentOS 3.2 Personal Edition |
| #SMOS | 4 |
| External storage device | DVD player |

**Output Specification** return value valid

**Environment Needs** Execute with ChipmunkDBM v3.4 database initialized from table MDB 15 32 03.

**Special Procedural Requirements** none

**Intercase Dependencies** none

# MODULE V

# Integration and component based software testing

## Overview

Divides testing into four main levels of granularity: module, integration, system, and acceptance test. Module or unit test checks module behavior against specifications or expectations; integration test checks module compatibility; system and acceptance tests check behavior of the whole system with respect to specifications and user needs, respectively

While integration testing may to some extent act as a process check on module testing (i.e., faults revealed during integration test can be taken as a signal of unsatisfactory unit testing), thorough integration testing cannot fully compensate for sloppiness at the module level. In fact, the quality of a system is limited by the quality of the modules and components from which it is built, and even apparently noncritical modules can have widespread effects.

Integration faults are ultimately caused by incomplete specifications or faulty implementations of interfaces, resource usage, or required properties

## Integration faults

**Inconsistent interpretation of parameters or values** each module's interpretation may be reasonable, but they are incompatible Example: Unit mismatch: A mix of metric and British measures (meters and yards) is believed to have led to loss of the Mars Climate Orbiter in September 1999

**Violations of value domains or of capacity or size limits** implicit assumptions on ranges of values or sizes Example: Buffer overflow, in which an implicit (unchecked) capacity bound imposed by one module is violated by another, has become notorious as a security vulnerability. For example, some versions of the Apache 2 Web server between 2.0.35 and 2.0.50 could overflow a buffer while expanding environment variables during configuration file parsing

**Side-effects on parameters or resources** Example: A module often uses resources that are not explicitly mentioned in its interface. Integration problems arise when these implicit effects of one module interfere with those of another. For example, using a temporary file "tmp" may be invisible until integration with another module that also attempts to use a temporary file "tmp" in the same directory of scratch files

**Missing or misunderstood functionality** Example: Under specification of functionality may lead to incorrect assumptions about expected results Example: Counting hits on Web sites may

be done in many different ways: per unique IP address, per hit, including or excluding spiders, and so on. Problems arise if the interpretation assumed in the counting module differs from that of its clients

**Nonfunctional problems** Example: Nonfunctional properties like performance are typically specified explicitly only when they are expected to be an issue. Even when performance is not explicitly specified, we expect that software provides results in a reasonable time. Interference between modules may reduce performance below an acceptable threshold

**Dynamic mismatches** many languages and frameworks allow for dynamic binding. Problems may be caused by failures in matching's when modules are integrated Example: Polymorphic calls may be dynamically bound to incompatible methods

# Integration Testing Strategies

Integration testing proceeds incrementally with assembly of modules into successively larger subsystems. Incremental testing is preferred, first, to provide the earliest possible feedback on integration problems. In addition, controlling and observing the behavior of an integrated collection of modules grows in complexity with the number of modules and the complexity of their interactions

**Big bang testing** One extreme approach is to avoid the cost of scaffolding by waiting until all modules are integrated, and testing them together - essentially merging integration testing into system testing. In this big bang approach, neither stubs nor drivers need be constructed, nor must the development be carefully planned to expose well-specified interfaces to each subsystem. These savings are more than offset by losses in observability, diagnosability, and feedback

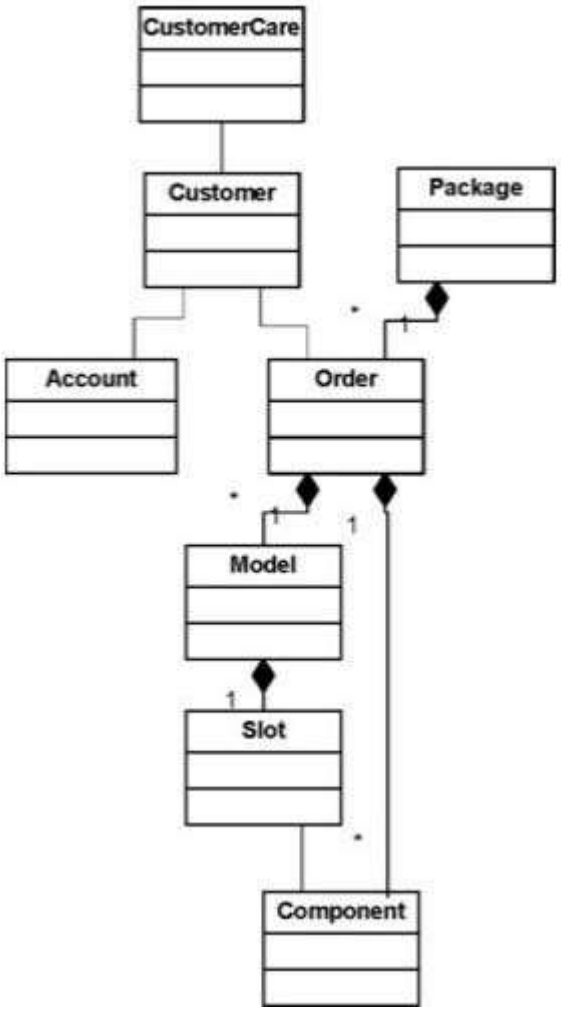
**Structural integration test strategy:** Among strategies for incrementally testing partially assembled systems, we can distinguish two main classes: structural and feature oriented. In a structural approach, modules are constructed, assembled, and tested together in an order based on hierarchical structure in the design. Structural approaches include bottom-up, top-down, and a combination sometimes referred to as sandwich or backbone strategy. Feature oriented strategies derive the order of integration from characteristics of the application, and include threads and critical modules strategies.

**Top-down and bottom up testing:** A top-down integration strategy begins at the top of the uses hierarchy, including the interfaces exposed through a user interface or top-level application program interface (API)

Bottom-up integration similarly reduces the need to develop stubs, except for breaking circular relations. Referring again to the example in Figure 21.1, we can start bottom-up by integrating Slot with Component, using drivers for Model and Order.We can then incrementally add Model

and Order. We can finally add either Package or Account and Customer, before integrating CustomerCare, without constructing stubs

**Sandwich or backbone:** Integration may combine elements of the two approaches, starting from both ends of the hierarchy and proceeding toward the middle. An early top-down approach may result from developing prototypes for early user feedback, while existing modules may be integrated bottom-up. This is known as the sandwich or backbone strategy




**Thread testing:** The thread integration testing strategy integrates modules according to system features. Test designers identify threads of execution that correspond to system features, and they incrementally test each thread. The thread integration strategy emphasizes module interplay for specific functionality

**Critical module:** Critical module integration testing focuses on modules that pose the greatest risk to the project. Modules are sorted and incrementally integrated according to the associated risk factor that characterizes the criticality of each module. Both external risks (such as safety) and project risks (such as schedule) can be considered.

# Testing Components and Assemblies

Many software products are constructed, partly or wholly, from assemblies of prebuilt software components.[1] A key characteristic of software components is that the organization that develops a component is distinct from the (several) groups of developers who use it to construct system

when reusing a component that has been in use in other applications for some time, one obtains the benefit not only of test and analysis by component developers, but also of actual operational use.

These advantages are balanced against two considerable disadvantages. First, a component designed for wide reuse will usually be much more complex than a module designed for a single use; a rule of thumb is that the development effort (including analysis and test) for a widely usable component is at least twice that for a module that provides equivalent functionality for a single application.

The interface specification of a component should provide all the information required for reusing the component, including so-called nonfunctional properties such as performance or capacity limits, in addition to functional behavior

The main problem facing test designers in the organization that produces a component is lack of information about the ways in which the component will be used. A component may be reused in many different contexts, including applications for which its functionality is an imperfect fit

Test designers cannot anticipate all possible uses of a component under test, but they can design test suites for classes of use in the form of scenarios. Test scenarios are closely related to scenarios or use cases in requirements analysis and design

## Terminology for Components and Frameworks

**Component** A software component is a reusable unit of deployment and composition that is deployed and integrated multiple times and usually by different teams. Components are characterized by a contract or interface and may or may not have state.

Components are often confused with objects, and a component can be encapsulated by an object or a set of objects, but they typically differ in many respects:

Components typically use persistent storage, while objects usually have only local state.

Components may be accessed by an extensive set of communication mechanisms, while objects are activated through method calls.

Components are usually larger grain subsystems than objects.

**Component contract or interface** The component contract describes the access points and parameters of the component, and specifies functional and nonfunctional behavior and any conditions required for using the component.

**Framework** A framework is a micro-architecture or a skeleton of an application, with hooks for attaching application-specific functionality or configuration-specific components. A framework can be seen as a circuit board with empty slots for components.

**Frameworks and design patterns** Patterns are logical design fragments, while frameworks are concrete elements of the application. Frameworks often implement patterns.

**Component-based system** A component-based system is a system built primarily by assembling software components (and perhaps a small amount of application specific code) connected through a framework or ad hoc "glue code."

**COTS** The term commercial off-the-shelf, or COTS, indicates components developed for the sale to other organizations.

# CHAPTER 2

# System, Acceptance, and Regression Testing

## Overview

System, acceptance, and regression testing are all concerned with the behavior of a software system as a whole, but they differ in purpose.

**System, Acceptance, and Regression Testing**

Open table as spreadsheet

| System test | Acceptance test | Regression test |
|---|---|---|
| Checks against requirements specifications | Checks suitability for user needs | Rechecks test cases passed by previous production versions |
| Performed by development test group | Performed by test group with user involvement | Performed by development test group |
| Validates usefulness and satisfaction with the product | Verifies correctness and completion of the product | Guards against unintended changes |

## System testing

System testing can be considered the culmination of integration testing, and passing all system tests is tantamount to being complete and free of known bugs. The system test suite may share some test cases with test suites used in integration and even unit testing, particularly when a thread-based or spiral model of development has been taken and subsystem correctness has been tested primarily through externally visible features and behavior

The appropriate notions of thoroughness in system testing are with respect to the system specification and potential usage scenarios, rather than code or design. Each feature or specified behavior of the system should be accounted for in one or several test cases

Some system properties, including performance properties like latency between an event and system response and reliability properties like mean time between failures, are inherently global

Global properties like performance, security, and safety are difficult to specify precisely and operationally, and they depend not only on many parts of the system under test, but also on its environment and use.

**Unit, Integration, and System Testing**

Open table as spreadsheet

| | Unit Test | Integration Test | System Test |
|---|---|---|---|
| Test cases derived from | module specifications | architecture and design specifications | requirements specification |
| Visibility required | all the details of the code | some details of the code, mainly interfaces | no details of the code |
| Scaffolding required | Potentially complex, to simulate the activation environment (drivers), the modules called by the module under test (stubs) and test oracles | Depends on architecture and integration order. Modules and subsystems can be incrementally integrated to reduce need for drivers and stubs. | Mostly limited to test oracles, since the whole system should not require additional drivers or stubs to be executed. Sometimes includes a simulated execution environment (e.g., for embedded systems). |
| Focus on | behavior of individual modules | module integration and interaction | system functionality |

## Acceptance Testing

The purpose of acceptance testing is to guide a decision as to whether the product in its current state should be released. The decision can be based on measures of the product or process.

Quantitative goals for dependability, including reliability, availability, and mean time between failure

Systematic testing, which includes all of the testing techniques presented heretofore in this book, does not draw statistically representative samples. Their purpose is not to fail at a "typical" rate, but to exhibit as many failures as possible. They are thus unsuitable for statistical testing.

A less formal, but frequently used approach to acceptance testing is testing with users. An early version of the product is delivered to a sample of users who provide feedback on failures and usability. Such tests are often called alpha and beta tests. The two terms distinguish between testing phases. Often the early or alpha phases are performed within the developing organization, while the later or beta phases are performed at users' sites.

In alpha and beta testing, the user sample determines the operational profil

# Usability

A usable product is quickly learned, allows users to work efficiently, and is pleasant to use. Usability involves objective criteria such as the time and number of operations required to perform tasks and the frequency of user error, in addition to the overall, subjective satisfaction of users

Even if usability is largely based on user perception and thus is validated based on user feedback, it can be verified early in the design and through the whole software life cycle. The process of verifying and validating usability includes the following main steps:

**Inspecting specifications** with usability checklists. Inspection provides early feedback on usability.

**Testing early prototypes** with end users to explore their mental model (exploratory test), evaluate alternatives (comparison test), and validate software usability. A prototype for early assessment of usability may not include any functioning software; a cardboard prototype may be as simple as a sequence of static images presented to users by the usability tester.

**Testing incremental releases** with both usability experts and end users to monitor progress and anticipate usability problems.

**System and acceptance testing** that includes expert-based inspection and testing, userbased testing, comparison testing against competitors, and analysis and checks often done automatically, such as a check of link connectivity and verification of browser compatibility.

User-based testing (i.e., testing with representatives of the actual end-user population) is particularly important for validating software usability. It can be applied at different stages, from early prototyping through incremental releases of the final system, and can be used with different goals: exploring the mental model of the user, evaluating design alternatives, and validating against established usability requirements and standards.

**Exploratory testing:** The purpose of exploratory testing is to investigate the mental model of end users. It consists of asking users about their approach to interactions with the system. For example, during an exploratory test for the Chipmunk Web presence, we may provide users with a generic interface for choosing the model they would like to buy, in order to understand how users will interact with the system

**comparison testing:** The purpose of comparison testing is evaluating options. It consists of observing user reactions to alternative interaction patterns. During comparison test we can, for example, provide users with different facilities to assemble the desired Chipmunk laptop configuration, and to identify patterns that facilitate users' interactions

**validation testing:** The purpose of validation testing is assessing overall usability. It includes identifying difficulties and obstacles that users encounter while interacting with the system, as well as measuring characteristics such as error rate and time to perform a task.

**Web Content Accessibility Guidelines (WCAG)[a]**

1. Provide equivalent alternatives to auditory and visual content that convey essentially the same function or purpose.
2. Ensure that text and graphics are understandable when viewed without color.
3. Mark up documents with the proper structural elements, controlling presentation with style sheets rather than presentation elements and attributes.
4. Use markup that facilitates pronunciation or interpretation of abbreviated or foreign text.
5. Ensure that tables have necessary markup to be transformed by accessible browsers and other user agents.
6. Ensure that pages are accessible even when newer technologies are not supported or are turned off.
7. Ensure that moving, blinking, scrolling, or auto-updating objects or pages may be paused or stopped.
8. Ensure that the user interface, including embedded user interface elements, follows principles of accessible design: device-independent access to functionality, keyboard operability, self-voicing, and so on.
9. Use features that enable activation of page elements via a variety of input devices.
10. Use interim accessibility so that assisting technologies and older browsers will operate correctly.
11. Where technologies outside of W3C specifications is used (e.g. Flash), provide alternative versions to ensure accessibility to standard user agents and assistive technologies (e.g., screen readers).
12. Provide context and orientation information to help users understand complex pages or elements.
13. Provide clear and consistent navigation mechanisms to increase the likelihood that a person will find what they are looking for at a site.
14. Ensure that documents are clear and simple, so they may be more easily understood.

# Regression Testing

Testing activities that focus on regression problems are called (non) regression testing. Usually "non" is omitted and we commonly say regression testing.

A simple approach to regression testing consists of reexecuting all test cases designed for previous versions. Even this simple retest all approach may present nontrivial problems and

costs. Former test cases may not be re executable on the new version without modification, and rerunning all test cases may be too expensive and unnecessary. A good quality test suite must be maintained across system versions

**Test case maintenance:** Changes in the new software version may impact the format of inputs and outputs, and test cases may not be executable without corresponding changes. Even simple modifications of the data structures, such as the addition of a field or small change of data types, may invalidate former test cases, or outputs comparable with the new ones.

Scaffolding that interprets test case specifications, rather than fully concrete test data, can reduce the impact of input and output format changes on regression testing

High-quality test suites can be maintained across versions by identifying and removing obsolete test cases, and by revealing and suitably marking redundant test cases. Redundant cases differ from obsolete, being executable but not important with respect to the considered testing criteria

## Regression Test Selection Techniques

Test case prioritization orders frequency of test case execution, executing all of them eventually but reducing the frequency of those deemed least likely to reveal faults by some criterion

Prioritization can be based on the specification and code-based regression test selection techniques described later in this chapter. In addition, test histories and fault-proneness models can be incorporated in prioritization schemes. For example, a test case that has previously revealed a fault in a module that has recently undergone change would receive a very high priority, while a test case that has never failed (yet) would receive a lower priority, particularly if it primarily concerns a feature that was not the focus of recent changes.

Regression test selection techniques are based on either code or specifications. Code based selection techniques select a test case for execution if it exercises a portion of the code that has been modified. Specification-based criteria select a test case for execution if it is relevant to a portion of the specification that has been changed. Code based regression test techniques can be supported by relatively simple tools. They work even when specifications are not properly maintained.

Control flow graph (CFG) regression techniques are based on the differences between the CFGs of the new and old versions of the software. Let us consider, for example, the C function cgi_ decode from Chapter 12. Figure 22.1 shows the original function as presented in Chapter 12, while Figure 22.2 shows a revision of the program. We refer to these two versions as 1.0 and 2.0, respectively. Version 2.0 adds code to fix a fault in interpreting hexadecimal sequences ‘%xy’. The fault was revealed by testing version 1.0 with input terminated by an erroneous subsequence ‘%x’, causing version 1.0 to read past the end of the input buffer and possibly overflow the

output buffer. Version 2.0 contains a new branch to map the unterminated sequence to a question mark.

```
1 #include "hex_values.h"
2 /** Translate a string from the CGI encoding to plain ascii te
3 *   '+' becomes space, %xx becomes byte with hex value xx,
4 *   other alphanumeric characters map to themselves.
5 *   Returns 0 for success, positive for erroneous input
6 *         1 = bad hexadecimal digit
7 */
8 int cgi_decode(char *encoded, char *decoded) {
9   char *eptr = encoded;
10  char *dptr = decoded;
11  int ok=0;
12  while (*eptr) {
13    char c;
14    c = *eptr;
15    if (c == '+') { /* Case 1: '+' maps to blank */
16      *dptr = ' ';
17    } else if (c == '%') { /* Case 2: '%xx' is hex for charact
18      int digit_high = Hex_Values[*(++eptr)]; /* note illegal
19      int digit_low = Hex_Values[*(++eptr)];
20      if ( digit_high == -1 || digit_low==-1) {
21        /* *dptr='?'; */
22        ok=1; /* Bad return code */
23      } else {
24        *dptr = 16* digit_high + digit_low;
25      }
26    } else { /* Case 3: Other characters map to themselves */
27      *dptr = *eptr;
28    }
29    ++dptr;
30    ++eptr;
31  }
32  *dptr = '\0';                    /* Null terminator for str
33  return ok;
34 }
```

**Figure 22.1**: C function *cgi_decode* version 1.0. The C function *cgi_decode* translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface of most Web servers. Repeated from Figure 12.1 in Chapter 12.

```
1 #include "hex values.h"
2 /** Translate a string from the CGI encoding to plain ascii te
3 * '+' becomes space, %xx becomes byte with hex value xx,
4 * other alphanumeric characters map to themselves, illegal to
5 * Returns 0 for success, positive for erroneous input
6 *       1 = bad hex digit, non-ascii char, or premature end.
7 */
8 int cgi_decode(char *encoded, char *decoded) {
9   char *eptr = encoded;
10  char *dptr = decoded;
11  int ok=0;
12  while (*eptr) {
13    char c;
14    c = *eptr;
15    if (c == '+') { /* Case 1: '+' maps to blank */
16      *dptr = ' ';
17    } else if (c == '%') { /* Case 2: '%xx' is hex for charact
18      if (! ( *(eptr + 1) && *(eptr + 2) )) { /* \%xx must pre
19        ok=1; return;
20      }
21      /* OK, we know the xx are there, now decode them */
22      int digit high = Hex Values[*(++eptr)]; /* note illegal
23      int digit low = Hex Values[*(++eptr)];
24      if ( digit high == -1 || digit low==-1) {
25        /* *dptr='?'; */
26        ok=1; /* Bad return code */
27      } else {
28        *dptr = 16* digit high + digit low;
29      }
30    } else { /* Case 3: Other characters map to themselves */
31      *dptr = *eptr;
32    }
33    if (! isascii(*dptr)) { /* Produce only legal ascii */
34      *dptr = '?';
35      ok=1;
36    }
37    ++dptr;
38    ++eptr;
39  }
40  *dptr = '\0'; /* Null terminator for string */
41  return ok;
42 }
```

**Figure 22.2**: Version 2.0 of the C function *cgi_decode* adds a control on hexadecimal escape sequences to reveal incorrect escape sequences at the end of the input string and a new branch to deal with non-ASCII characters.

# Test Case Prioritization and Selective Execution

Regression testing criteria may select a large portion of a test suite. When a regression test suite is too large, we must further reduce the set of test cases to be executed.

**Random sampling** is a simple way to reduce the size of the regression test suite. high-priority test cases are selected more often than low-priority test cases. With a good selection strategy, all test cases are executed sooner or later

**Execution history priority schema** Priorities can be assigned in many ways. A simple priority scheme assigns priority according to the execution history: Recently executed test cases are given low priority, while test cases that have not been recently executed are given high priority. In the extreme, heavily weighting execution history approximates round robin selection.

**Fault revealing priority schema** Other history-based priority schemes predict fault detection effectiveness. Test cases that have revealed faults in recent versions are given high priority. Faults are not evenly distribute

**Structural priority schema** Structural coverage leads to a set of priority schemes based on the elements covered by a test case.

Structural priority schemes produce several criteria depending on which elements we consider: statements, conditions, decisions, functions, files, and so on. The choice of the element of interest is usually driven by the testing level. Fine-grain elements such as statements and conditions are typically used in unit testing, while in integration or system testing one can consider coarser grain elements such as methods, features, and files.