CSE72
SOFTWARE TESTING

Unit 1: A Perspective on Testing

Basic Definitions

Error:

- A good synonym of **Error** is ***mistake***. When mistakes are done while coding, these are called as ***bugs***.
- Errors tend to propagate. An example is a requirements error may be magnified during design and amplified still more during coding.

Fault:

- A fault is the result of an error. A fault is the representation of an error, where representation is the mode of expression, such as narrative text, dataflow diagrams, hierarchy charts, source code, and so on.
- Defect is a good synonym for fault.
- Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation.
- **A fault of commission** occurs when we enter something into a representation that is incorrect.
- **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 a fault executes.
- Two subtleties arise here:
  1. Failures only occur in an executable representation, which is usually taken to be source code or loaded object code.
  2. This definition relates failures only to faults of commission. How to deal with failures that correspond to faults of omission? What about faults that never happens to execute, or do not execute for a long time? Reviews prevent many failures by finding faults and well-done reviews can find faults of omission.

Incident:

- When a failure occurs, it may or may not be readily apparent to the user. **An incident** is the symptom associated with a failure that **alerts** the user to the occurrence of a failure.

<u>Test</u>:

- A test is the act of exercising software with test cases.
- A test has two distinct goals: to find failures and to demonstrate correct execution.

<u>Test case:</u>

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

<u>A Testing Life Cycle:</u>

Figure 1.1 portrays a life cycle model for testing. In the development phases, three opportunities arise for errors to be made, resulting in faults that propagate through the remainder of the development process.

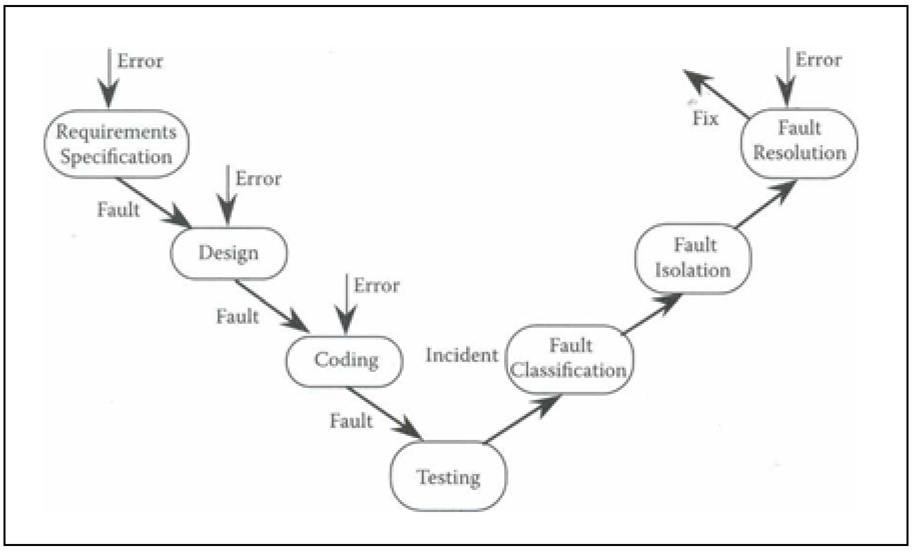


Figure: A Tesing Life Cycle

We can summarize this life cycle as follows:

The first three phases are Putting Bugs IN;
The testing phase is Finding Bugs;
The last three phases are Getting Bugs OUT.
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.

The process of testing can be subdivided into separate steps:

Test planning
Test case development

Running test cases
Evaluating test results

Test Cases

The essence of software testing is to determine a set of test cases for the item to be tested. Before going on, we need to clarify what information should be in a test case

A test case should contain the following information:

1. **Inputs:** There are two types:
   Preconditions: Circumstances that hold prior to test case execution
   Actual inputs: These are identified by some testing method
2. **Expected outputs:** There are two types:
   Postconditions
   Actual outputs
3. Test cases should have an **identity** and **a reason for being** (requirements tracing is a fine reason).
4. The execution **history** of a test case, including **when** and **by whom** it was run, the pass/fail **result** of each execution, and the **version** (of software) on which it was run.

Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date   Result   Version   Run By

Figure 1.2 Typical test case information.

The act of testing 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. Test cases need to be developed, reviewed, used, managed, and saved.

<u>Insights from a Venn diagram</u>

Testing is concerned with behavior, and behavior is orthogonal to the structural view common to software (system) developers. The structural view focuses on what it is and the behavioral view considers what it does.

Here a simple Venn diagram that clarifies several nagging questions about testing.

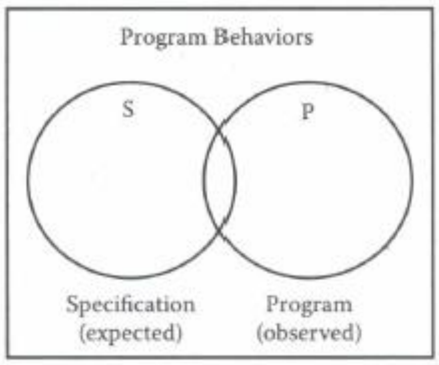


Figure 1.3 Specified and implemented program behaviors.

- Figure 1.3 shows the relationship among universe of program behaviors as well as the specified and programmed behaviors. Of all the possible program behaviors, the speci-fied behaviors are in the circle labeled **S**, and all those behaviors actually programmed are in **P**. The intersection of S and P is the "correct" portion, that is, behaviors that are both specified and implemented.

- With this diagram, we can see more clearly the problems faced by a tester. What if certain specified behaviors have not been programmed? These are faults of omission. What if certain programmed behaviors have not been specified? These correspond to faults of commission

- A view of testing is that it is the determination of the extent of program behavior that is both specified and implemented.

<u>The new circle in Figure 1.4 is for test cases.</u>

Consider the relationships among the 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).

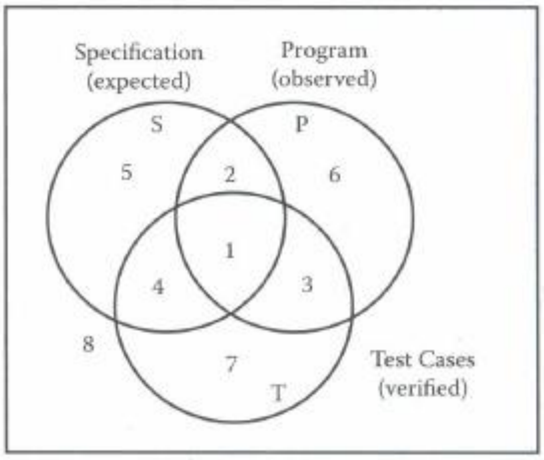


Figure 1.4 Specified, implemented, and tested behaviors.

- 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 unprogrammed behaviors (regions 4 and 7).
- If specified behaviors exist for which no test cases are available, the testing is **incomplete.** If certain test cases correspond to unspecified behaviors, then either such a test case is **unwarranted**, the specification is **deficient**, or the tester wishes to determine that specified **non-behavior** does not occur.

Identifying Test Cases

Two fundamental approaches are used to identify test cases,

1. Functional Testing
2. Structural testing.

Functional Testing

Functional testing consider any program to be a function that maps values from its input domain to values in its output range. This leads to the term **black box testing**, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs.

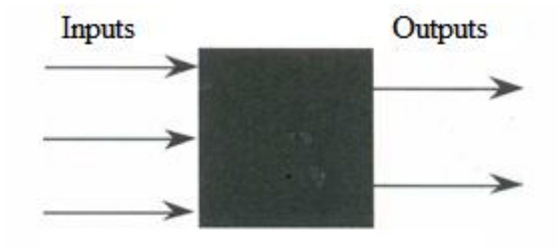


Figure 1.5 An engineer's black box

In the functional approach to test case identification, the only information used is the specification of the software.

Functional test cases have two distinct 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.

Functional test cases frequently suffer from two problems:

Significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software.

Figure 1.6 shows the results of test cases identified by two functional 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 functional methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified.

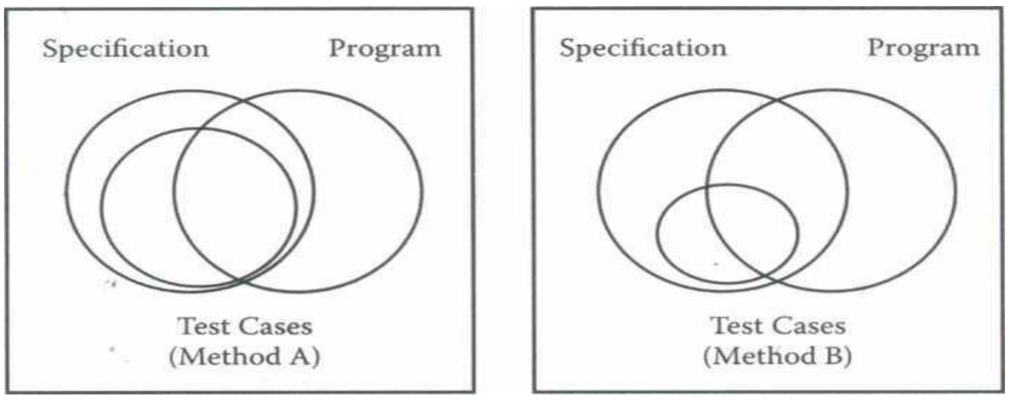


Figure 1.6 Comparing functional test case identification methods

Structural Testing

Structural testing is the other fundamental approach to test case identification. To contrast it with functional testing, it is sometimes called white box (or even clear box) testing. The clear box metaphor is probably more appropriate, because the essential difference is that the implementation (of the black box) is known and used to identify test cases. The ability to "see inside" the black box allows the tester to identify test cases based on how the function is actually implemented.
Structural testing has been the subject of some fairly strong theory. To really understand structural testing, familiarity with the concepts of linear graph theory is essential

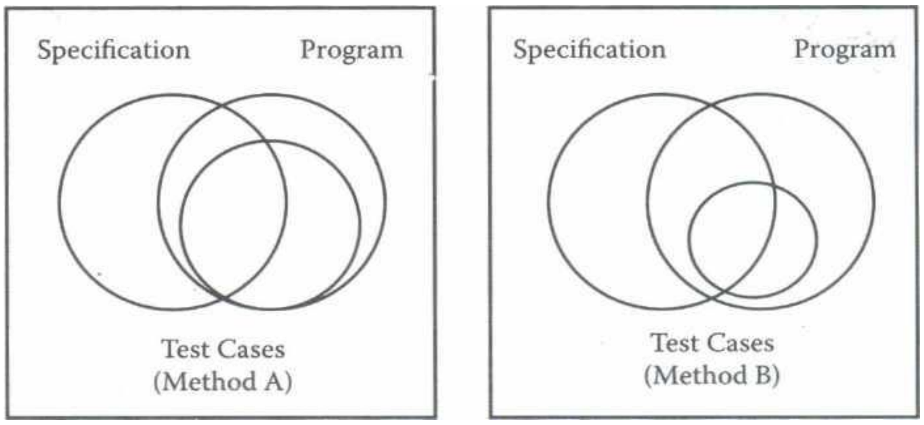


Figure 1.7 Comparing structural test case identification methods.

With these concepts, the tester can rigorously describe exactly what is tested. Because of its strong theoretical basis, structural testing lends itself to the definition and use of test coverage metrics. Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested, and this in turn makes testing management more meaningful.

Figure 1.7 shows the results of test cases identified by two structural methods. As before, Method A identifies a larger set of test cases than does Method B. Is a larger set of test cases necessarily better? This is an excellent question, and structural testing provides important ways to develop an answer. Notice that, for both methods, the set of test cases is completely contained within the set of programmed behavior. Because structural methods are based on the program, it is hard to imagine these methods identifying behaviors that are not programmed. It is easy to imagine, however, that a set of structural test cases is relatively small with respect to the full set of programmed behaviors. In Chapter 11 we will see direct comparisons of test cases generated by various structural methods.

The Functional versus Structural Debate

Given two fundamentally different approaches to test case identification, it is natural to question which is better. If you read much of the literature, you will find strong adherents to either choice. Referring to structural testing, Robert Poston writes, "This tool has been wasting tester's time since the 1970s ... [it] does not support good software testing practice and should not be in the tester's toolkit" (Poston, 1991).

In defense of structural testing, Edward Miller writes, "Branch coverage [a structural test coverage metric], if attained at the 85% or better level, tends to identify twice the number of defects that would have been found by 'intuitive' [functional] testing" (Miller, 1991).

The Venn diagrams presented earlier yield a strong resolution to this debate. Recall that the goal of both approaches is to identify test cases. Functional testing uses only the specification to identify test cases, while structural testing uses the program source code (implementation) as the basis of test case identification. Our earlier discussion forces the conclusion that neither approach alone is sufficient.
Consider program behaviors: if all specified behaviors have not been implemented, structural test cases will never be able to recognize this. Conversely, if the program implements behaviors that have not been specified, this will never be revealed by functional test cases. (A Trojan horse is a good example of such unspecified behavior.) The quick answer is that both approaches are needed; the testing craftsperson's answer is that a judicious combination will provide the confidence of functional testing and the measurement of structured testing.

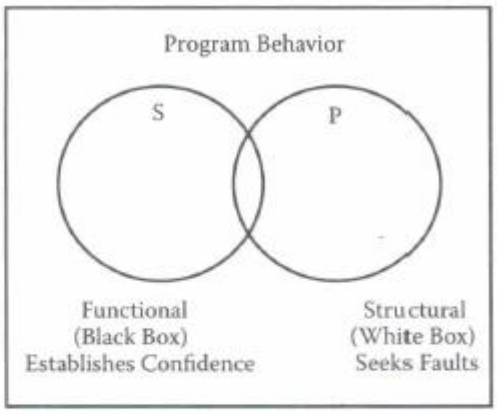


Figure 1.8 Sources of test cases.

Earlier, we asserted that functional testing often suffers from twin problems of redundancies and gaps. When functional test cases are executed in combination with structural test coverage metrics, both of these problems can be recognized and resolved (Figure 1.8).

The Venn diagram view of testing provides one final insight. What is the relationship between the set T of test cases and the sets S and P of specified and implemented behaviors? Clearly, the test cases in T are determined by the test case identification method used. A very good question to ask is how appropriate (or effective) is this method? To close a loop from an earlier discussion, recall the causal trail from error to fault, failure, and incident.
If we know what kind of errors we are prone to make, and if we know what kinds of faults are likely to reside in the software to be tested, we can use this to employ more appropriate test case identification methods. This is the point at which testing really becomes a craft.

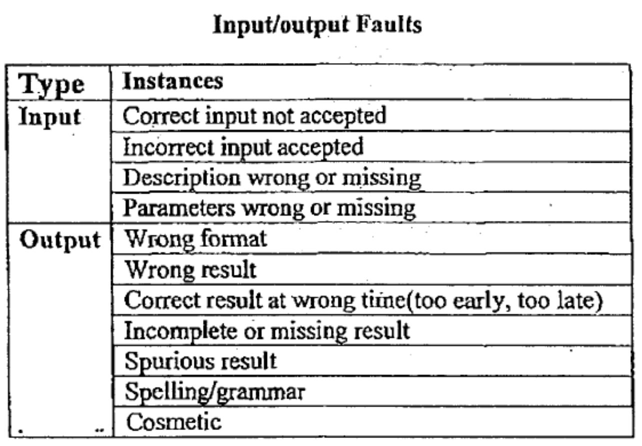
Error and Fault Taxonomies

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. In that sense, testing is clearly more product oriented. SQA is more concerned with reducing errors endemic in the development process, while testing is more concerned with discovering faults in a product. Both disciplines benefit from a clearer definition of types of faults.

Faults can be classified in several ways: the development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly occurrence: one time only, intermittent, recurring, or repeatable. Figure 1.9 contains a fault taxonomy (Beizer, 1984) that distinguishes faults by the severity of their consequences.
For a comprehensive treatment of types of faults, see the *IEEE Standard Classification for Software Anomalies* (IEEE, 1993). (A software anomaly is defined in that document as "a departure from the expected," which is pretty close to our definition.) The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition. Some of the more useful anomalies are given in Table 1.1 through Table 1.5; most of these are from the IEEE standard, but I have added some of my favorites.

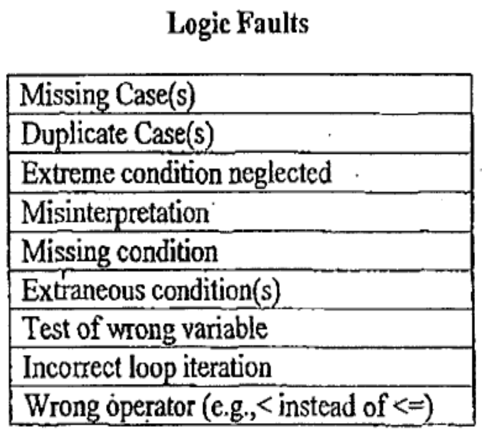
1. Mild	Misspelled word
2. Moderate	Misleading or redundant information
3. Annoying	Truncated names, bill for $0.00
4. Disturbing	Some transaction(s) not processed
5. Serious	Lose a transaction
6. Very serious	Incorrect transaction execution
7. Extreme	Frequent "very serious" errors
8. Intolerable	Database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

Figure 1.9 Faults classified by severity.



Input/output Faults

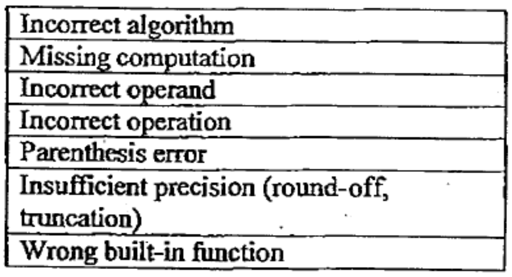
Type	Instances
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time(too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic



Logic Faults

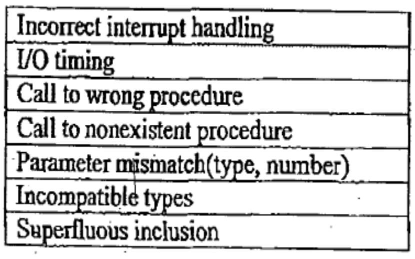
Missing Case(s)
Duplicate Case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g.,< instead of <=)

Computation Faults



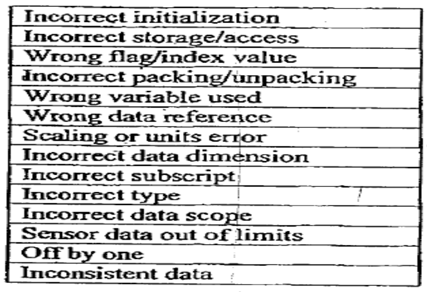
Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Interface faults



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

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

Thus far, we have said nothing about one of the key concepts of testing — levels of abstraction. Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level.

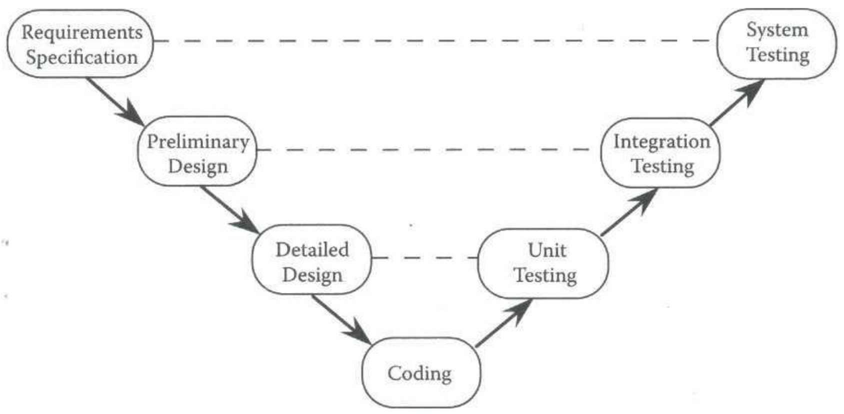


Figure 1.10 Levels of abstraction and testing in the waterfall model.

A diagrammatic variation of the waterfall model is given in Figure 1.10; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of functional 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 functional and structural testing. Most practitioners agree that structural testing is most appropriate at the unit level, while functional testing is most appropriate at the system level. This is generally true, but it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases. The constructs defined for structural 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. We develop such structures in Part IV to support structural testing at the integration and system levels for both traditional and object-oriented software.

Examples

Three examples will be used to illustrate the various unit testing methods. They are the triangle problem (a venerable example in testing circles); a logically complex function, NextDate; and an example that typifies Management Information Systems (MIS) applications, known here as the commission problem. Taken together, these examples raise most of the issues that testing craftspersons will encounter at the unit level. The discussion of integration and system testing in Part IV uses three other examples: a simplified version of an automated teller machine (ATM), known here as the simple ATM (SATM) system; the currency converter, an event-driven application typical of graphical user interface (GUI) applications; and the windshield wiper control device from the Saturn™ automobile. Finally, an object-oriented version of NextDate is provided, called o-oCalendar, which is used to illustrate aspects of testing object-oriented software.

For the purposes of structural testing, pseudocode implementations of the three unit-level examples are given in this chapter. System-level descriptions of the SATM system, the currency converter, and the Saturn windshield wiper system. These applications are described both traditionally (with E/R diagrams, dataflow diagrams, and finite state machines) and with the de facto object-oriented standard, the Unified Modeling Language (UML).

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). This definition is somewhat informal; terms such as expression, variable list, and field description are used with no formal definition. Items in angle brackets indicate language elements that can be used at the identified positions. Part of the value of any pseudocode is the suppression of unwanted detail; here, we illustrate this by allowing natural language phrases in place of more formal, complex conditions (see Table 2.1).

Language Element	*Generalized Pseudocode Construct*
Comment	' <text>
Data structure declaration	Type <type namexlist of field descriptions>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> <Condition>
Sequence	Statements in sequential order
Simple selection	If <condition> Then <then clause>EndIf
Selection	If <condition>
Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> Case n: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end>
Pretest repetition	While <condition> ... End While
Posttest repetition	Do ... until <condition>
Procedure definition (similarly for functions and o-o methods)	<procedure name>(Input: <list of variables>;Output: <list of variables>)
Interunit communication	Call <procedure name> (<list of variables>; <list of variables>)
Class/object definition	<name> (ottribute list>; <method list>, <body>)End <name>
Interunit communication	msg <destination object name>.<method name> (<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>

The Triangle Problem

The triangle problem is the most widely used example in software testing literature. Some of the more notable entries in three decades of testing literature are Gruenberger (1973), Brown and Lipov (1975), Myers (1979), Pressman (1982, and subsequent editions), Clarke (1983, 1984), Chellappa (1987), and Hetzel (1988). There are others, but this list makes the point.

Problem Statement

Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or Not A Triangle. Sometimes this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

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

The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. If an input value fails any of conditions cl, c2, or c3, the program notes this with an output message, for example, "Value of b is not in the range of permitted values." If values of a, b, and c satisfy conditions cl, c2, and c3, one of four mutually exclusive outputs is given:

If all three sides are equal, the program output is Equilateral.
If exactly one pair of sides is equal, the program output is Isosceles.
If no pair of sides is equal, the program output is Scalene.
If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

Discussion

Perhaps one of the reasons for the longevity of this example is that it contains clear but complex logic. It also typifies some of the incomplete definitions that impair communication among customers, developers, and testers. The first specification presumes the developers know some details about triangles, particularly the triangle inequality: the sum of any pair of sides must be strictly greater than the third side. The upper limit of 200 is both arbitrary and convenient; it will be used when we develop boundary value test cases.

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. The flowchart box numbers correspond to comment numbers in the (Fortran-like) pseudocode program given next.

```
Program trianglel   'Fortran-like version
Dim a, b, c, match As Integer
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
match = 0
If (a = b)
    Then match = match +1
EndIf
If a = c
    Then match = match + 2
EndIf
If b = c
```

```
        Then match = match + 3
EndIf

If match =0
        Then    If  (a+b)< = c
                Then      Output  ("NotATriangle")
                Else    If  (b+c)<=a
                        Then    Output  ("NotATriangle")
                Else    If  (a+c)<=b    ' (10)
                        Then   Output  ("NotATriangle")
                        Else   Output  ("Scalene")
                        EndIf
                EndIf
         EndIf

Else   If match=l
        Then   If  (a+c)  <=b
                        Then   Output ("NotATriangle")
                        Else   Output  ("Isosceles")
                EndIf
        Else   If match=2
                        Then   If (a+c)<=b
                                        Then  Output  ("NotATriangle")
                                        Else  Output  ("Isosceles")
                                EndIf
                Else   If match=3
                        Then   If  (b+c)<=a
                                Then  Output ("NotATriangle")
                                Else  Output  ("Isosceles")
                               EndIf
                          Else  Output  ("Equilateral")
                        EndIf
                  EndIf
                EndIf
        EndIf

End Trianglel
```

The variable "match" is used to record equality among pairs of the sides. A classical 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. (Because b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the number of comparisons that must be made.
The efficiency of this version is obtained at the expense of clarity (and ease of testing). Notice that six ways are used to reach the NotATriangle box, and three ways are used to reach the Isosceles box.

Structured Implementation

Figure 2.2 is a dataflow diagram description of the triangle program. We could implement it as a main program with the three indicated procedures. We will use this example later for unit testing; therefore, the three procedures have been merged into one pseudocode program. Comment lines relate sections of the code to the decomposition given in Figure 2.2

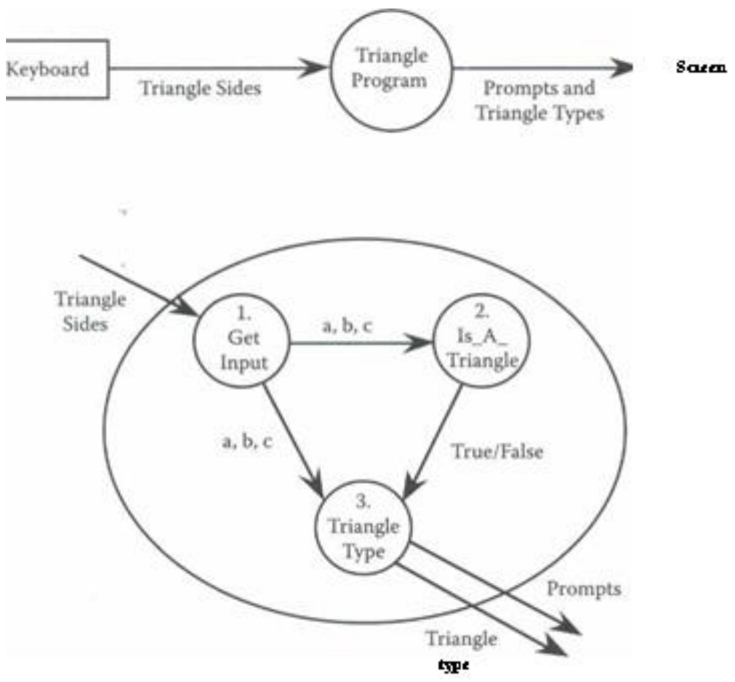


Figure 2.2 Dataflow diagram for a structured triangle program implementation.

Program triangle2 'Structured programming version of simpler specification

```
Dim a,b,c As Integer
Dim IsATriangle As Boolean

Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)

Step 2:  Is A Triangle?
If  (a < b + c) AND  (b < a + c) AND  (c < a + b)
        Then IsATriangle = True
        Else IsATriangle = False
EndIf
Step 3: Determine Triangle Type
If IsATriangle
```

```
     Then   If  (a = b)  AND  (b = c)
            Then Output ("Equilateral")
            Else      If   (a ≠ b) AND  (a ≠ c)  AND  (b ≠ c)
                           Then     Output ("Scalene")
                           Else     Output ("Isosceles")
                      EndIf
            EndIf
     Else   Output   ("Not a Triangle")
EndIf
End Triangle2
```

<u>Program triangle3   'Structured programming version of improved specification</u>

```
Dim a,b,c As Integer
Dim IsATriangle As Boolean

Step 1: Get Input
Do
     Output ("Enter 3 integers which are sides of a triangle")
     Input (a,b,c)
     c1 = (1 <= a) AND (a <= 200)
     c2 = (1 <= b) AND (b <= 200)
     c3 = (1 <= c) AND (c <= 200)
     If NOT (c1)
          Then Output ("Value of a is not in the range of permitted values")
     EndIf
     If NOT (c2)
          Then Output ("Value of b is not in the range of permitted values")
     EndIf

     If NOT (c3)
          Then Output ("Value of c is not in the range of permitted values")
     EndIf

Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)

Step 2:  Is A Triangle?
If  (a < b + c) AND  (b < a + c) AND  (c < a + b)
     Then IsATriangle = True
     Else IsATriangle = False
EndIf
Step 3: Determine Triangle Type
```

```
If IsATriangle
        Then   If (a = b) AND (b = c)
               Then Output ("Equilateral")
               Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                      Then   Output ("Scalene")
                      Else    Output ("Isosceles")
               EndIf
          EndIf
        Else    Output   ("Not a Triangle")
EndIf
```

Problem Statement

NextDate is a function of three variables: month, day, and year. It returns the date of the day after the input date. The month, day, and year variables have integer values subject to these conditions:

cl. $1 < month < 12$

c2. $1 < day < 31$

c3. $1812 < year < 2012$

As we did with the triangle program, we can make our specification stricter. This entails defining responses for invalid values of the input values for the day, month, and year. We can also define responses for invalid combinations of inputs, such as June 31 of any year. If any of conditions cl, 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.'

Discussion

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. The Gregorian calendar (after Pope Gregory) resolves this by adjusting leap years on century years. Thus, a year is a leap year if it is divisible by 4, unless it is a century • year. Century years are leap years only if they are multiples of 400 (Inglis, 1961), so 1992, 1996, and 2000 are leap years, while the year 1900 is not. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's law, which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations. In the second implementation, notice how much code is devoted to input value validation.

Implementation

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

```
                                        Else If day = 29
                                                Then tomorrowDay = 1
                                                tomorrowMonth = 3
                                        Else Output  ("Cannot have Feb.", day)
                                        EndIf
                                 EndIf
                        EndIf
EndCase

Output  ("Tomorrow's date is",  tomorrowMonth, tomorrowDay, tomorrowYear)

End NextDate
```

<u>Program NextDate2   Improved version</u>

```
Dim tomorrowDay,tomorrowMonth,  tomorrowYear As Integer
Dim day,month,year As Integer
Dim cl,  c2,  c3 As Boolean

Do
        Output  ("Enter today's date in the form MM DD YYYY")
        Input  (month,day,year)
        cl =   (1 <= day)  AND  (day <= 31)
        c2 =  (1 <= month) AND  (month <= 12)
        c3 =  (1812 <= year)  AND  (year <= 2012)
        If NOT(cl)
                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 cl AND c2 AND c3
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 = 2 9  '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

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions.

Problem Statement

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gun-smith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to sell at least 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. After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town. At the end of a month, the salesperson sent a very short telegram showing -1 lock sold. 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. The commission program produced a monthly sales report that gave the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and, finally, the commission.

Discussion

This example is somewhat contrived to make the arithmetic quickly visible to the reader. It might be more realistic to consider some other additive function of several variables, such as various calculations found in filling out a U.S. 1040 income tax form. (We will stay with rifles.) This problem separates into three distinct pieces: the input data portion, in which we could deal with input data validation (as we did for the triangle and NextDate programs); the sales calculation; and the commission calculation portion. This time, we will omit the input data validation portion. We will replicate the telegram convention with a sentinel-controlled While loop that is typical of MIS data gathering applications.

Implementation

```
Program Commission  (INPUT,OUTPUT)

Dim locks,   stocks,  barrels As Integer
Dim lockPrice,   stockPrice,  barrelPrice As Real
Dim totalLocks,  totalStocks,totalBarrels As Integer
Dim lockSales,  stockSales,  barrelSales As Real
Dim sales,commission  : REAL
lockPrice = 45.0
stockPrice = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
```

```
totalBarrels = 0

Input(locks)
While NOT(locks = -1) 'Input device uses -1 to indicate end of data
        Input(stocks, barrels)
        totalLocks = totalLocks + locks
        totalStocks = totalStocks + stocks
        totalBarrels = totalBarrels + barrels
        Input(locks)
EndWhile
Output("Locks sold:   ", totalLocks)
Output("Stocks sold:   ", totalStocks)
Output("Barrels sold:   ",totalBarrels)

lockSales = lockPrice * totalLocks
stockSales = stockPrice * totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output("Total sales:   ", sales)

If   (sales > 1800.0) Then
        commission = 0.10 * 1000.0
        commission = commission + 0.15 * 800.0
        commission = commission + 0.20 *(sales-1800.0)
Else If   (sales > 1000.0) Then
        commission = 0.10 * 1000.0
        commission = commission + 0.15* (sales-1000.0)
        Else commission = 0.10 * sales
     EndIf
EndIf
Output("Commission is $",commission)
End Commission
```

2.5 The SATM System

To better discuss the issues of integration and system testing, we need an example with larger scope. The automated teller machine described here is a refinement of that in Topper (1993); it contains an interesting variety of functionality and interactions that typify the client side of client/server systems.

Problem Statement

The SATM system communicates with bank customers via the 15 screens shown in Figure 2.4. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. These transactions can be done on two types of accounts: checking and savings.

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 customer account 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.

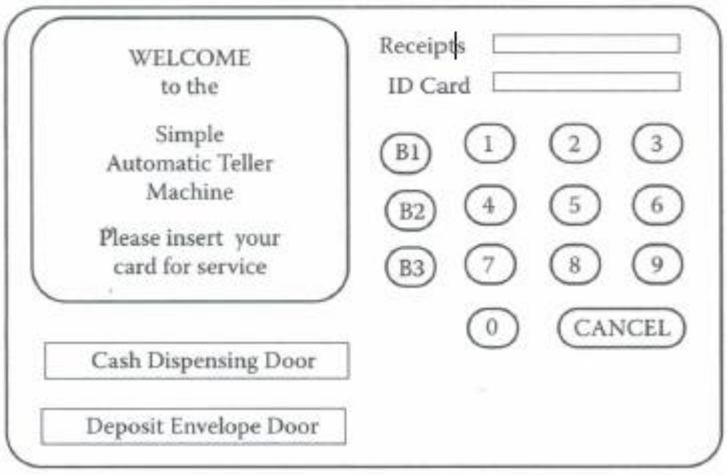


Figure 2.3 The SATM terminal

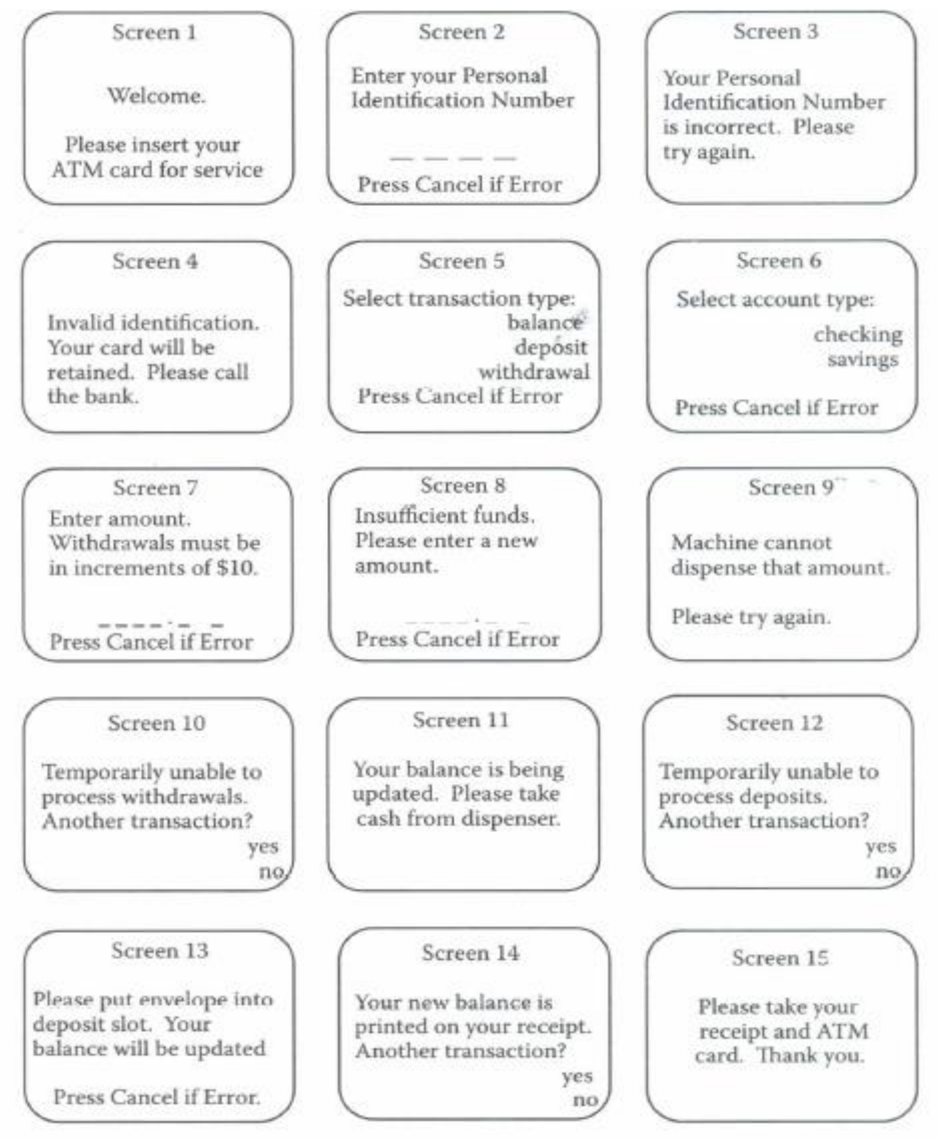
At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.

On entry to screen 5, the system adds two pieces of information to the customer's account file: the current date and an increment to the number of ATM sessions. The customer selects the desired transaction from the options shown on screen 5; then the system immediately displays screen 6, where the customer chooses the account to which the selected transaction will be applied.

If balance is requested, the system checks the local ATM file for any unposted transactions and reconciles these with the beginning balance for that day from the customer account file. 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 deposit amount is entered as an unposted amount in the local ATM file, and the count of deposits per month is incremented. Both of these (and other information) are processed by the master ATM (centralized) system once a day. 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 money to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed.



Screen 1
Welcome.
Please insert your ATM card for service

Screen 2
Enter your Personal Identification Number
_ _ _ _
Press Cancel if Error

Screen 3
Your Personal Identification Number is incorrect. Please try again.

Screen 4
Invalid identification. Your card will be retained. Please call the bank.

Screen 5
Select transaction type:
balance
deposit
withdrawal
Press Cancel if Error

Screen 6
Select account type:
checking
savings
Press Cancel if Error

Screen 7
Enter amount. Withdrawals must be in increments of $10.
_ _ _ _ . _ _
Press Cancel if Error

Screen 8
Insufficient funds. Please enter a new amount.
_ _ _ _ . _ _
Press Cancel if Error

Screen 9
Machine cannot dispense that amount.
Please try again.

Screen 10
Temporarily unable to process withdrawals. Another transaction?
yes
no

Screen 11
Your balance is being updated. Please take cash from dispenser.

Screen 12
Temporarily unable to process deposits. Another transaction?
yes
no

Screen 13
Please put envelope into deposit slot. Your balance will be updated
Press Cancel if Error.

Screen 14
Your new balance is printed on your receipt. Another transaction?
yes
no

Screen 15
Please take your receipt and ATM card. Thank you.

Fig 2.4 SATM Screens

The withdrawal amount is written to the unposted local ATM file, and the count of withdrawals per month is incremented. 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 screen 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 screen 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

Discussion

A surprising amount of information is "buried" in the system description just given. For instance, if you read it closely, you can infer that the terminal only contains $10 bills (see screen 7). This textual definition is probably more precise than what is usually encountered in practice. The example is deliberately simple (hence the name).

A plethora of questions could be resolved by a list of assumptions. For example: Is there a borrowing limit? What keeps a customer from taking out more than his actual balance if he goes to several ATM terminals? A lot of start-up questions are used: How much cash is initially in the machine? How are new customers added to the system? These and other real-world refinements are eliminated to maintain simplicity.

The Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a graphical user interface (GUI). A sample GUI built with Visual Basic is shown in Figure 2.5.

The application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (Visual Basic option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label; for example, "Equivalent in ..." becomes "Equivalent in Canadian dollars" if the Canada button is clicked. Also, a small Canadian flag appears next to the output position for the equivalent currency amount. Either before or after currency selection, the user inputs an amount in U.S. dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, ot the Quit button. Clicking on the Compute button results in the conversion of the U.S. dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the U.S. dollar amount, and the equivalent currency amount and the associated label. Clicking on the Quit button ends the application.

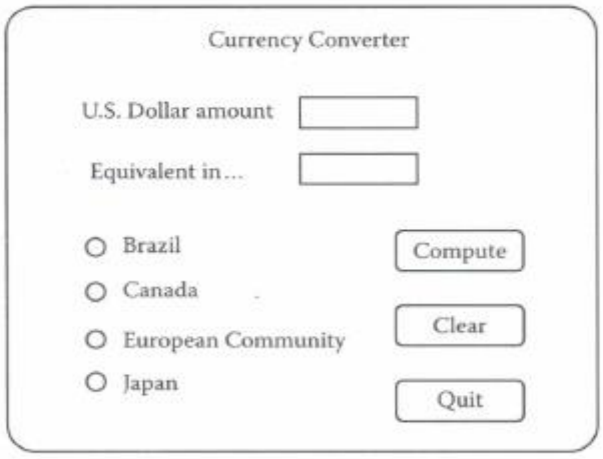
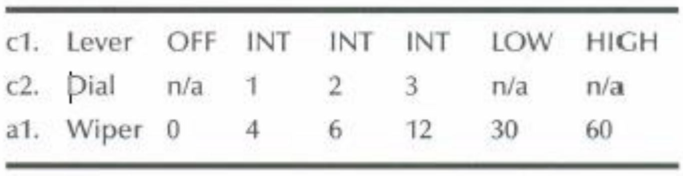


Figure: Currency Converter GUI

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

Chapter 2: Boundary Value Testing, Equivalence Class Testing, Decision Table- Based Testing

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 is the best-known functional testing technique. In this and the next two chapters, we examine how to use knowledge of the functional nature of a program to identify test cases for the program. Historically, this form of testing has focused on the input domain, but it is often a good supplement to apply many of these techniques to develop range-based test cases.

2.1 Boundary Value Analysis

For the sake of comprehensible drawings, the discussion relates to a function, F, of two variables, x, and x_2. When the function F is implemented as a program, the input variables x, and x_2 will have some (possibly unstated) boundaries:

$$a < x_1 < b$$
$$c < x_2 < d$$

Unfortunately, the intervals [a, b] and [c, d] are referred to as the ranges of x, and $x_{2>}$ so right away we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada® and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kinds of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, FORTRAN, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in such .languages. The input space (domain) of our function F is shown in Figure 2.1. Any point within the shaded rectangle is a legitimate input to the function F.

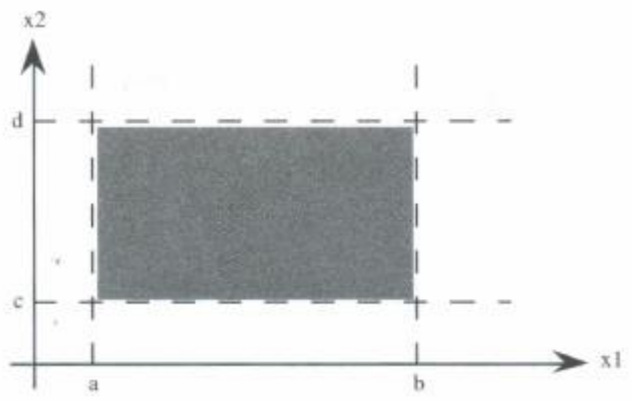


Figure 5.1 Input domain of a function of two variables.

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. Loop conditions, for example, may test for < when they should test for <, and counters are often "off by 1."
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 their maximum.

The next part of boundary value analysis is based on a critical assumption; it is known as the single fault assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. Thus, the boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values. The boundary value analysis test cases for our function F of two variables (illustrated in Figure 2.2) are

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

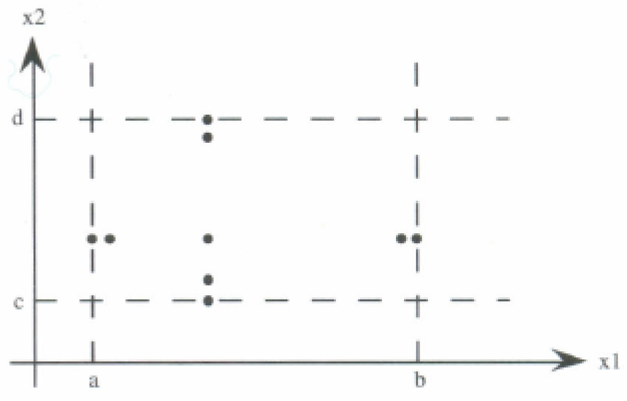


Figure 2.2 Boundary value analysis test cases for a function of two variables.

5.1.1 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. In a Fortran-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on.
In a language that supports user-defined types (like Pascal or Ada), we could define the variable month as an enumerated type {Jan., Feb., ... , Dec.}, Either way, the values for min, min+, nom, max-, and max are clear from the context.
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
When no explicit bounds are present, as in the triangle problem, we usually have to create artificial bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly), but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility, or we might impose an arbitrary upper limit such as 200 or 2000.

Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, bur no clear choice is available for the remaining three. Logical variables also present a problem for boundary value analysis.

5.7.2 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. The key words here are *independent* and *physical quantities.* A quick look at the boundary value analysis test cases for NextDate shows them to be inadequate. Very little stress occurs on February and on leap years, for example. The real problem here is that interesting dependencies exist among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extreme of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, or of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary because they are obtained with very little insight and imagination. As with so many things, you get what you pay for.
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. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to, close on June 26, 1992, because the air temperature was 122°F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120°F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell.

As an example of logical (versus physical) 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.

5.2 Robustness Testing

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

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 runtime errors that abort normal execution. This raises an interesting question of implementation philosophy: Is it better to perform explicit range checking and use exception handling to deal with robust values, or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

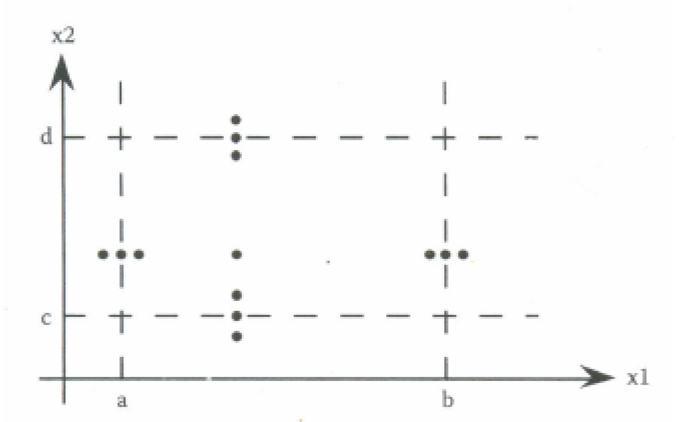


Figure 2.3 Robustness test cases for a function of two variables

2.3 Worst-Case Testing

Boundary value analysis, as we said earlier, makes the single fault assumption of reliability theory. Rejecting this 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 worst-case 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 of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 2.4.
Worst case testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst-case test cases. It also represents much more effort: worst-case testing for a function of n variables generates 5^n test cases, as opposed to 4n + 1 test cases for boundary value analysis.

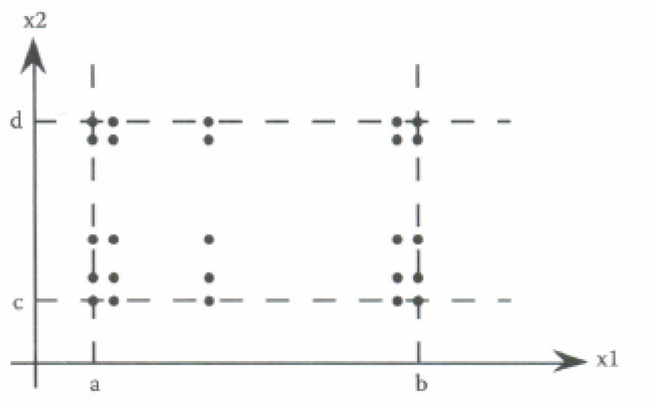


Figure 2.4 Worst-case test cases for a function of two variables.

Worst-case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst-case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, 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 2.5 shows the robust worst-case test cases for our two variable functions.

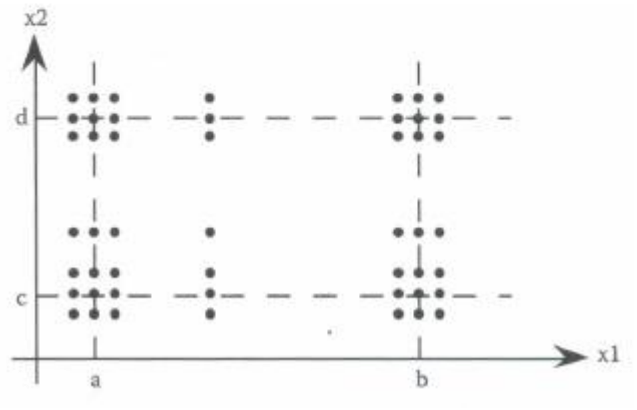


Figure 2.5 Robust worst-case test cases for a function of two variables.

2.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and least uniform. Special value testing occurs when a tester uses domain knowledge, experience with similar programs, and information about "soft spots" to devise test cases.

We might also call this ad hoc testing or "seat-of-the-pants” testing. No guidelines are used other than to use "best engineering judgment." As a result, special value testing is very dependent on the abilities of the tester.
Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for three of our examples.
If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. If an interested tester defined special value test cases for NextDate, we would see several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test sets generated by the other methods we have studied—testimony to the craft of software testing.

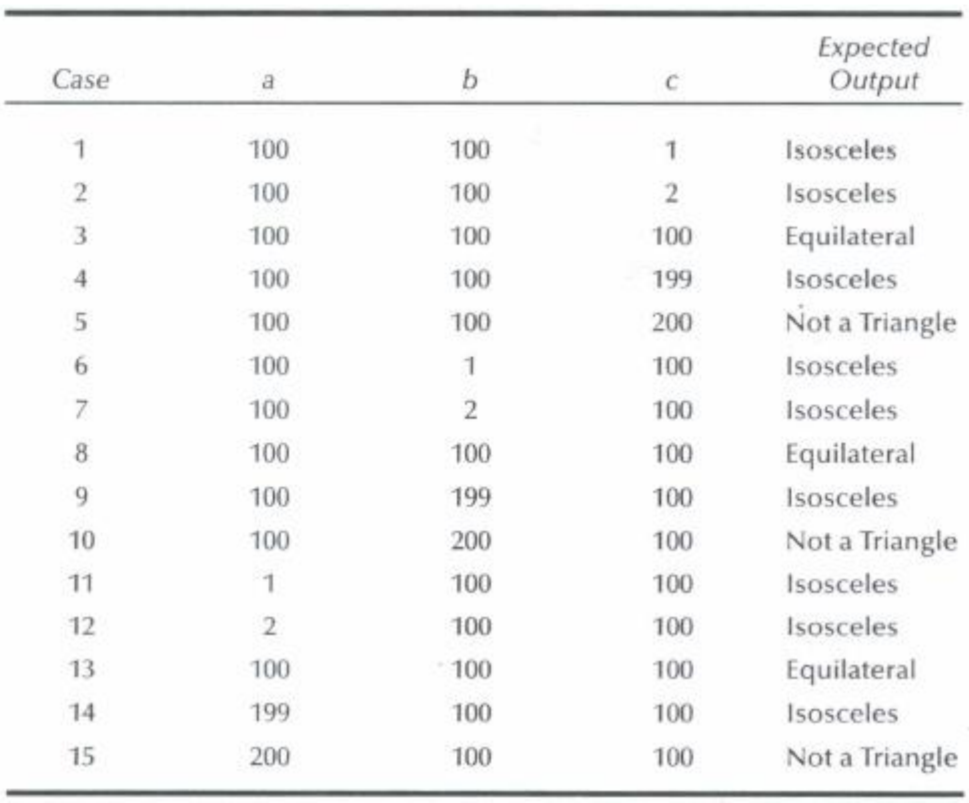
2.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we have just selected examples.

5.5.1 Test Cases for the Triangle Problem

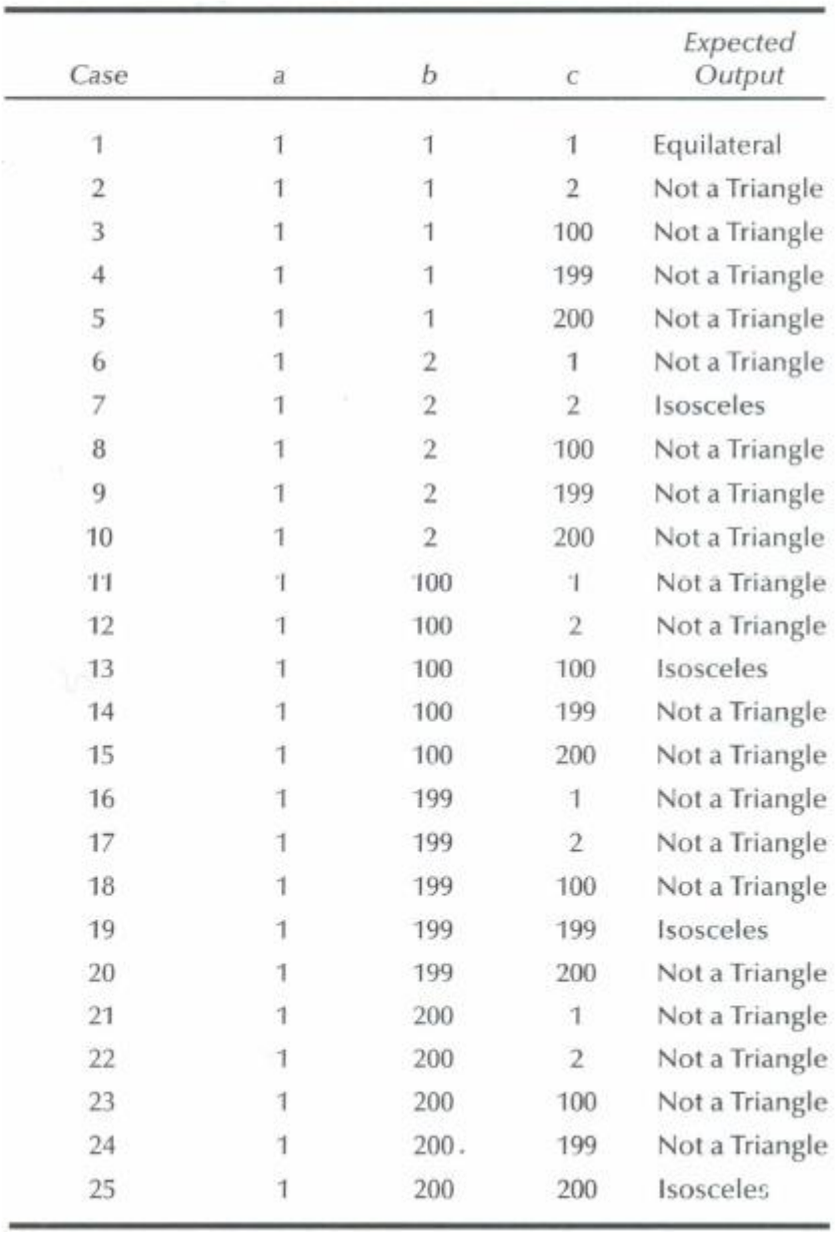
In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. Table 2.1 contains boundary value test cases using these ranges. Notice that test-cases 3, 8, and 13 are identical so two should be deleted. Table 2.2 shows the worst case test cases in just "one corner" of the input space cube.

Table 2.1 Triangle Problem Boundary Value Analysis 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

Table 2.2 Triangle Problem Worst-Case Test Cases

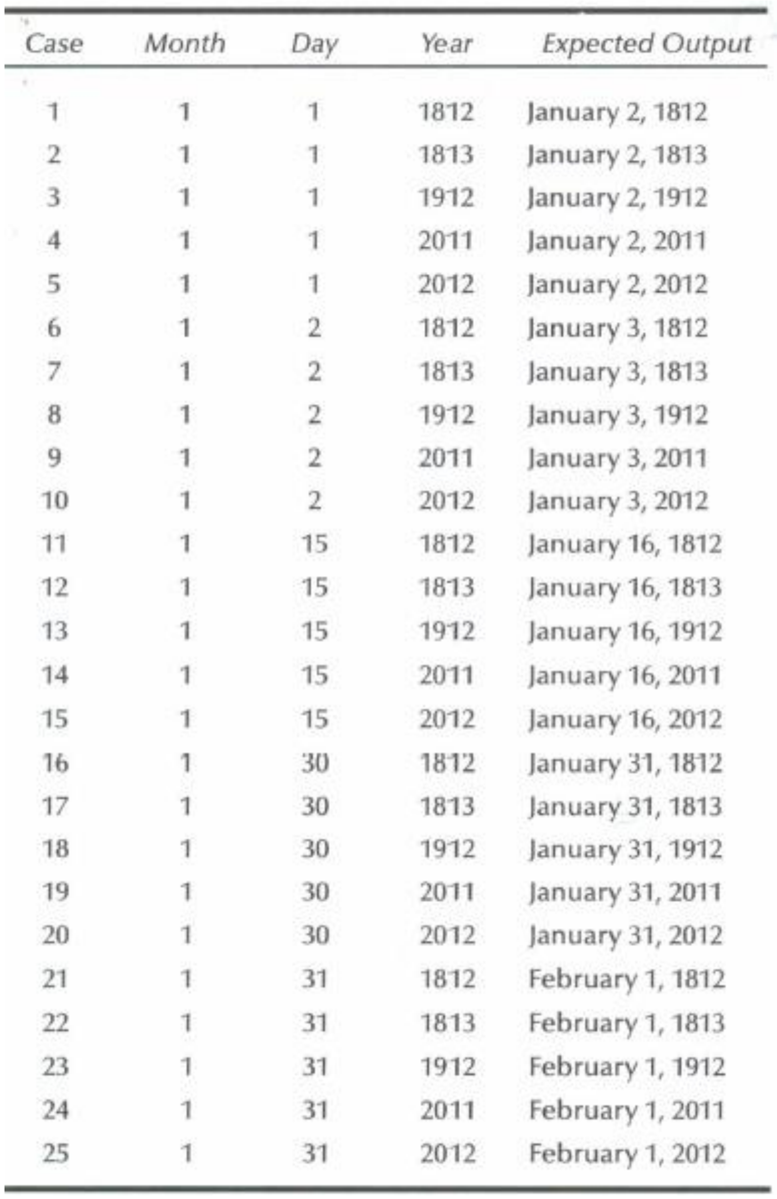


Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not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

2.5.2 Test Cases for the NextDate Function

Table 2.3 contains the worst case test cases for the NextDate function. As before only one corner of the input space cube B is shown.

Table 5.3 NextDate Worst-Case Test Cases



Case	Month	Day	Year	Expected Output
1	1	1	1812	January 2, 1812
2	1	1	1813	January 2, 1813
3	1	1	1912	January 2, 1912
4	1	1	2011	January 2, 2011
5	1	1	2012	January 2, 2012
6	1	2	1812	January 3, 1812
7	1	2	1813	January 3, 1813
8	1	2	1912	January 3, 1912
9	1	2	2011	January 3, 2011
10	1	2	2012	January 3, 2012
11	1	15	1812	January 16, 1812
12	1	15	1813	January 16, 1813
13	1	15	1912	January 16, 1912
14	1	15	2011	January 16, 2011
15	1	15	2012	January 16, 2012
16	1	30	1812	January 31, 1812
17	1	30	1813	January 31, 1813
18	1	30	1912	January 31, 1912
19	1	30	2011	January 31, 2011
20	1	30	2012	January 31, 2012
21	1	31	1812	February 1, 1812
22	1	31	1813	February 1, 1813
23	1	31	1912	February 1, 1912
24	1	31	2011	February 1, 2011
25	1	31	2012	February 1, 2012

2.5.3 Test Cases for the Commission Problem

We will look at boundary values for the output range, especially near the threshold points of $1000 and $1800. The output space of the commission is shown in Figure 2.6. The intercepts of these threshold planes with the axes are shown.

Table 2.4 Commission Problem Output Boundary Value Analysis Test Cases

Case	Locks	Stocks	Barrels	Sales	Commission	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

Table 2.4 shows the worst case test cases in the output space for the commission problem. These test cases exercise the points at which the commission percentage changes. The volume below the lower plane corresponds to sales below the $1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test-cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the boundary values: $100, $1000, $1800, and $7800. These test cases were developed with a spreadsheet, which' saves a lot of calculator pecking. The minimum and maximum were easy, and the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the $1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6 to 8 and 10 to 12). If we wanted to, we could pick values near the intercepts such as (22, 1, 1) and (21, 1, 1). As we continue in this way, we have a sense that we are exercising interesting parts of the code.

We might claim that this is really a form of special value testing, because we used our mathematical insight to generate test cases. Table 2.5 contains some typical special value test cases for the commission problem.

Table 2.5 Output Special Value Test Cases

Case	Locks	Stocks	Barrels	Sales	Commission	Comment
1	10	11	9	1005	100.75	Border point +
2	18	17	19	1795	219.25	Border point –
3	18	19	17	1805	221	Border point +

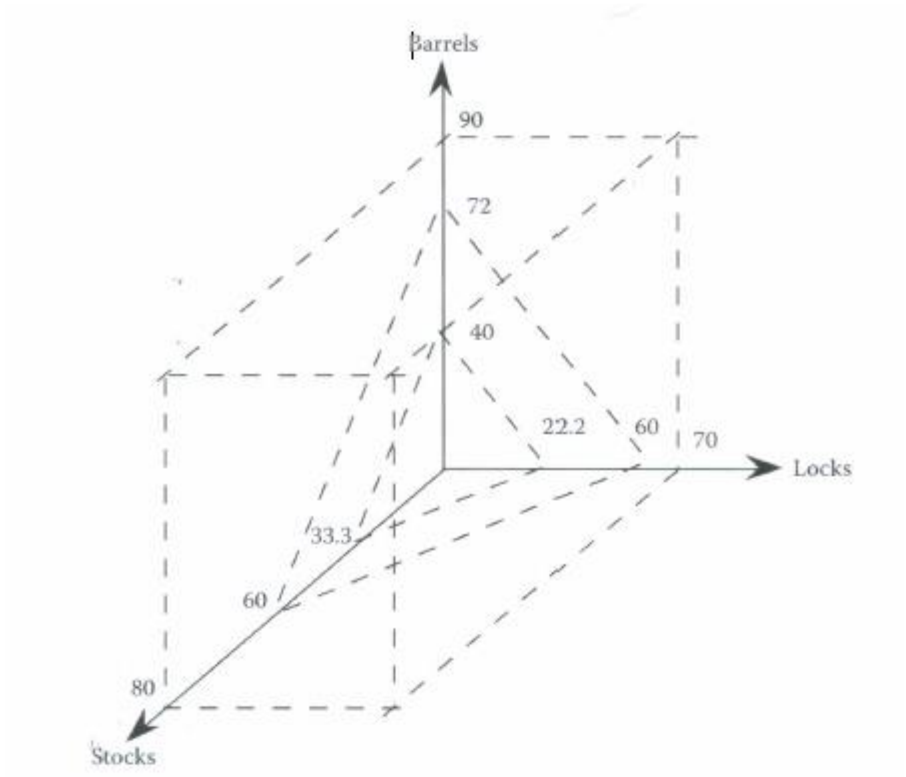


Figure 2.6 Input space of the commission problem.

Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations: we would like to have a sense of complete testing and, at the same time, we would hope to avoid redundancy.

Equivalence class testing echoes the two deciding factors of boundary value testing, robustness, and the single/multiple fault assumptions

2.6 Equivalence Classes

- Equivalence classes 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 is represented provides a form of completeness, and the disjointedness ensures a form of non-redundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common.
- 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.
- **Ex:** In the triangle problem, 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). Our intuition tells us that these would be treated the same as the first test case; thus, they would be redundant.
- The key (and the craft) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by second-guessing the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation.
- A function, F, of two variables, X1 and X2, 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. The intervals presumably correspond to some distinction in the program being tested, for example, the commission ranges in the commission problem. Invalid values of X1 and X2 are X1 < a, X1 > d and X2 < e, X2 > g.

2.6.1 Weak Normal Equivalence Class Testing

With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. For the previous example, we would end up with the weak equivalence class test cases shown in Figure 6.1.
These three test cases use one value from each equivalence class. We identify these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets.

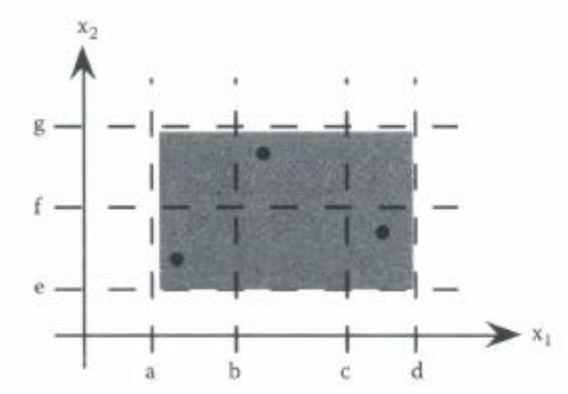


Figure 6.1 Weak normal equivalence class test cases.

2.6.2 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 6.2. Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. 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.

The key to good equivalence class testing is the selection of the equivalence relation. Most of the time, equivalence class testing defines classes of the input domain. There is-no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the triangle problem.

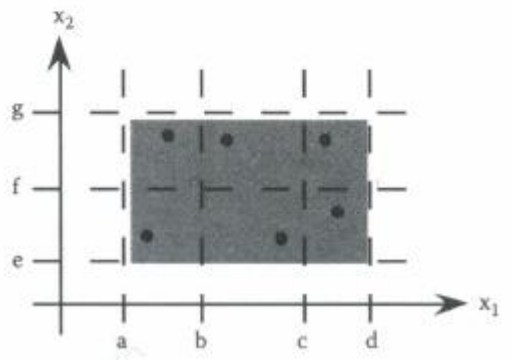


Figure 6.2 Strong normal equivalence class test cases

2.6.3 WeakRobust Equivalence Class Testing

The robust part comes from consideration of invalid values, and the weak part refers to the single fault assumption.

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.

The test cases resulting from this strategy are shown in Figure 6.3.

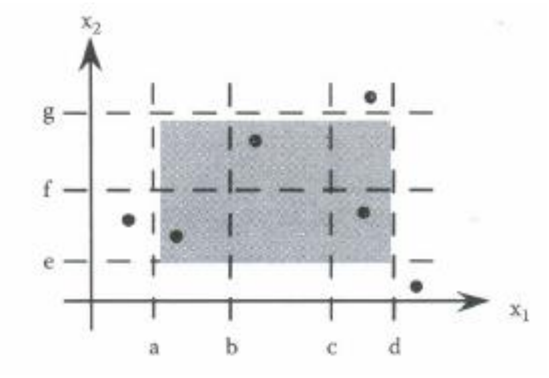


Figure 6.3 Weak robust equivalence class test cases.

Two problems occur with robust equivalence testing.

- The first is that, the specification does not define what the expected output for an invalid input should be. Thus, testers spend a lot of time defining expected outputs for these cases.
- The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as- FORTRAN and COBOL were dominant; thus, this type of error was common.

2.6.4 Strong Robust Equivalence Class Testing

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, as shown in Figure 6.4.

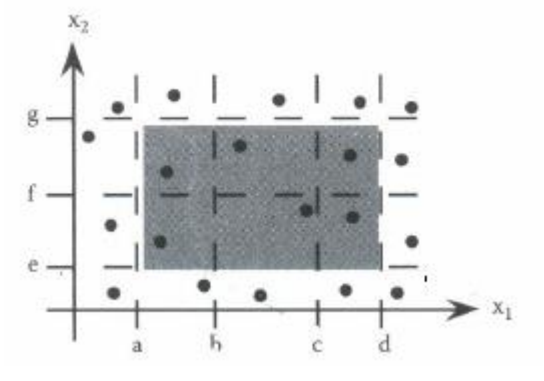


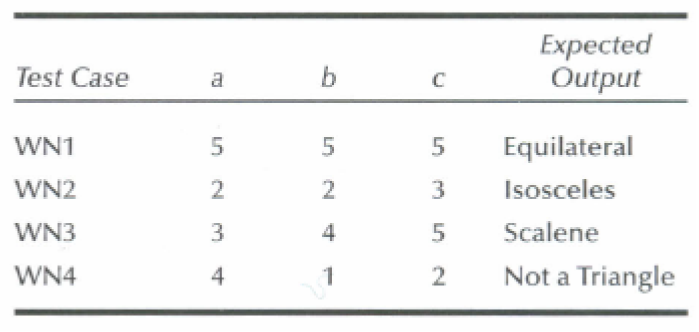
Figure 6.4 Strong robust equivalence class test cases.

2.7 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: Not a Triangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows:

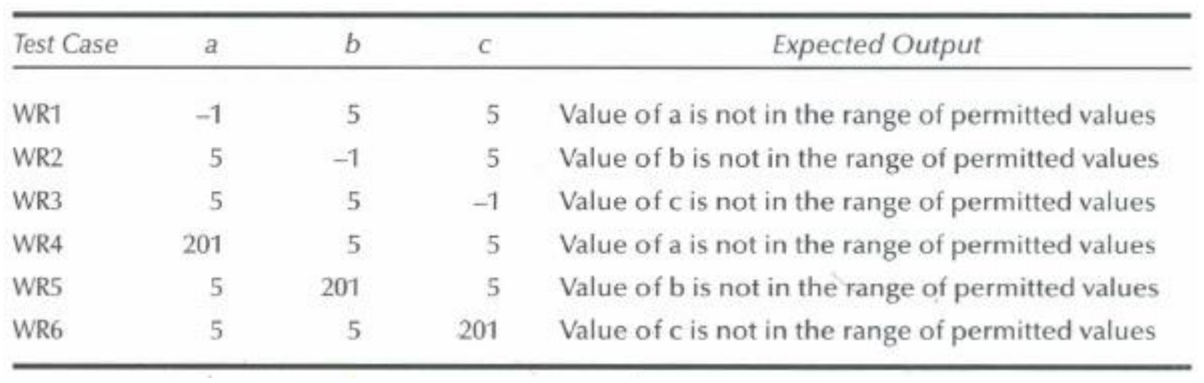
R1 = {<a, b, c> : the triangle with sides a, b, and c is equilateral}
R2 = {<a, b, c> : the triangle with sides a, b, and c is isosceles}
R3 = {<a, b, c> : the triangle with sides a, b, and c is scalene}
R4 = {<a, b, c> : sides a, b, and c do not form a triangle}

Four weak normal equivalence class test cases, chosen arbitrarily from each class, are:



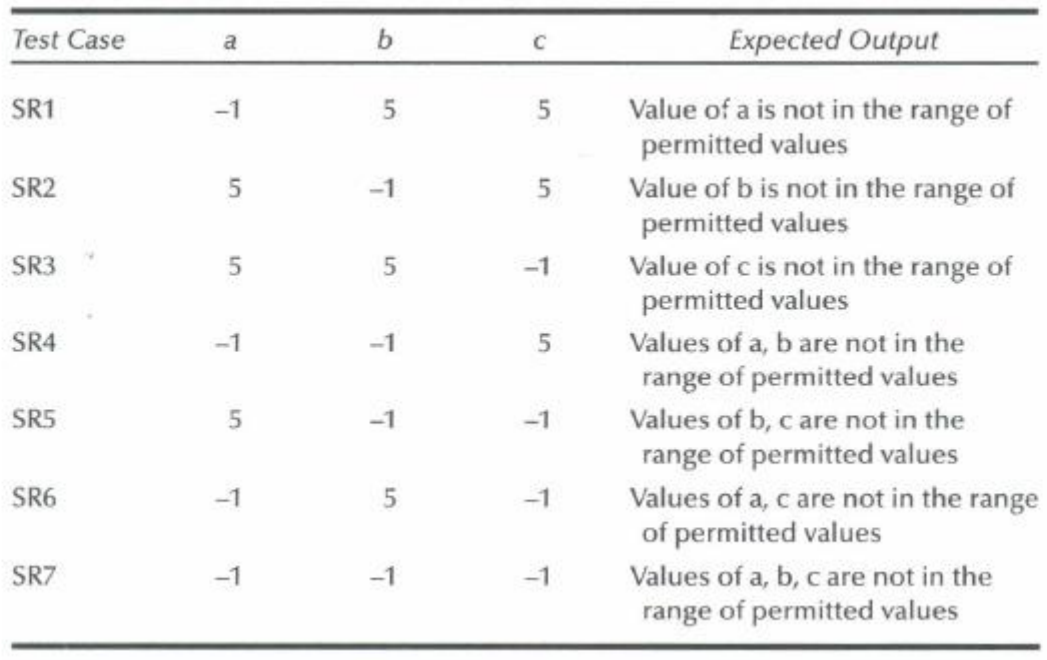
Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Triangle

Because no valid subintervals of variables a, b, and c exist, the strong normal equivalence class test 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.



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 3-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

Notice how thoroughly the expected outputs describe the invalid input values.
Equivalence class testing is clearly sensitive to the equivalence relation used to define classes. Here is another instance of craftsmanship. If we base equivalence classes on the input domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen in three ways), or none can be equal:
D1 = {<a, b, c> : a = b = c}
D2 = {<a, b, c> : a -= b, a ≠ c}
D3 = {<a, b, c> : a = c, a ≠ b}
D4 = {<a, b, c> : b = c, a ≠ b}
D5 = {<a, b, c> : a ≠ b, a ≠ c, b ≠ c}

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet <1, 4, 1> has exactly one pair of equal sides, but these sides do not form a triangle.)

D6 = {<a, b, c> : a ≥ b + c}
D7 = {<a, b, c> : b ≥ a + c}
D8 = {<a, b, c> : c ≥ a + b}

If we wanted to be even more thorough, we could separate the "greater than or equal to" into two distinct cases; thus, the set D6 would become:

D6' = {<a, b, c> : a = b + c}
D6" = {<a, b, c> : a > b + c} and similarly for D7 and D8

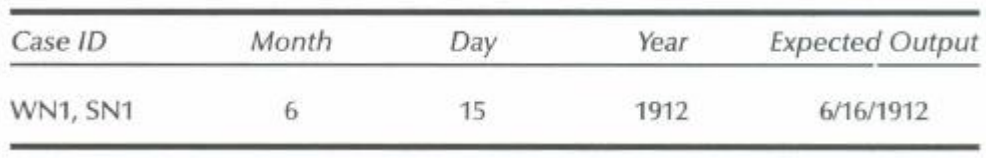
2.8 Equivalence Class Test Cases for the NextDate Function

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 ≤ month ≤ 12}
D1 = {day: 1 ≤ day ≤ 31}
Y1 = {year: 1812 ≤ year ≤ 2012}

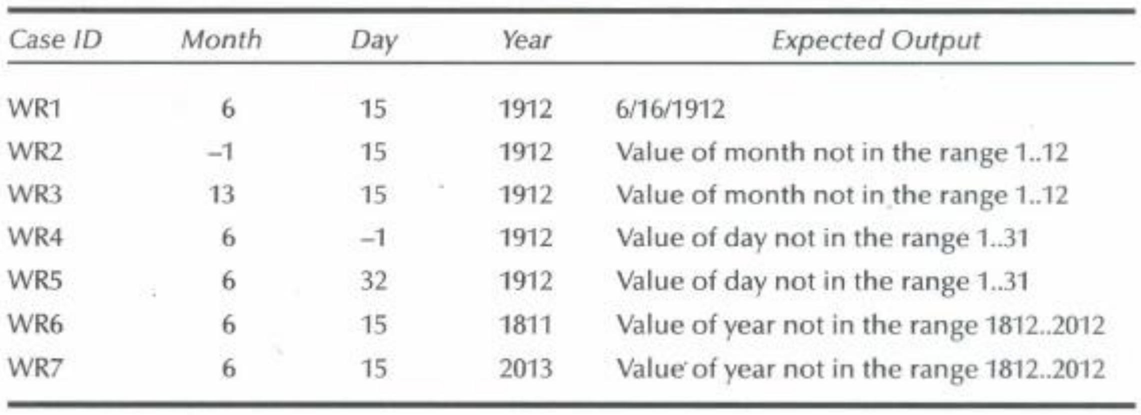
The invalid equivalence classes are:
M2 = {month: month < 1}
M3 = {month: month > 12}
D2 = {day: day < 1}
D3 = {day: day> 31}
Y2 = {year: year < 1812}
Y3 = {year: year > 2012}

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



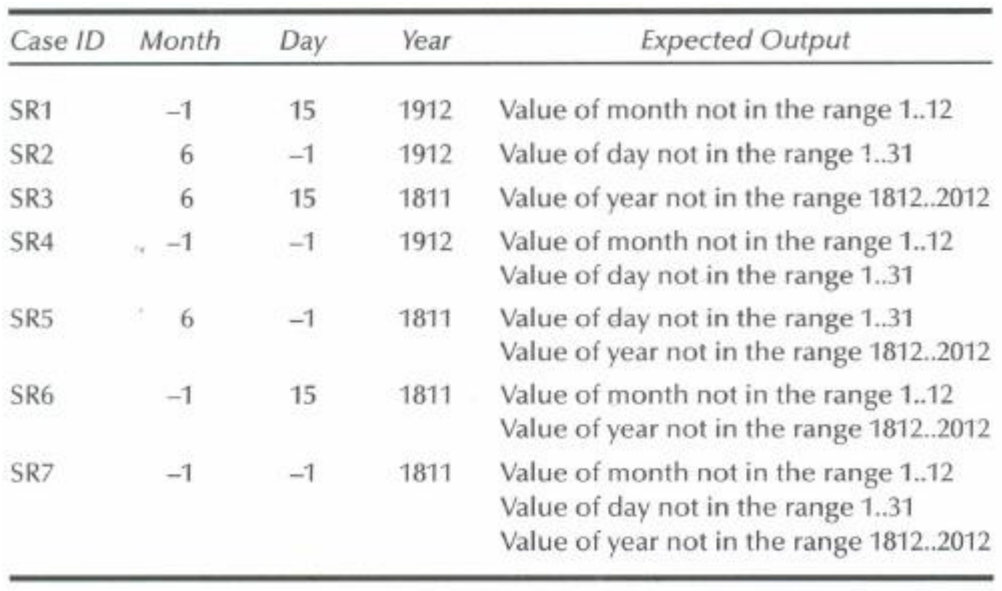
Case ID	Month	Day	Year	Expected Output
WN1, SN1	6	15	1912	6/16/1912

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

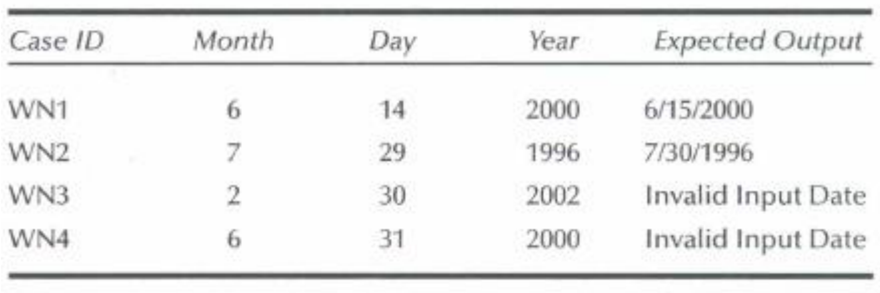
As with the triangle problem, here is one corner of the cube in 3-space of the additional strong robust equivalence class test cases:



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

2.8.1 Equivalence Class Test Cases

These classes yield the following weak 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

Mechanical selection of input values makes no consideration of our domain knowledge thus the two impossible dates. This will always be a problem with automatic test case generation, because all of our domain knowledge is not captured in the choice of equivalence classes. The strong normal equivalence class test cases for the revised classes are:

Case ID	Month	Day	Year	Expected Output
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996
SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	Invalid Input Date
SN8	6	30	1996	Invalid Input Date
SN9	6	30	2002	Invalid Input Date
SN10	6	31	2000	Invalid Input Date
SN11	6	31	1996	Invalid Input Date
SN12	6	31	2002	Invalid Input Date
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/30/2002
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	Invalid Input Date
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

Moving from weak to strong normal testing raises some of the issues of redundancy Three month classes times four day classes times three year classes results in 36 strong normal equivalence class test cases. Adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases

2.9 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 input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be an improvement.

The valid classes of the input variables are:

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

The corresponding invalid classes of the input variables are:

L3	$= \{\text{locks} : \text{locks} = 0 \text{ OR locks} < -1\}$
L4	$= \{\text{locks} : \text{locks} > 70\}$
S2	$= \{\text{stocks} : \text{stocks} < 1\}$
S3	$= \{\text{stocks} : \text{stocks} > 80\}$
B2	$= \{\text{barrels} : \text{barrels} < 1\}$
B3	$= \{\text{barrels} : \text{barrels} > 90\}$

One problem occurs, however. The variable locks are also used as a sentinel to indicate no more telegrams. When a value of -1 is given for locks, the While loop terminates, and the values of totallocks, totalStocks, and totalBarrels are used to compute sales, and then commission. Except for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. Therefore, we will have exactly one weak normal equivalence class test case - and again, it is identical to the strong normal equivalence class test case. Note that the case for locks = -1 just terminates the iteration. 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

Finally, a corner of the cube will be 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 Value of Stocks not in the range 1..80
SR5	–2	40	–1	Value of Locks not in the range 1..70 Value of Barrels not in the range 1..90
SR6	35	–1	–1	Value of Stocks not in the range 1..80 Value of Barrels not in the range 1..90
SR7	–2	–1	–1	Value of Locks not in the range 1..70 Value of Stocks not in the range 1..80 Value of Barrels not in the range 1..90

2.9.1 Output Range Equivalence Class Test Cases

Notice that of strong test cases-whether normal or robust-only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks, and barrels sold:

sales = 45 X locks + 30 X stocks + 25 X barrels

We could define equivalence classes of three variables by commission ranges:

51 = {clocks, stocks, barrels> : sales ~ 1000}
52 = [clocks, stocks, barrels> : 1000 < sales ~ 1800}
53 = [clocks, stocks, barrels> : sales> 1800}

Figure 2.6 helps us get a better feel for the input space. Elements of 51 are points with integer coordinates in the pyramid near the origin. Elements of 52 are points in the triangular slice between the pyramid and the rest of the input space. Finally, elements of 53 are all those points in the rectangular volume that are not in 51 or 52. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Fig 2.6. As was the case with the triangle problem, 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

Decision Table-Based Testing

7.1 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Table 7.l.

A decision table has four portions: the left-most column is the stub portion; to the right is the entry portion. The condition portion is noted by c's, the action portion is noted by a's. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries.

A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. In the decision table in Table 7.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true and c3 is false, actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a "don't care" entry. The don’t care entries have two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the "n/a" symbol for this latter interpretation.

When we have binary conditions (true/false, yes/no, 011), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing. Decision tables in which all the conditions are binary are called limited entry decision tables. If conditions are allowed to have several values, the resulting tables are called extended entry decision tables.

Table 7.1 Portions of a Decision Table

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4*	*Rule 5*	*Rule 6*	*Rules 7, 8*
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

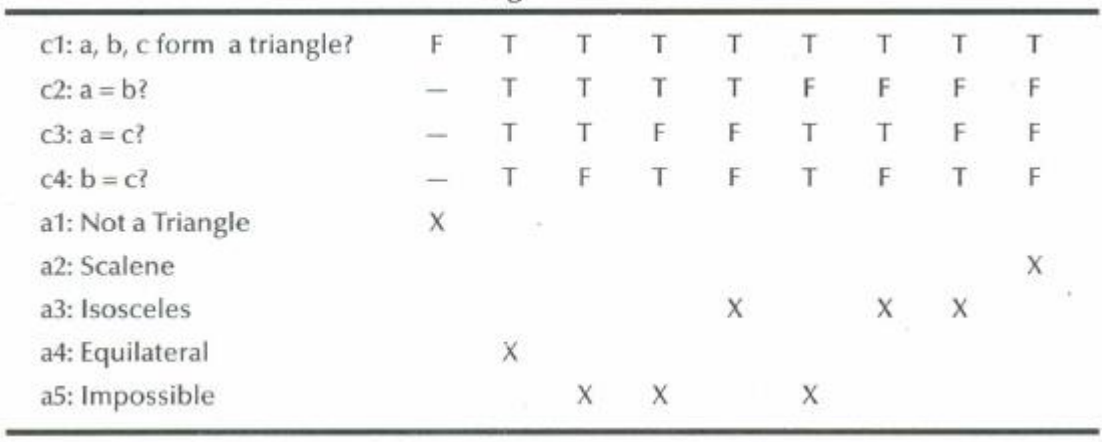
7.1.1 Technique

To identify test cases with decision tables, we interpret conditions as inputs and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be complete, we know we have a comprehensive set of test cases.

Several techniques that produce decision tables are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible.

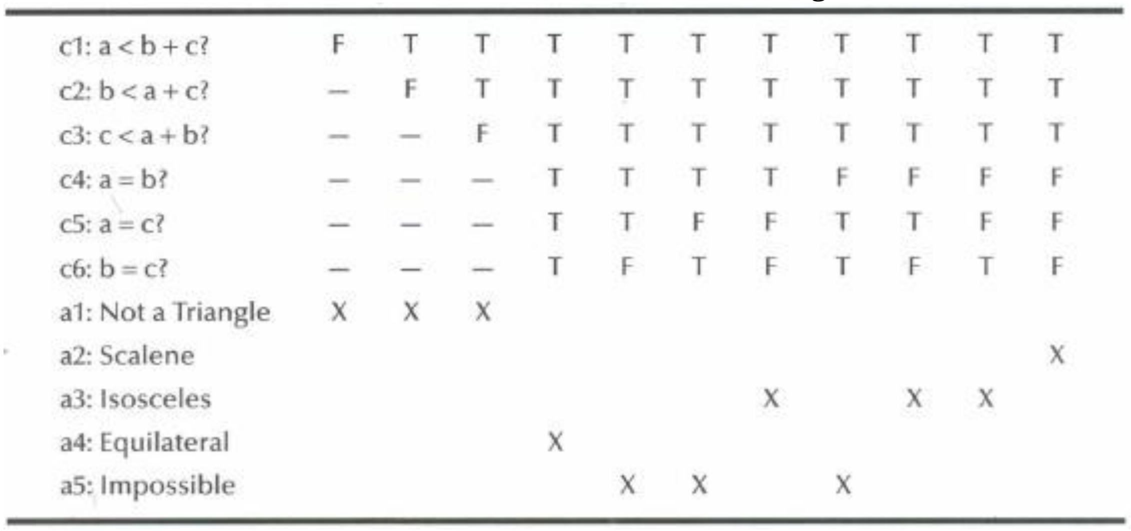
In the decision table in Table 7.2, we see examples of don't care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we do not even care about possible equalities, as indicated in the first rule. In rules 3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal; thus, the negative entry makes these rules impossible.

Table 7.2 Decision Table for the Triangle Problem



c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a Triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Table 7.3 Refined Decision Table for the Triangle Problem



c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a Triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

The decision table in Table 7.3 illustrates another consideration related to technique: the choice of conditions can greatly expand the size of a decision table. Here, we expanded the old condition (c1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If anyone of these fails, the three integers do not constitute sides of a triangle. We could expand this still further because there are two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater.

When conditions refer to equivalence classes, decision tables have a characteristic appearance.

Conditions in the decision table in Table 7.4 are from the NextDate problem; they refer to the mutually exclusive possibilities for the month variable. Because a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The don't care entries (-) really mean "must be false." Some decision table aficionados use the notation F! to make this point.

Use of don't care entries has a subtle effect on the way in which complete decision tables are recognized. For limited entry decision tables, if n conditions exist, there must be Z" rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows rules in which no don't care entries occur count as one rule, and each don't care entry in a rule doubles the count of that rule. The rule counts for the decision table in Table 7.3 are shown in Table 7.5. Notice that the sum of the rule counts is 64 (as it should be).

Table 7.4 Decision Table with Mutually Exclusive Conditions

Conditions	R1	R2	R3
c1: month in M1?	T	—	—
c2: month in M2?	—	T	—
c3: month in M3?	—	—	T
a1			
a2			
a3			

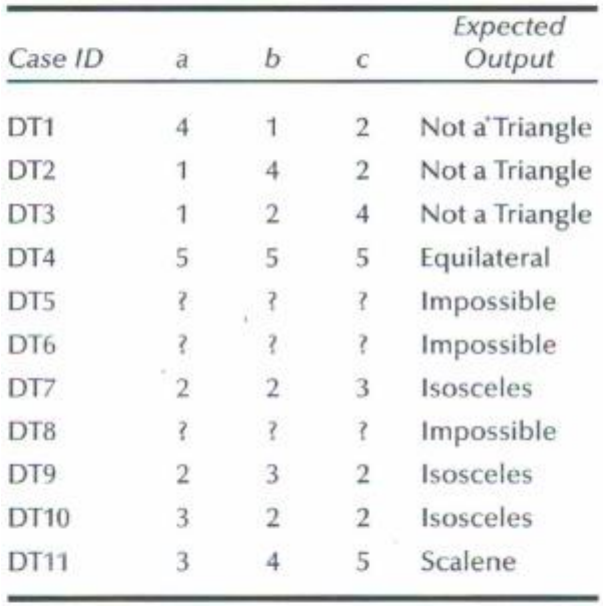
Table 7.5 Decision Table for Table 7.3 with Rule Counts

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a Triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

7.2 Test Cases for the Triangle Problem

Using the decision table in Table 7.3, we obtain 11 functional test cases: 3 impossible cases, 3 ways to fail the triangle property, 1way to get an equilateral triangle, 1way to get a scalene triangle, and 3 ways to get an isosceles triangle (see Table 7.l1). If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases. Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries and more impossible rules.

Table 7.11 Test Cases from Table 7.3



Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

7.3 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table-based testing, because decision tables can highlight such dependencies. The equivalence classes in the input domain of the NextDate function. One of the limitations we found is that indiscriminate selection of input values from the equivalence classes resulted in "strange" test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian product of the classes makes sense.

When logical dependencies exist among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian product. The decision table format lets us emphasize such dependencies using the notion of the "impossible" action to denote impossible combinations of conditions (which are actually impossible rules). In this section, we will make three tries at a decision table formulation of the NextDate function.

7.3.1 First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \leq$ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is not a leap year}

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

a1: Day invalid for this month
a2: Cannot happen in a non-leap year
a3: Compute the next date

Table 7.12 First Try Decision Table with 256 Rules

Conditions			
c1: month in M1?	T		
c2: month in M2?		T	
c3: month in M3?			T
c4: day in D1?			
c5: day in D2?			
c6: day in D3?			
c7: day in D4?			
c8: year in Y1?			
a1: impossible			
a2: next date			

7.3.2 Second Try

If we focus on the leap year aspect of the NextDate function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian product that contains 36 entries (test cases), with several that are impossible. To illustrate another decision table technique, this time we will develop an extended entry decision table, and we will take a closer look at the action stub. In making an extended entry decision table, we must ensure that the equivalence classes form a true partition of the input domain.

If there were any overlaps among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y2 is the set of years between 1812 and 2012, evenly divisible by four, excluding the year 2000:

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: 1 ≤ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year = 2000}
Y2 = {year: year is a non-century leap year}
Y3 = {year: year is a common year}

Table 7.13 Second Try Decision Table with 36 Rules

	1	2	3	4	5	6	7	8
c1: month in	M1	M1	M1	M1	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D1	D2	D3	D4
c3: year in	—	—	—	—	—	—	—	—
Rule count	3	3	3	3	3	3	3	3
Actions								
a1: impossible				X				
a2: increment day	X	X			X	X	X	
a3: reset day			X					X
a4: increment month			X					?
a5: reset month								?
a6: increment year								?

	9	10	11	12	13	14	15	16
c1: month in	M3	M3	M3	M3	M3	M3	M3	M3
c2: day in	D1	D1	D1	D2	D2	D2	D3	D4
c3: year in	Y1	Y2	Y3	Y1	Y2	Y3	—	—
Rule count	1	1	1	1	1	1	3	3
Actions								
a1: impossible						X	X	X
a2: increment day		X						
a3: reset day	X		X	X	X			
a4: increment month	X		X	X	X			
a5: reset month								
a6: increment year								

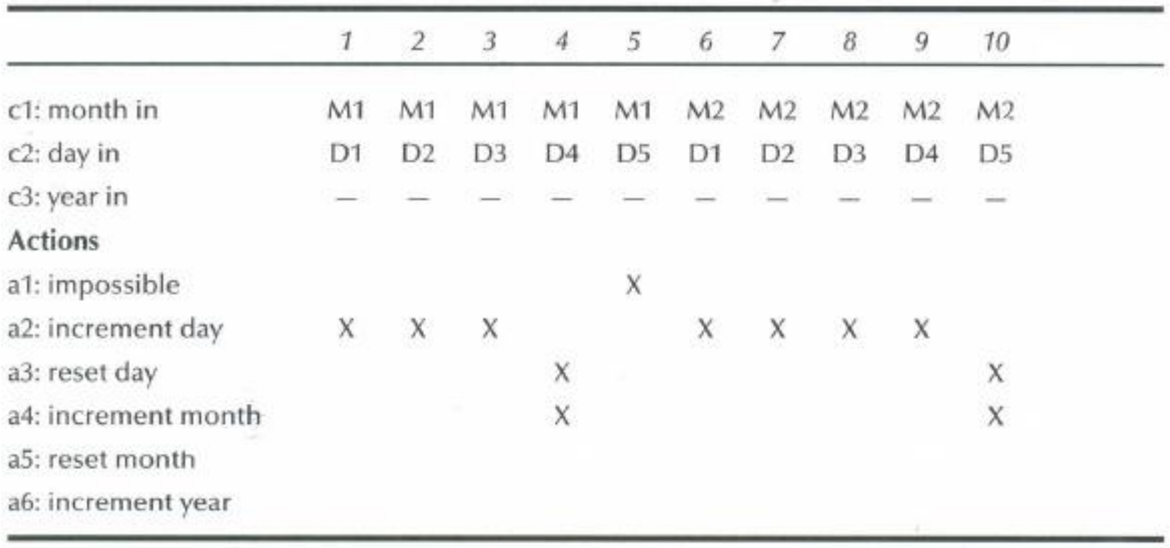
7.3.3 Third Try

We can clear up the end-of-year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap year or non-leap year condition of the first try-so the year 2000 gets no special attention.

Ml = {month: month has 30 days}
M2 = {month: month has 31 days except December}
M3 = {month: month is December}
M4 = {month: month is February}
D1 = {day: $1 \leq$ day ≤ 27}
D2 = {day: day = 28}
D3 = {day: day= 29}
D4 = {day: day = 30}
D5 = {day: day;' 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is a common year}

The Cartesian product of these contains 40 elements. The result of combining rules with don't care entries is given in Table 7.14; it has 22 rules, compared with the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set.

Table 7.14 Decision Table for the NextDate Function



	1	2	3	4	5	6	7	8	9	10
c1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
c3: year in	—	—	—	—	—	—	—	—	—	—
Actions										
a1: impossible					X					
a2: increment day	X	X	X			X	X	X	X	
a3: reset day				X						X
a4: increment month				X						X
a5: reset month										
a6: increment year										

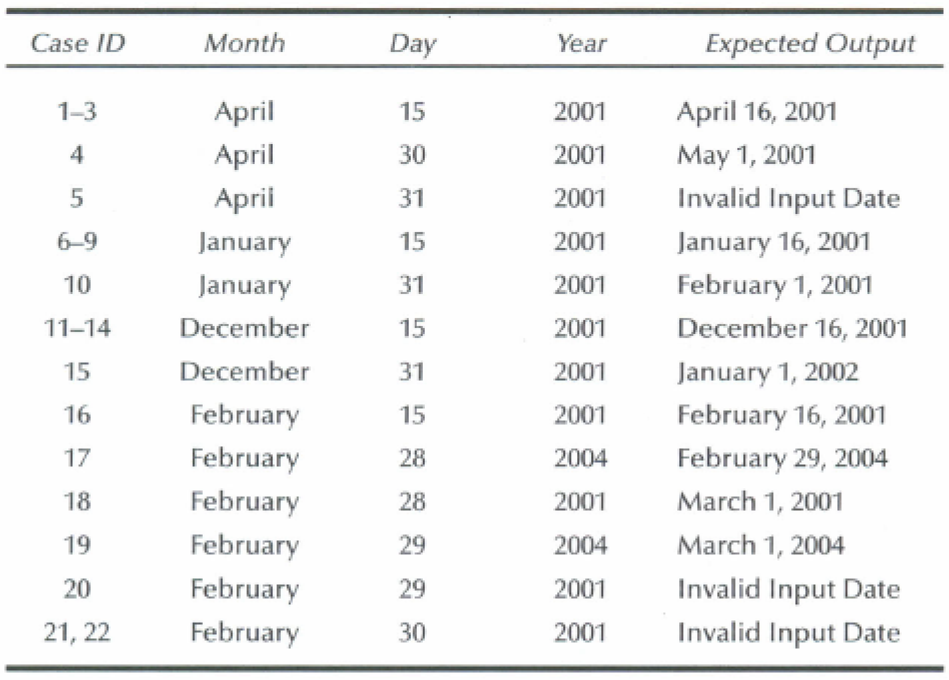
	11	12	13	14	15	16	17	18	19	20	21	22
c1: month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
Actions												
a1: impossible										X	X	X
a2: increment day	X	X	X	X		X	X					
a3: reset day					X			X	X			
a4: increment month								X	X			
a5: reset month					X							
a6: increment year					X							

Table 7.15 Reduced Decision Table for the NextDate Function

	1–3	4	5	6–9	10
c1: month in	M1	M1	M1	M2	M2
c2: day in	D1, D2, D3	D4	D5	D1, D2, D3, D4	D5
c3: year in	—	—	—	—	—
Actions					
a1: impossible			X		
a2: increment day	X			X	
a3: reset day		X			X
a4: increment month		X			X
a5: reset month					
a6: increment year					

	11–14	15	16	17	18	19	20	21, 22
c1: month in	M3	M3	M4	M4	M4	M4	M4	M4
c2: day in	D1, D2, D3, D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: year in	—	—	—	Y1	Y2	Y1	Y2	—
Actions								
a1: impossible							X	X
a2: increment day	X		X	X				
a3: reset day		X			X	X		
a4: increment month					X	X		
a5: reset month		X						
a6: increment year		X						

Table 7.16 Decision Table Test Cases for NextDate



Case ID	*Month*	*Day*	*Year*	*Expected Output*
1–3	April	15	2001	April 16, 2001
4	April	30	2001	May 1, 2001
5	April	31	2001	Invalid Input Date
6–9	January	15	2001	January 16, 2001
10	January	31	2001	February 1, 2001
11–14	December	15	2001	December 16, 2001
15	December	31	2001	January 1, 2002
16	February	15	2001	February 16, 2001
17	February	28	2004	February 29, 2004
18	February	28	2001	March 1, 2001
19	February	29	2004	March 1, 2004
20	February	29	2001	Invalid Input Date
21, 22	February	30	2001	Invalid Input Date

Unit 3: Path Testing, Data Flow Testing

Path Testing

The distinguishing characteristic of structural testing methods is that they are all based on the source code of the program tested, and not on the specification.

Program Graph:

- Given a program written in an imperative programming language, the ***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*** iff the statement fragment corresponding to node ***j*** can be executed immediately after the statement fragment corresponding to node ***i***.

The Program graph can be illustrated with the Pseudocode implementation of the triangle program

```
1.    Program triangle 2 'Structured programming version of simpler specification
2.    Dim a, b, c As Integer
3.    Dim IsATriangle As Boolean
          'Step 1: Get Input
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 n,b)
8.    Output ("Side C is ",c)
          'Step'2: Is A Triangle?
9.    If (a < b + c) AND (b < a + c) AND (c < a + b)
```

```
10.         Then IsATriangle True
11.         Else IsATriangle = False
12.     EndIf

        'Step 3: Determine Triangle Type
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 ("Not a Triangle")
22.     EndIf
23.     End triangle2
```

Here, line numbers refer to statements and statement fragments.

In the program graph, non-executable statements such as variable and type declarations are not included.

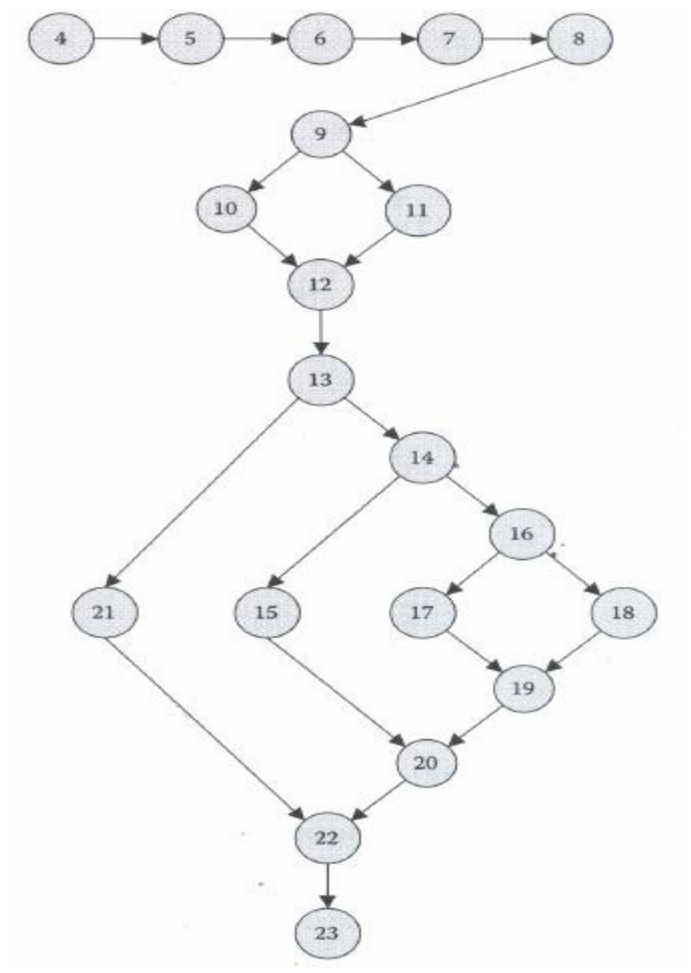


Fig.1: Program graph of the triangle program

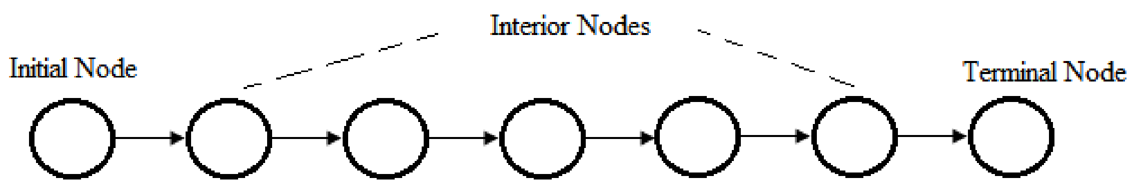
Nodes 4 through 8 are a sequence, nodes 9 through 12 are an if-then-else construct, and nodes 13 through 22 are nested if-then-else constructs. Nodes 4 and 23 are the program source and sink nodes, corresponding to the single-entry, single-exit criteria. No loops exist, so this is a **directed acyclic graph**.

3.2 DD- Path

- Structural testing is based on a construct known as a **decision-to-decision path (DD-Path).** The name refers to a sequence of statements that 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.

- Definition: DD-Paths in terms of paths of nodes in a directed graph (Chains)

  A chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1



The length (Number of edges) of chain is 6 of above figure

Definition: DD-Path

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

Case 1: It consist of a single node with in-degree = 0

Case 2: It consist of a single node with out-degree = 0

Case 3: It consist of a single node with in-degree ≥ 2 or out-degree ≥ 2

Case 4: It consist of a single node with in-degree = 1 and out-degree = 1

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

Cases 1 & 2: Establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-Paths.

Case 3: It deals with complex nodes; it ensures that no node is contained in more than one DD-Path.

Case 4: Needed for short branches; it also preserves the one-fragment, one-DD-Path principle.

Case 5: It is the "normal" case, in which a DD-Path is a single-entry, single-exit sequence of nodes (a chain).

Table 1: Types of DD-Paths of Fig 1

Program Graph Nodes	DD-Path Name	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

Definition DD-Path Graph

Given a program written in an imperative language, the **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.

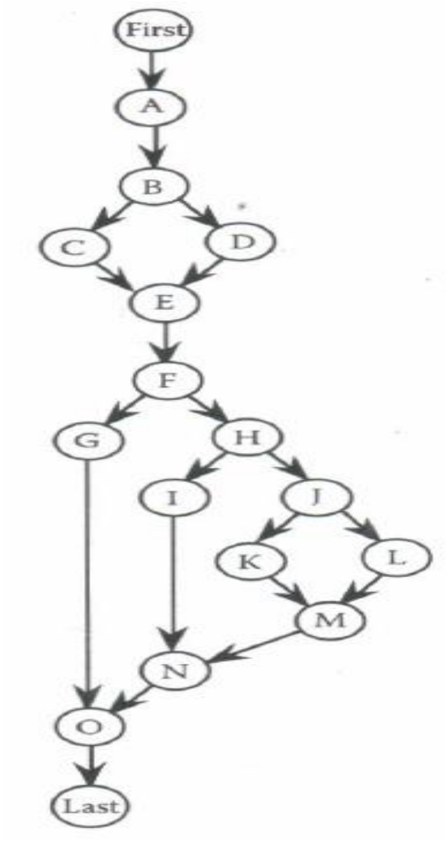


Fig.2: DD-Path graph for the Triangle Program

The DD-Path graph is a form of **condensation** graph in which, 2-connected components are collapsed into individual nodes that correspond to case 5

The single-node DD-Paths (corresponding to cases 1 to 4) are required to preserve the convention that a statement is in exactly one DD-Path.

3.3 Test Coverage Metrics or E.F. Miller 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.

Table: Structural Test Coverage Metrics

Metric	***Description* of *Coverage***
C_0	Every statement

C_1	Every DD-Path (predicate outcome)
C_{1P}	Every predicate to each outcome
C_2	C1 coverage + loop coverage
C_d	C1 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_∞	All possible execution paths

Most quality organizations expect the C_1 metric (DD-Path coverage) as the minimum acceptable level of test coverage. The statement coverage metric (C_0) is less adequate but still widely accepted

These coverage metrics forms a **lattice** in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level while escaping detection by inferior levels of testing.

When DD-Path coverage is attained by a set of test cases, roughly 85% of all faults are revealed.

3.3.1 Metric-Based Testing

Metric based testing takes a closer look on techniques that exercise source code in terms of the test coverage metrics

Note: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas the formulations allows statement fragments to be nodes

1. Statement and Predicate Testing

The formulation allows statement fragments to be individual nodes, the statement and predicate levels (C_0 and C_1) collapse into one consideration.

Example: In triangle problem nodes 9, 10, 11, and 12 are a complete if-then-else statement.

If nodes correspond to **full statements** then execute just one of the decision alternatives and satisfy the statement coverage criterion

If **statement fragments** are considered, then divide such a statement into three nodes. Doing so results in predicate outcome coverage.

Whether or not this convention is followed, these coverage metrics require a set of test cases such that, when executed, every node of the program graph is traversed at least once.

2. DD-Path Testing

When every DD-Path is traversed (the C1 metric), we know that each predicate outcome has been executed. This shows traversing every edge in DD-Path graph.

Example: For **if-then** and **if-then-else** statements, both the true and the false branches are covered (C_{1p} coverage).

For **CASE** statements, each clause is covered.

Longer DD-Paths represent complex computations, which can considered as individual functions. For such DD-Paths, it may be appropriate to apply a number of functional tests.

3. Dependent Pairs of DD-Paths

The dependency among pairs of DD-Paths is the **define/reference** relationship, in which a variable is defined in one DD-Path and is referenced in another DD-Path.

The importance of these dependencies is that they are closely related to the problem of infeasible paths.

Example: In Figure 2, C and H is dependent pair, as are DD-Paths D and H.

The variable **IsATriangle** is set to **TRUE** at node C and **FALSE** at node D. Node H is the branch taken when **IsATriangle** is TRUE in the condition at node B, so any path containing nodes D and H is infeasible.

Dependent pair of DD-Path coverage exercises these dependencies and hence a deeper class of faults are revealed

4. Multiple Condition Coverage

Consider the compound conditions in DD-Paths **B** and **H**. Instead of simply traversing such predicates to their true and false outcomes, better investigate the different ways that each outcome can occur.

One possibility is to make a **Truth Table**; **Example:** Compound condition of three simple conditions would have eight rows, yielding eight test cases.

Another possibility is to **reprogram** compound predicates into nested **simple if-then-else** logic, which will result in more DD-Paths to cover.

Multiple condition coverage ensures that statement complexity versus path complexity is swept.

5. Loop Coverage

Consider the loops such as concatenated, nested, and knotted as shown in Figure 3.

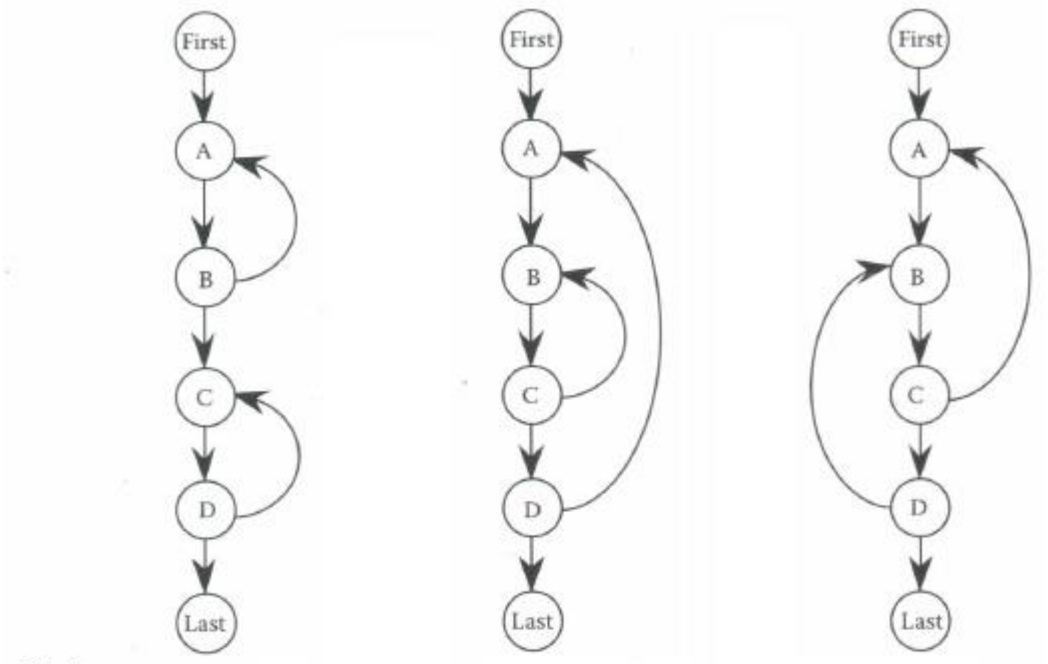


Fig.3: Concatenated loops Nested loops Knotted loops

1. **Concatenated loops:** These are simply a sequence of disjoint loops
2. **Nested loops:** One loop is contained inside another.
3. **Knotted loops**: When it is possible to branch into (or out from) the middle of a loop, and these branches are internal to other loops, the result is Belzer's knotted loop.

The view of loop testing

1. Every loop involves a decision and need to test both outcomes of the decision that is **(1)** Traverse the loop (**2)** The other is to exit the loop.
2. Use a modified boundary value approach to test a loop, where the loop index is given its minimum, nominal & maximum values.

- Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and working outward. This results multiples of test cases because each loop index variable acts like an input variable.
- If loops are knotted, analyze carefully in terms of the dataflow methods.

Test Coverage Analyzers

- Coverage analyzers are a class of test tools that provides automated support for metrics based testing to testing management.

- With a coverage analyzer, the tester runs a set of test cases on a program that has been "instrumented" by the coverage analyzer. The analyzer then uses information produced by the instrumentation code to generate a coverage report.

- **Example:** For DD-Path coverage, the instrumentation identifies and labels all DD-Paths in an original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-Paths traversed by each test case. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set.

3.4 Basis Path Testing

Mathematicians define a basis in terms of a structure called a vector space.

Vector space: Which is a set of elements (vectors) as well as operations that correspond to multiplication and addition defined for the vectors.

The basis of a vector space: The basis of a vector space contains a set of vectors that are independent of one another, and have a spanning property; this means that everything within the vector space can be expressed in terms of the elements within the basis.

What McCabe noticed was that if a basis could be provided for a program graph, this basis could be subjected to rigorous testing; if proven to be without fault, it could be assumed that those paths expressed in terms of that basis are also correct.

3.4.1 McCabe's Basis Path Testing

The method devised by McCabe to carry out basis path testing has four steps. These are:

1. Compute the program graph.
2. Calculate the cyclomatic complexity.
3. Select a basis set of paths.
4. Generate test cases for each of these paths

<u>Workings of McCabe's basis path method</u>

Step 1: To begin, we need a program graph from which to construct a basis. Figure 4 is a directed graph which is the program graph (or the DD-Path graph) of some program. The program does have a single entry (A) and a single exit (G).

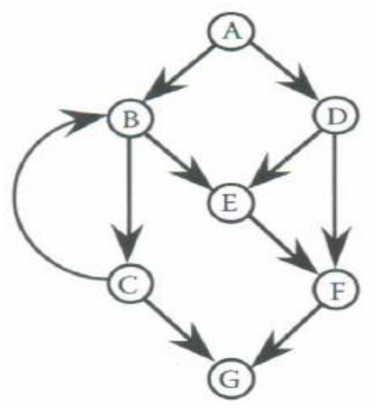


Fig. 4: McCabe's Control Graph

We can create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. Figure 5 shows McCabe's derived strongly connected graph.

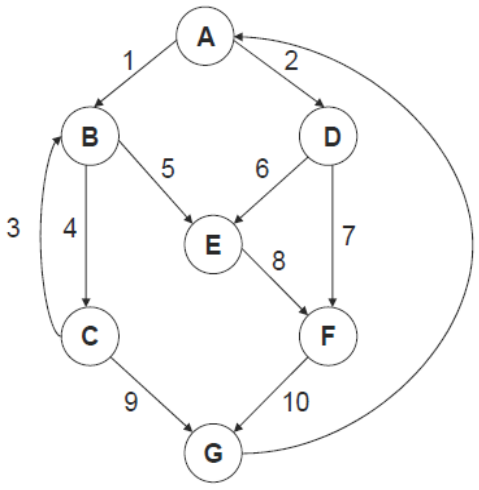


Fig.5: McCabe's strongly connected program graph

Step 2: In graph theory, the cyclomatic complexity is defined for a strongly connected graph is the number of linearly independent circuits in the graph. (**A circuit** is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node.

A circuit is a set of 3-connected nodes.)

The formula for cyclomatic complexity is given by

V(G) = e - n + p or **V(G) = e - n + 2p**

Where, **e:** is the number of edges

n: is the number of nodes

p: is the number of connected regions.

The number of **linearly independent paths** from the source node to the sink node in

Fig. 4 is

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

The number of **linearly independent circuits** in the graph in Fig. 5 is

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

The cyclomatic complexity of the strongly connected graph in Figure 5 is 5. Hence there are five linearly independent circuits. If the added edge is deleted from node G to node A, these five circuits become five linearly independent paths from node A to node G.

Step3: An independent path is any path through the software that introduces at least one new set of processing statements or a new condition.

McCabe's Baseline Method to determine a set of basis paths,

1. Select a "baseline path" that corresponds to normal execution. (The baseline should have as many decisions as possible.)
2. To get next basis paths, 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.
3. Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.

Take the example in Figure 5; here the first path is through nodes A, B, C, B, E, F, and G as the baseline. The first decision node (outdegree ≥ 2) in this path is node A; so for the next basis path, traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path l to be as minimally different as possible. For the next path, follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes Band C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G.

Notice that this set of basis paths is distinct from the one in below Paths: this is not problematic, because a unique basis is not required.

The five linearly independent paths of our graph are as follows:

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

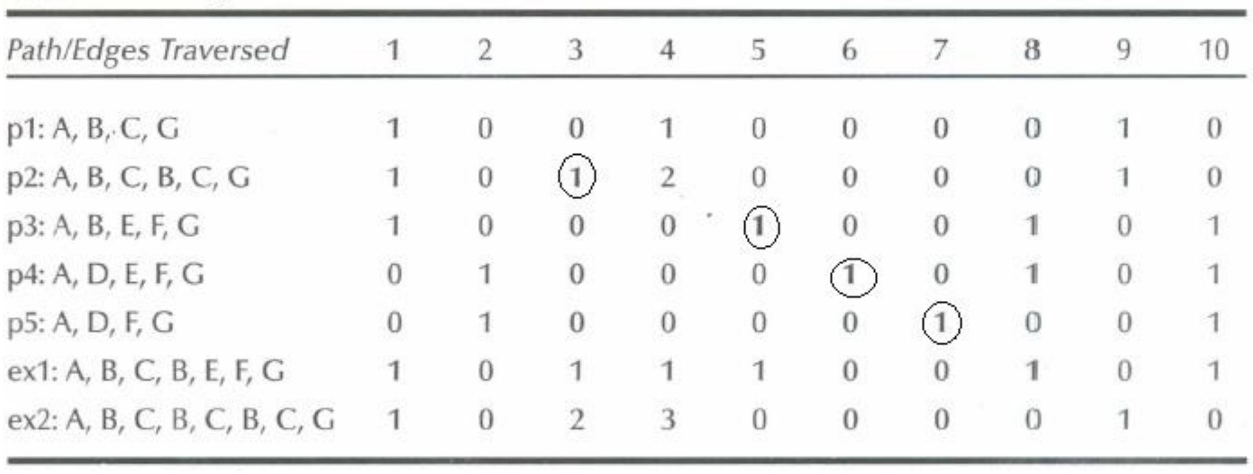
These paths look like a vector space by defining notions of addition and scalar multiplication: path addition is one path followed by another path, and multiplication corresponds to repetitions of a path.

The path A, B, C, B, E, F, G is the basis sum p2 + p3 – p1

The path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1

We can check the independence of paths pl to p5 by examining the first five rows of this incidence matrix. The bold with circled entries show edges that appear in exactly one path, so paths p2 to p5 must be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G.

Table : **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

Observations on McCabe's Basis Path Method

Two major soft spots occur in the McCabe view:

1. Testing the set of basis paths is sufficient (it is not)
2. Need to do with the yoga-like contortions to make program paths look like a vector space.

McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? What does the – p1 part mean? Execute path p1 backward?

To understand these problems, take DD-Path graph of the triangle program in Figure 9.4. The first baseline path that corresponds to a scalene triangle is with sides 3, 4, 5. This test case will traverse the path p1. Now, if we flip the decision at node B, we get path p2. Flip the decision at node F, which yields the path p3. Continue to flip decision nodes in the baseline path p1; the next node with outdegree = 2 is node H. When we node H is flipped we get the path p4. Next flip node J to get p5.

In reality paths p2 and p3 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.

Table: Basis Path in Figure 2

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

One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. For the triangle 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.

Incorporate these solutions with McCabe's baseline method, will yield the following feasible basis path set. Notice that logical dependencies reduce the size of a basis set when basis paths must be feasible.

p1: A-B-C-E-F-H-j-K-M-N-O-Last	Scalene
p6: A-B-O-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

Essential Complexity

Essential complexity, which is only the cyclomatic complexity of yet another form of condensation graph. Condensation graphs are a way of simplifying an existing graph.

The concept behind essential complexity is that the program graph of a piece of software is traversed until a structured programming construct is discovered; once located, the structured programming construct is collapsed into a single node and the graph traversal continues. The desired outcome of this procedure is to end up with a graph of V(G) = 1, that is, a program made up of one node.

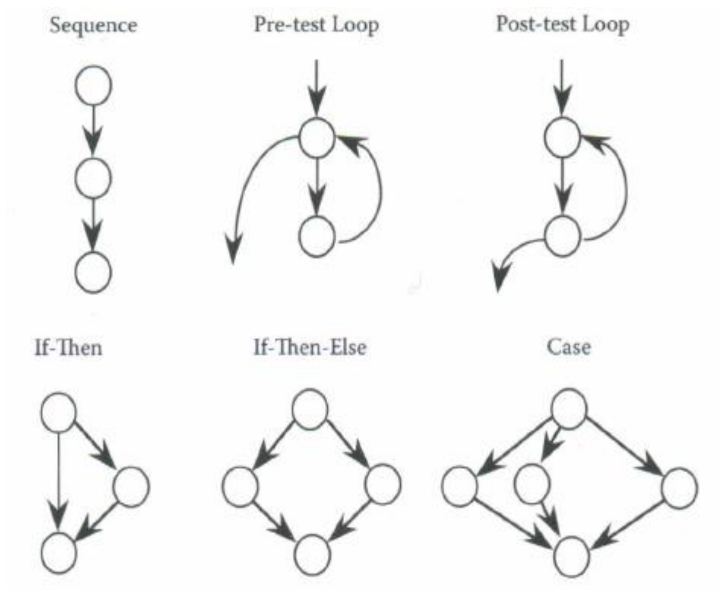


Fig 6: **Structured programming constructs**

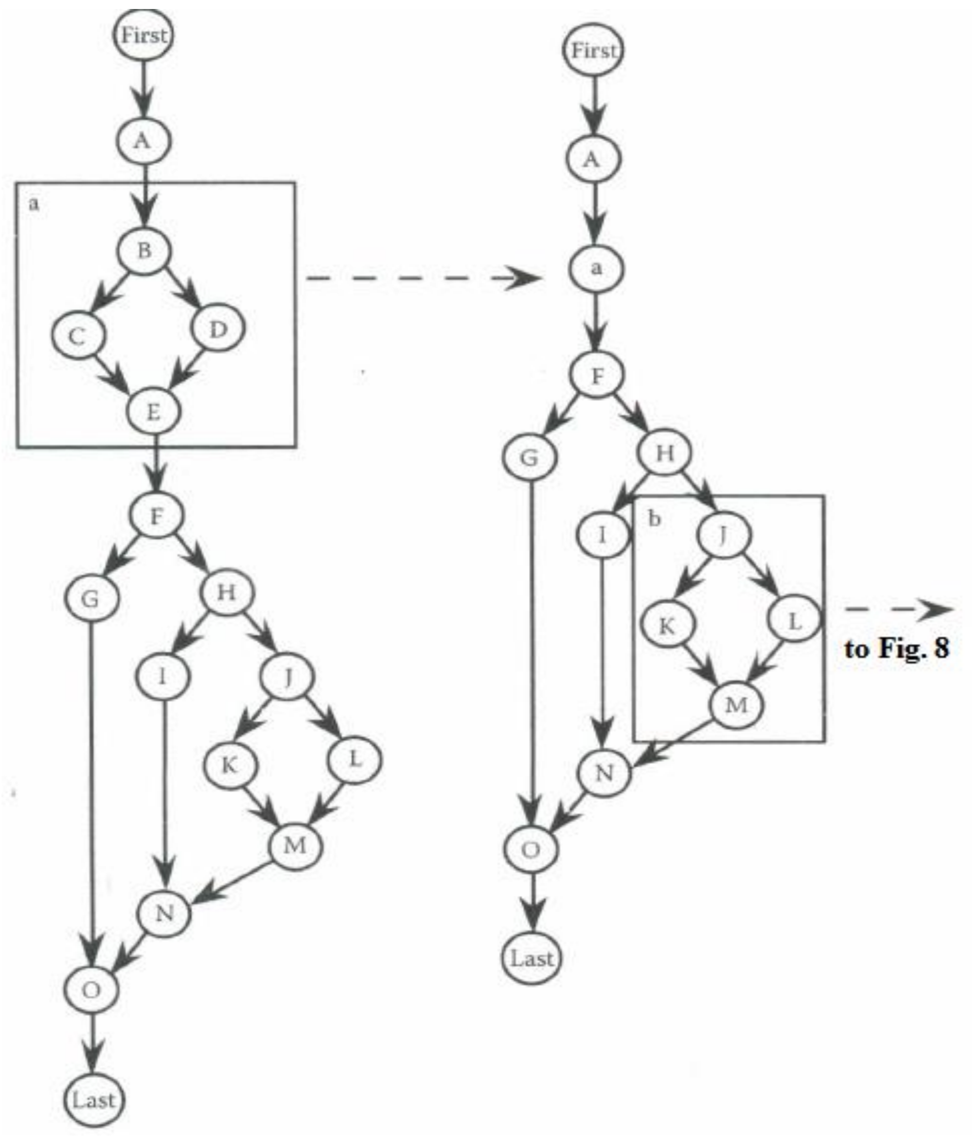


Fig. 7: Condensing with respect to the structured programming constructs

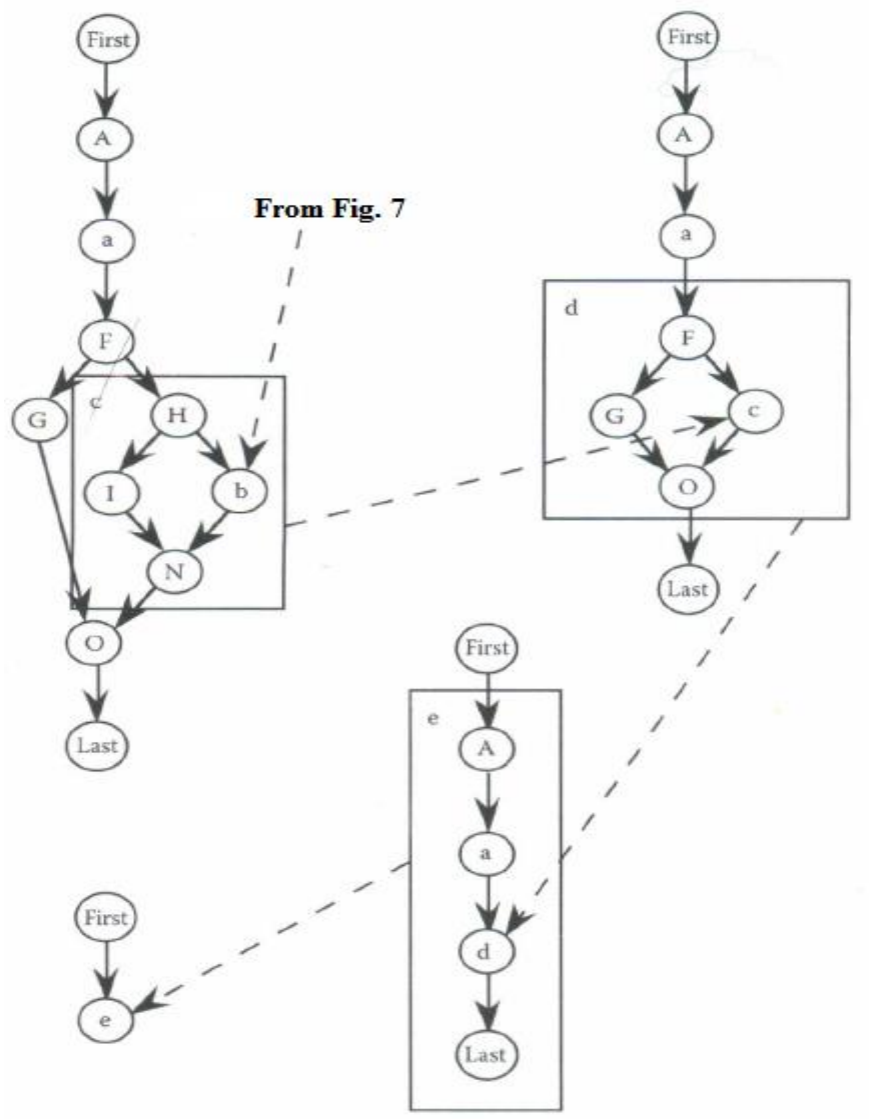


Fig. 8: Condensing with respect to the structured programming constructs

This process is followed in Figure 7 and Figure 8, 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 is condensed into node ***e***, resulting in a condensed graph with cyclomatic complexity V(G) = 1.

The bottom line for testers is the programs with high cyclomatic complexity require more testing. The organizations that use the cyclomatic complexity metric, most will set maximum acceptable complexity V(G) = 10 is a common choice.

What happens if a unit has a higher complexity?

1. Simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1, so it can be simplified easily.
2. If the unit has an essential complexity that exceeds the guidelines, the best choice is to eliminate the unstructures

Dataflow Testing

Dataflow 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).

Two mainline forms of dataflow testing is discussed:

1. Provides a set of basic definitions and a unifying structure of test coverage metrics,
2. Other is based on a concept called a program slice.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values and these values are used to compute values for other variables.

Early dataflow analyses centered on a set of faults that are 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. Because the concordance information is compiler generated, these anomalies can be discovered by static analysis: finding faults in source code without executing it.

3.5 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. The set of all paths in P is PATHS(P)

Definition: *DEF (v, n)*

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

Ex: Input statements, assignment statements, loop control statements, and procedure calls

When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition: *USE (v, n)*

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

Ex: Output statements, assignment statements, conditional statements, loop control statements and procedure calls. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition: (P-use) and (C-use)

A usage node USE(v, n) is a *predicate use* (P-use) iff the statement n is a predicate statement; otherwise, USE(v, n) is a *computation use* (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: (du-path)

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

Definition: (dc-path)

A *definition-clear path with respect to a variable* v (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.

3.5.1 Example (Commission Problem)

- This program computes the commission on the sales of the total numbers of locks, stocks and barrels sold. The While loop is a classical 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

Program:

```
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.   totalLocks = 0
11.   total Stocks = 0
12.   totalBarrels = 0
13.   Input (locks)
14.   While NOT(locks = -1) "Loop' condition uses -1 to indicate end of data
15.       Input (stocks, barrels)
16.       total Locks = 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.   barrelSales = barrel Price * 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
```

Fig. 9 shows the program graph of commission problem and

Fig.10 shows decision-to-decision path (DD-Path) graph of fig. 9. Some DD-Paths are combined to simplify the graph.

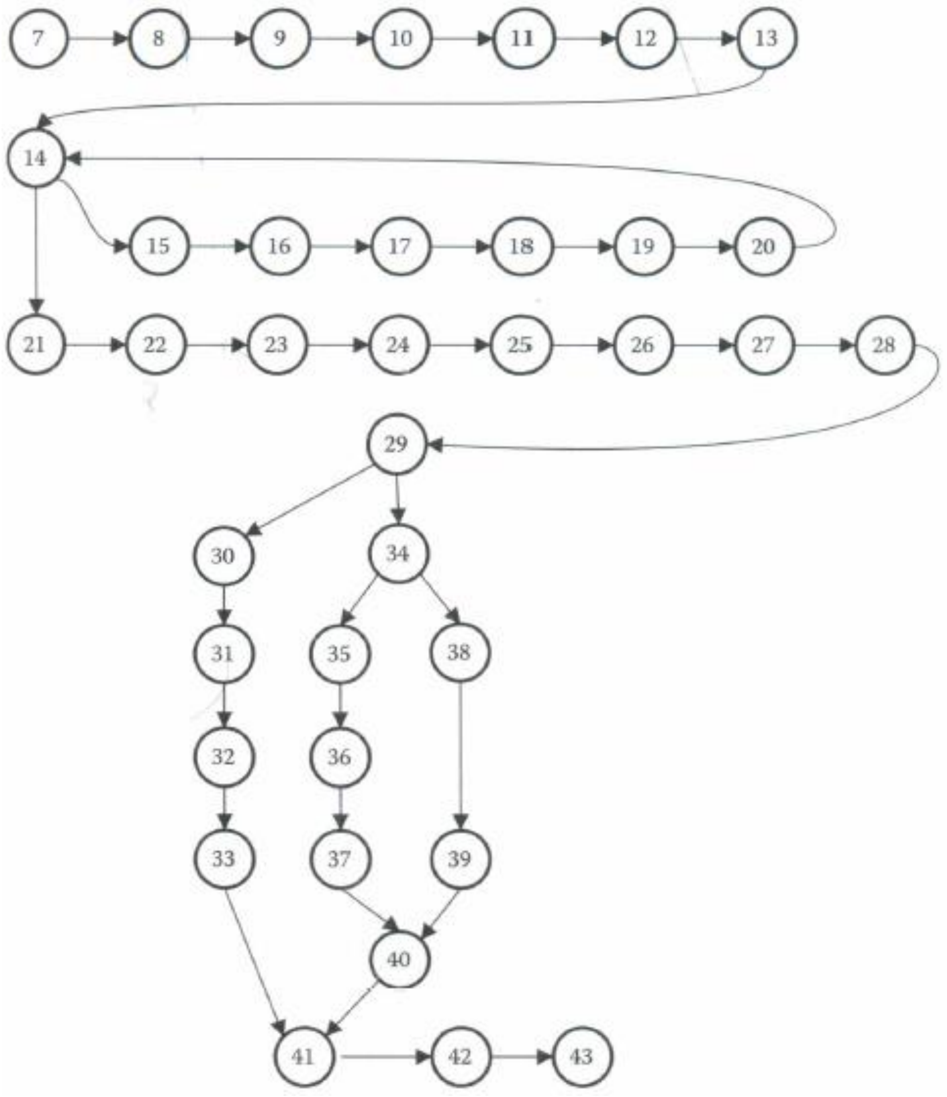


Fig. 9 Program graph of the commission program

Table: DD-Paths in Figure 9

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, 39
J	40
K	41, 42, 43

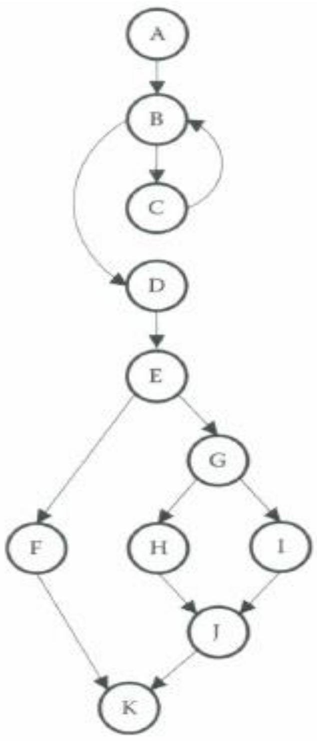
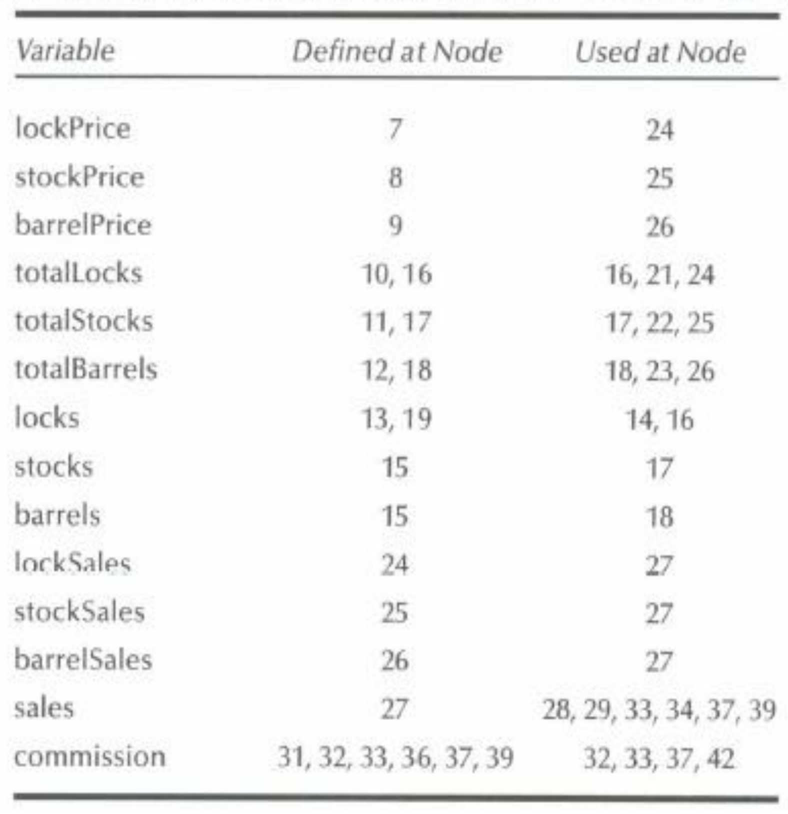


Fig. 10: DD-Path graph of the commission program

Below table lists the **define/usage nodes** for the variables in the commission problem. Use this information in conjunction with the program graph in Figure 9 to identify various definition-use and definition-clear paths.

Table : Define/Use Nodes for Variables in the 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, 39
commission	31, 32, 33, 36, 37, 39	32, 33, 37, 42

The table for selected Define/Use path is shown below. Table presents du-paths in the commission problem; they are named by their beginning and ending nodes. The third column indicates whether the du-paths are ***definition-clear***.

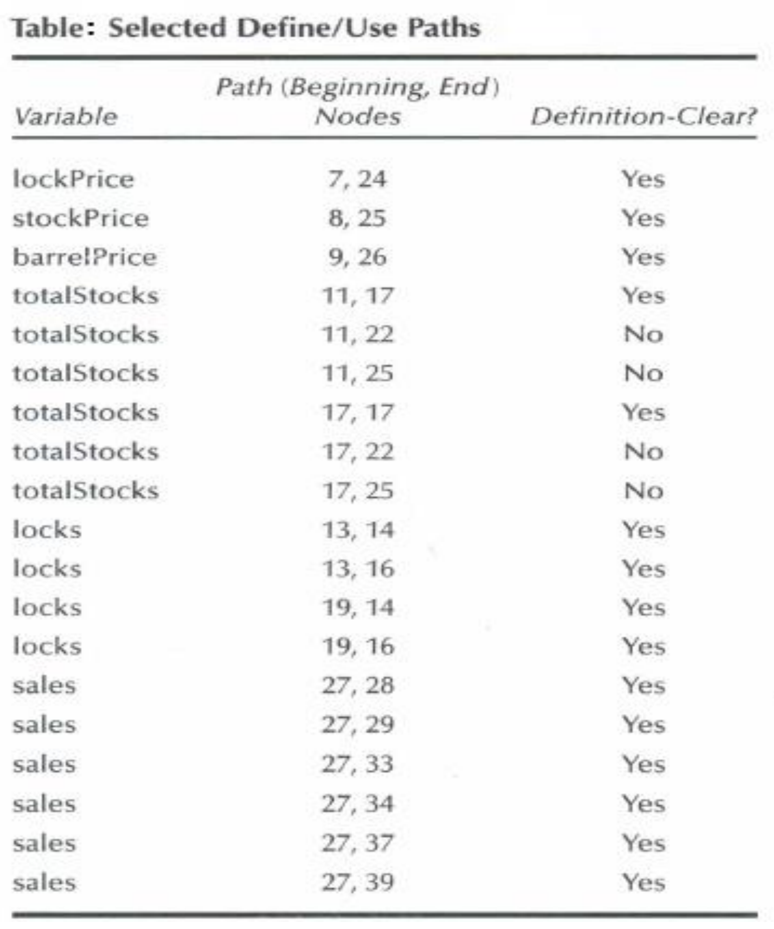


Table: Selected Define/Use Paths

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, 39	Yes

The initial value definition for totalStocks occurs at node 11 and it is first used at node 17.Thus, the path (11, 17), which consists of the node sequence <11, 12, 13, 14, 15, 16, 17>, is definition-clear. The path (11, 22), which consists of the node sequence <11, 12, 13, (14, 15, 16, 17, 18, 19, 20)*, 21, 22> is not definition-clear because values of totalStocks are defined at node 11 and node 17.

1. du-Paths for Stocks

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.

2. du-Paths for Locks

There are 2 defining and 2 usage nodes such as DEF(locks, 13) DEF(locks, 19) USE(locks, 14) and USE(locks, 16). These yield four du-paths:

p1 = <13, 14>

p2 = <13, 14, 15, 16>

p3 = <19, 20, 14>

p4 = <19, 20, 14, 15, 16>

3. du-Paths for totalLocks

There are two defining nodes (DEF (totalLocks, 10) and DEF(totaILocks, 16)) and three usage nodes (USE(totalLocks, 16), USE(totaILocks, 21), USE(totaILocks, 24)),

These yield 6 du-paths:

p5 = <10, 11, 12, 13, 14, 15, 16> is a du-path in which the initial value of totalLocks (0) has a computation use. This path is definition-clear.

p6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21> Path p6 ignores the possible repetition of the While loop (subpath <16, 17, 18, 19, 20, 14, 15>) might be traversed several times. But p6 is a du-path that fails to be definition-clear.

p7 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

p7 = < p6, 22, 23, 24> Du-path p7 is not definition-clear because it includes node 16.

p8 = <16, 17, 18, 19, 20, 14, 21>

p9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24> Both p8 & p9 are definition-clear

4. du-Paths for Sales

Only one defining node is used for sales therefore, all the du-paths with respect to sales must be definition-clear.

p10 = <27, 28>

p11 = <27, 28, 29>

p12 = <27, 28, 29, 30, 31, 32, 33>

p13 = <27, 28, 29, 34>

p14 = <27, 28, 29, 34, 35, 36, 37>

p15 = <27, 28, 29, 34, 38, 39>

5. du-Paths for Commission

- In statements 29 through 41, the calculation of commission is controlled by ranges of the variable sales. Statements 31 to 33 build up the value of commission by using the memory location to hold intermediate values.

- The "built-up" version uses intermediate values, and these will appear as define and usage nodes in the du-path analysis. So disallow du-paths from assignment statements like 31 and 32, just consider du-paths that begin with the three "real" defining nodes: DEF(commission, 33), DEF(commission, 37), and DEF(commission, 38). Only one usage node is used: USE(commission, 42)

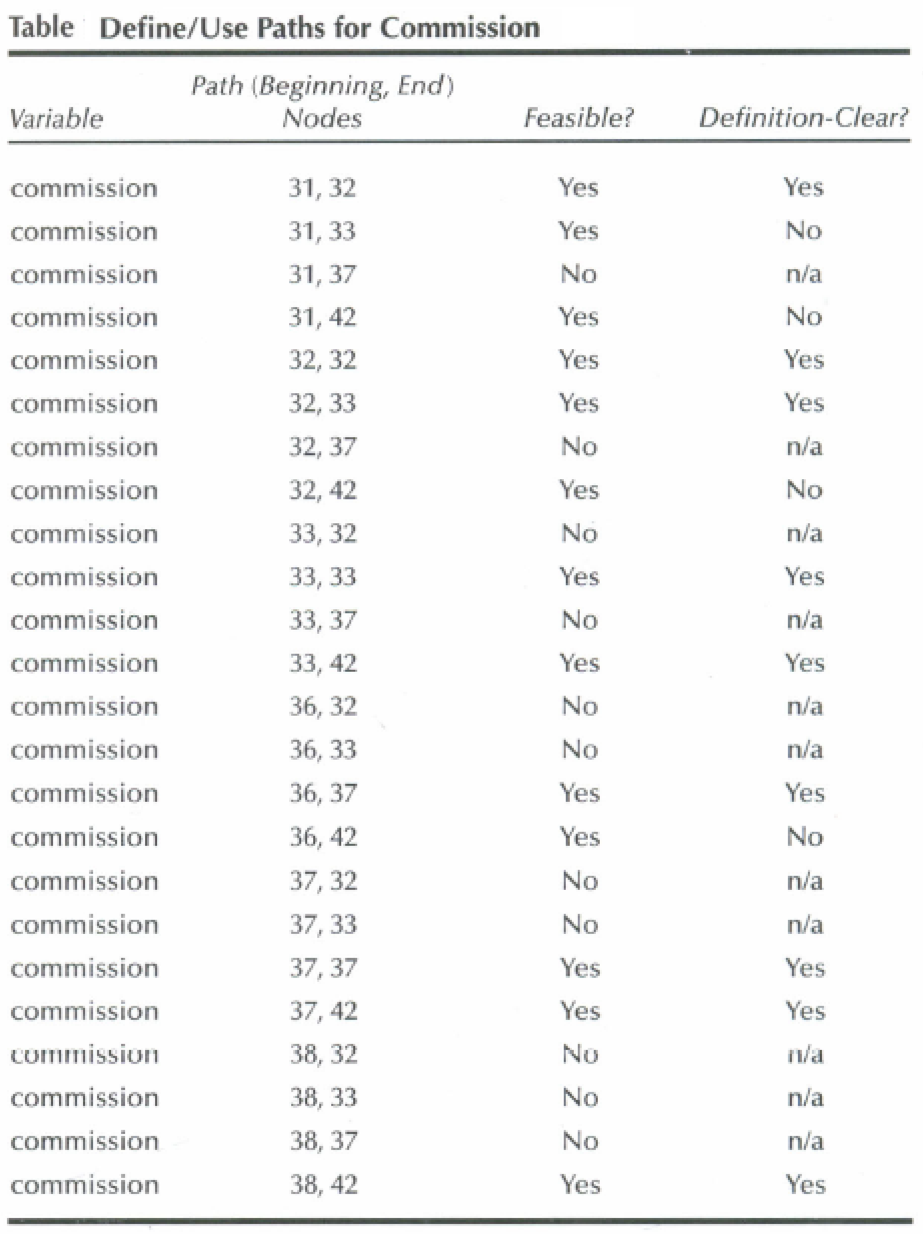


Table Define/Use Paths for Commission

Variable	Path (Beginning, End) Nodes	Feasible?	Definition-Clear?
commission	31, 32	Yes	Yes
commission	31, 33	Yes	No
commission	31, 37	No	n/a
commission	31, 42	Yes	No
commission	32, 32	Yes	Yes
commission	32, 33	Yes	Yes
commission	32, 37	No	n/a
commission	32, 42	Yes	No
commission	33, 32	No	n/a
commission	33, 33	Yes	Yes
commission	33, 37	No	n/a
commission	33, 42	Yes	Yes
commission	36, 32	No	n/a
commission	36, 33	No	n/a
commission	36, 37	Yes	Yes
commission	36, 42	Yes	No
commission	37, 32	No	n/a
commission	37, 33	No	n/a
commission	37, 37	Yes	Yes
commission	37, 42	Yes	Yes
commission	38, 32	No	n/a
commission	38, 33	No	n/a
commission	38, 37	No	n/a
commission	38, 42	Yes	Yes

du-Path Test Coverage Metrics or Rapps-Weyuker dataflow metrics

In the following definitions, T is a set of paths in the program graph G(P) of a program P, with the set V of variables. Assume that the define/use paths are all feasible.

Definition: *(All-Defs)*

The set T satisfies the All-Defs criterion for the program P iff for every variable v   V,

T contains definition-clear paths from every defining node of v to a use of v.

Definition: *(All-Uses)*

The set T satisfies the All-Uses criterion for the program P iff 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: *(All-P-Uses/Some C-Uses)*

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

Definition: *(All-C-Uses/Some P-Uses)*

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

Definition: *(All-du-paths)*

The set T satisfies the All-du-paths criterion for the program P iff for every variable v E 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) and that these paths are either single-loop traversals or cycle-free.

These test coverage metrics have several set theory-based relationships, which are referred to as "subsumption". These relationships are shown in Figure 11

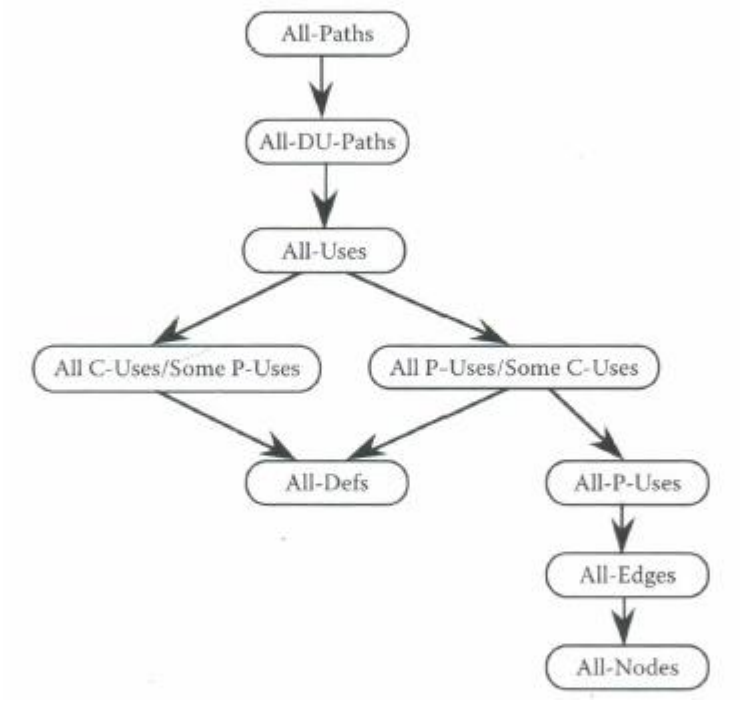


Fig. 11: Rapps-Weyuker hierarchy of dataflow coverage metrics

3.6 Slice-Based Testing

A program slice is a set of program statements that contributes to or affects a value for a variable at some point in the program.

Definitions of a program slice

A program P that has a program graph G(P) and a set of program variables V. The nodes in P(G) to refer to statement fragments.

Definition: ***A slice on the variable at statement n***

Given a program P and a set V of variables in P, ***a slice on the variable set V*** at statement n, written S(V, n), is the set of all statements in ***P*** prior to node ***n*** that contribute to the values of variables in ***V*** at node ***n***

<u>Definition: *A slice on the variable at statement fragment n*</u>

Given a program P and a program graph G(P) in which statements and statement fragments are numbered and a set ***V*** of variables in ***P***, the slice on the variable set ***V*** at statement fragment ***n***, written S(V, n), is the set of node numbers of all statement fragments in ***P*** prior to and including n that contribute to the values of variables in ***V*** at statement fragment ***n***.

"*The idea of slices is to separate a program into components that have some useful (functional) meaning"*.

<u>Explanation of the Definition:</u>

We need to explain two parts of the definition

1. **"<u>prior to</u>"** in the dynamic sense, a slice captures the execution time behavior of a program with respect to the variable(s) in the slice. Eventually, develop a **lattice (a directed, acyclic graph)** of slices, in which nodes are slices and edges correspond to the subset relationship.

2. The **"contribute"**: Means data declaration statements have an effect on the value of a variable. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage

<u>The USE relationship pertains to five forms of usage:</u>

P-use	Used in a predicate (decision)
C-use	Used in computation
O-use	Used for output
L-use	Used for location (pointers, subscripts)

I-use	Iteration (internal counters, loop indices)

Two forms of definition nodes:

I-def	Defined by input
A-def	Defined by assignment

Imp

Assume that the slice **S(V, n)** is a slice on one variable; that is the set V consists of a single variable v.

- If statement fragment ***n*** is a defining node for v, then ***n*** is included in the slice.
- If statement fragment ***n*** is a usage node for v, then ***n*** is not included in the slice.
- P-uses and C-uses of other variables (not the v in the slice set V) are included to the extent that their execution affects the value of the variable v.
- If the value of v is the same whether a statement fragment is included or excluded, exclude the statement fragment.
- O-use, L-use, and l-use nodes are excluded from slices.

Example: The commission problem

1. Slices on the locks: Slices on the locks variable show why it is potentially fault-prone. It has a P-use at node 14 and a C-use at node 16 and has two definitions, the I-defs at nodes 13 and 19.

S1: S(locks, 13) = {13}

S2: S(locks, 14) = {13, 14, 19, 20}

S3: S(locks, 16) = {13, 14, 19, 20}

S4: S(locks, 19) = {19

2. Slices for stocks and barrels:

S5: S(stocks, 15) = {13, 14, 15, 19, 20}

S6: S(stocks, 17) = {13, 14, 15, 19, 20}

S7: S(barrels, 15) = {13, 14, 15, 19, 20}

S8: S(barrels, 18) = {l3, 14, 15, 19, 20}

3. Slices for totalLocks:

S9: S(totalLocks, 10) = {l0}

S10: S(totalLocks, 16) = {l0, 13, 14, 16, 19, 20}

S11: S(totalLocks, 21) = {10,13, 14, 16, 19, 20}

Slices S10and S11are equal because nodes 21and 24 are an O-use and a C-use of totalLocks

4. Slices on totalStocks and totalBarrels:

S12: S(totalStocks, 11) = {11}

S13: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}

S14: S(totalStocks, 22) = {11, 13, 14, 15, 17, 19, 20}

S15: S(totalBarrels, 12) = {12}

S16: S(totalBarrels,18) = {12, 13, 14, 15, 18, 19, 20}

S17: S(totalBarrels, 23) = {12, 13, 14, 15, 18, 19, 20}

5. Assignment statements:

S18: S(lockPrice, 24) = {7}

S19: S(stockPrice, 25) = {8}

S20: S(barrelPrice, 26) = {9}

S21: S(lockSales, 24) = {7, 10, 13, 14, 16, 19,20, 24}

S22: S(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19,20, 25}

S23: S(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19,20, 26}

6. Slices on sales and commission:

Only one defining node exists for sales, the A-def at node 27. The remaining slices on sales show the P-uses, C-uses, and the O-use in definition-clear paths.

S24: S(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S25: S(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S26: S(sales, 29) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S27: S(sales, 33) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S28: S(sales, 34) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S29: S(sales, 37) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

S30: S(sales, 39) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

Six A-def nodes are used for commission. Three computations of commission are controlled by P-uses of sales in the IF, ELSE IF logic. This yields three paths of slices that compute commission.

S31: S(commission, 31) = {3l}

S32: S(commission, 32) = {31, 32}

S33: S(commission, 33) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33}

S34: S(commission, 36) = {36}

S35: S(commission, 37) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 36, 37}

S36: S(commission, 39) = {7,8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,20,24,25,26,27,29, 34, 38, 39}

S37: S(commission, 41) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36,37, 38, 39}

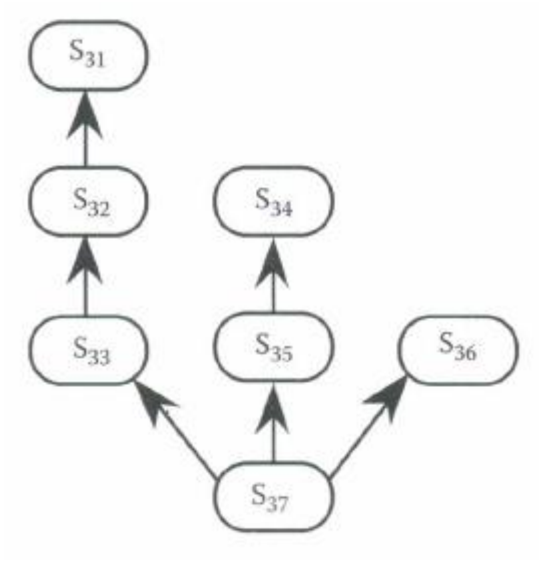

$$\begin{tikzcd}
S_{31} & & \\
S_{32} \arrow[u] & S_{34} & \\
S_{33} \arrow[u] & S_{35} \arrow[u] & S_{36} \\
 & S_{37} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Fig. 11: lattice of slices on commission

$$\begin{tikzcd}
S_{31} & S_{10} & S_{13} & S_{16} & S_{21} & S_{22} & S_{23} \\
S_{32} \arrow[u] & & & S_{24} \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] \arrow[urr] \arrow[urrr] & & S_{34} & \\
& S_{33} \arrow[ul] \arrow[urr] & & S_{36} \arrow[u] & & S_{35} \arrow[ull] \arrow[u] & \\
& & & S_{37} \arrow[ull] \arrow[u] \arrow[urr] & & &
\end{tikzcd}$$

Fig. 12: lattice on sales and commission

Unit -5 Selenium and TestNG

Selenium Training Tutorials
After hundreds of requests from STH readers, today we are finally launching our FREE Selenium Tutorial series. In this Selenium training series we will cover all Selenium testing concepts and its packages in detail with easy to understand practical examples.
These Selenium tutorials are helpful for beginner to advanced level Selenium users. Starting from the very basic Selenium concepts tutorial, we will gradually move on to the advanced topics like Framework creation, Selenium Grid and Cucumber BDD.
Note: We will be increasing our article posting frequency for this series. Please don't miss any tutorial. Keep track of all the tutorials by bookmarking this page as we will keep updating it with links to all new Selenium tutorials.
How to start Learning Selenium?
This is the best time to start learning Selenium testing by your own with the help of this free Selenium Training series. Read tutorials, practice examples at your home, and put your queries in comment section of respective tutorials. We will address all of these queries.
Experienced Selenium professionals – you too can take part in this series by providing answers to reader's queries in comments.
This is our serious effort to help you learn and master one of the most popular software testing tools!
Selenium Introduction:
We are delighted to launch our yet another series of software testing training tutorials. The belief behind introducing this tutorial is to make you an expert in a widely used software test automation solution, Selenium.
In this series we will look at the various facets of Selenium. Selenium is not just a tool; it is a cluster of independent tools. We will look into some of the tools in detail, providing practical examples wherever applicable.
Before you jump in to reading this exciting and useful series, let us take a look at what it has got in store for you.
Why Selenium?
As the current industry trends have shown that there is mass movement towards automation testing. The cluster of repetitive manual testing scenarios has raised a demand to bring in the practice of automating these manual scenarios.
The benefits of implementing automation test are many; let us take a look at them:

- Supports execution of repeated test cases
- Aids in testing a large test matrix
- Enables parallel execution
- Encourages unattended execution
- Improves accuracy thereby reducing human generated errors

- Saves time and money

All this results in to the following:

- High ROI
- Faster GoTo market

Automation testing benefits are many and well understood and largely talked about in the software test industry.
One of the most commonly asked question comes with this is –

- What is the best tool for me to get my tests automated?
- Is there a cost involved?
- Is it easy to adapt?

One of the best answers to all the above questions for automating web based applications is Selenium. Because:

- It's open source
- have a large user base and helping communities
- have multi browser and platform compatibility
- has active repository developments
- supports multiple language implementations

First glance at Selenium
Selenium is one of the most popular automated testing suites. Selenium is designed in a way to support and encourage automation testing of functional aspects of web based applications and a wide range of browsers and platforms. Due to its existence in the open source community, it has become one of the most accepted tools amongst the testing professionals.
Selenium supports a broad range of browsers, technologies and platforms.
Selenium Components
Selenium is not just a single tool or a utility, rather a package of several testing tools and for the same reason it is referred to as a Suite. Each of these tools is designed to cater different testing and test environment requirements.
The suite package constitutes of the following sets of tools:

- Selenium Integrated Development Environment (IDE)
- Selenium Remote Control (RC)
- Selenium WebDriver
- Selenium Grid

Selenium RC and WebDriver, in a combination are popularly known as Selenium 2. Selenium RC alone is also referred as Selenium 1.
Brief Introduction to Selenium tools
Selenium Core

Selenium is a result of continuous efforts by an engineer at ThoughtWorks, named as Jason Huggins. Being responsible for the testing of an internal Time and Expenses application, he realized the need for an automation testing tool so as to get rid of repetitive manual tasks without compromising with the quality and accuracy.
As a result, he built a JavaScript program, named as "JavaScriptTestRunner" in early 2004 that could automatically control the browser's actions which seemed very much similar to that of a user communicating with the browser.
Henceforth, Jason started demoing the tool to the vast audience. Eventually the discussions were laid out to categorize this tool in the open source category as well as its potential to grow as a re-usable testing framework for other web based applications.
The tool was later on acclaimed with the name "Selenium Core".
Selenium IDE (Selenium Integrated Development Environment)
Selenium IDE was developed by Shinya Kasatani. While studying Selenium Core, he realized that this JavaScript code can be extended to create an integrated development environment (IDE) which can be plugged into Mozilla Firefox. This IDE was capable of recording and playing back the user actions on a Firefox instance to which it was plugged-in. Later on Selenium IDE became a part of Selenium Package in the year 2006. The tool turned out a great value and potential to the community.
Selenium IDE is the simplest and easiest of all the tools within the Selenium Package. Its record and playback feature makes it exceptionally easy to learn with minimal acquaintances to any programming language. With several advantages, a few disadvantages accompanied Selenium IDE, thus making it inappropriate to be used in cases of more advanced test scripts.
Advantages and disadvantages of Selenium IDE:
The disadvantages of IDE are in reality not disadvantages of selenium, rather just limitations to what IDE could achieve. These limitations can be overcome by using Selenium RC or WebDriver.
Selenium RC (Selenium Remote Control)
Selenium RC is a tool which is written in java that allows a user to construct test scripts for a web based application in which ever programming language he/she chooses. Selenium RC came as result to overcome various disadvantages incurred by Selenium IDE or Core.
Loopholes and restrictions which were imposed while using Selenium Core made it difficult for the user to leverage the benefits of the tool to its totality. Thus it made the testing process a cumbersome and a far reaching task.
One of the crucial restrictions was same origin policy.
Problem of same origin policy: The problem of same origin policy disallows to access the DOM of a document from an origin that is different from the origin we are trying to access the document.
Origin is a sequential combination of scheme, host and port of the URL. For example, for a URL http://www.seleniumhq.org/projects/, the origin is a combination of http, seleniumhq.org, 80 correspondingly.
Thus the Selenium Core (JavaScript Program) cannot access the elements from an origin that is different from where it was launched.
For Example, if I have launched the JavaScript Program from "http://www.seleniumhq.org/", then I would be able to access the pages within the same domain such as "http://www.seleniumhq.org/projects/" or "http://www.seleniumhq.org/download/". The other domains like google.com, yahoo.com would no more be accessible.
Thus, to test the application using Selenium Core, one has to install the entire application on the Selenium Core as well as web server to overcome the problem of same origin policy.

So, In order to govern the same origin policy without the need of making a separate copy of Application under test on the Selenium Core, Selenium Remote Control was introduced. While Jason Huggins was demoing Selenium, another fellow colleague at ThoughtWorks named Paul Hammant suggested a work around of same origin policy and a tool that can be wired up with a programming language of our choice. Thus Selenium RC came into existence.
Unlike selenium IDE, selenium RC supports a wide range of browsers and platforms.
Workflow Description

- User creates test scripts in a desired programming language.
- For every programming language, there is a designated client library.
- Client library deports the test commands to the selenium server.
- Selenium server deciphers and converts the test commands into JavaScript commands and sends them to the browser.
- Browser executes the commands using selenium core and sends results back to the selenium server
- Selenium server delivers the test results to the client library.

There are a few pre-requisites to be in place before creating Selenium RC scripts:

- A Programming Language – Java, C#, Python etc.
- An Integrated Development Environment –Eclipse, Netbeans etc.
- A Testing Framework (optional) – JUnit, TestNG etc.
- And Selenium RC setup off course

Advantages and disadvantages of selenium RC:
Coming on to the advantages and disadvantages of selenium RC, refer the following figure.
Selenium Grid
With selenium RC, life of a tester has always been positive and favorable until the emerging trends raised a demand to execute same or different test scripts on multiple platforms and browsers concurrently so as to achieve distributed test execution, testing under different environments and saving execution time remarkably. Thus, catering these requirements selenium grid was brought into the picture.
Selenium Grid was introduced by Pat Lightbody in order to address the need for executing the test suites on multiple platforms simultaneously.
Selenium WebDriver
Selenium WebDriver was created by yet another engineer at ThoughtWorks named as Simon Stewart in the year 2006. WebDriver is also a web-based testing tool with a subtle difference with Selenium RC. Since, the tool was built on the fundamental where an isolated client was created for each of the web browser; no JavaScript Heavy lifting was required. This led to a compatibility analysis between Selenium RC and WebDriver. As a result a more powerful automated testing tool was developed called Selenium 2.
WebDriver is clean and a purely object oriented framework. It utilizes the browser's native compatibility to automation without using any peripheral entity. With the increasing demand it has gained a large popularity and user base.
Advantages and disadvantages of Selenium WebDriver:
Refer the following figure for the advantages and disadvantages of WebDriver.

Selenium 3

Selenium 3 is an advance version of Selenium 2. It is a tool focused for automation of mobile and web applications. Stating that it supports mobile testing, we mean to say that the WebDriver API has been extended to address the needs of mobile application testing. The tool is expected to be launched soon in the market.

Environment and Technology Stack

With the advent and addition of each new tool in the selenium suite, environments and technologies became more compatible. Here is an exhaustive list of environments and technologies supported by selenium tool set.

Supported Browsers

Supported Programming Languages

Supported Operating Systems

Supported Testing Frameworks

Conclusion

In this tutorial, we tried to make you acquainted with the Selenium suite describing its various components, their usages and their advantages over each other.

Here are the cruxes of this article.

- Selenium is a suite of several automated testing tools, each of them catering to different testing needs.
- All these tools fall under the same umbrella of open source category and supports only web based testing.
- Selenium suite is comprised of 4 basic components; Selenium IDE, Selenium RC, WebDriver, Selenium Grid.
- User is expected to choose wisely the right Selenium tool for his/her needs.
- Selenium IDE is distributed as a Firefox plug-in. It is easier to install and use. User is not required to possess prior programming knowledge. Selenium IDE is an ideal tool for a naive user.
- Selenium RC is a server that allows user to create test scripts in a desired programming language. It also allows executing test scripts within the large spectrum of browsers.
- Selenium Grid brings out an additional feature to Selenium RC by distributing its test script on different platforms and browsers at the same time for execution, thus implementing the master slave architecture.
- WebDriver is a different tool altogether that has various advantages over Selenium RC. The fusion of Selenium RC and WebDriver is also known as Selenium 2. WebDriver directly communicates with the web browser and uses its native compatibility to automate.
- Selenium 3 is the most anticipated inclusion in the Selenium suite which is yet to be launched in the market. Selenium 3 strongly encourages mobile testing.

Getting Started with Selenium IDE (Installation and its Features) –Selenium Tutorial #2

Before moving ahead, let's take a moment to look at the agenda of this tutorial. In this tutorial, we will learn all about Selenium IDE, starting from its installation to the details about each of its features. At the end of this tutorial, the reader is expected to be able to install Selenium IDE tool and play around with its features.

=> This is a 2nd tutorial in our free online Selenium training series. If you have not read the first Selenium tutorial in this series please get started from here: Free online Selenium Tutorial #1

Note: This is quite a extensive tutorial with lots of images so allow it to load completely. Also click on image or open in new window to enlarge images.

Introduction to Selenium IDE

Selenium integrated development environment, acronym as Selenium IDE is an automated testing tool that is released as a Firefox plug-in. It is one of the simplest and easiest tools to install, learn and to go ahead with the creation of test scripts. The tool is laid on a record and playback fundamental and also allows editing of the recorded scripts.

The most impressive aspect of using selenium IDE is that the user is not required to possess any prior programming knowledge. The minimum that the user needs is the little acquaintances with HTML, DOMS and JavaScript to create numerous test scripts using this tool.

Being a Firefox plug-in, Selenium IDE supports only Firefox, thus the created test scripts could be executed only on Firefox. A few more loopholes make this tool inappropriate to be used for complex test scripts. Thus, other tools like Selenium RC, WebDriver comes into the picture.

So, before gripping on to the details of Selenium IDE, let's have a look at its installation first.

Selenium IDE Download and Installation

For the ease of understanding, I have bifurcated the entire IDE installation process in the following chunks/steps.

Before taking off, there is one thing that needs to be in place prior to the installation; Mozilla Firefox. You can download it from here => Mozilla Firefox download. Step #1: Selenium IDE download: Open the browser (Firefox) and enter the URL – http://seleniumhq.org/ . This would open the official Selenium head quarter website. Navigate to the "Download" page; this page embodies all the latest releases of all the selenium components. Refer the following figure.

Step #2: Move under the selenium IDE head and click on the link present. This link represents the latest version of the tool in the repository. Refer the following figure.

Step #3: As soon as we click on the above link, a security alert box would appear so as to safeguard our system against potential risks. As we are downloading the plug-in from the authentic website, thus click on the "Allow" button.

Step #4: Now Firefox downloads the plug-in in the backdrop. As soon as the process completes, software installation window appears. Now click on the "Install Now" button.

Step #5: After the installation is completed, a pop up window appears asking to re-start the Firefox. Click on the "Restart Now" button to reflect the Selenium IDE installation.

Step #6: Once the Firefox is booted and started again, we can see selenium IDE indexed under menu bar -> Web Developer -> Selenium IDE.

Step #7: As soon as we open Selenium IDE, the Selenium IDE window appears.

Features of Selenium IDE

Let's have a look at each of the feature in detail.

#1. Menu Bar

Menu bar is positioned at the upper most of the Selenium IDE window. The menu bar is typically comprised of five modules.

- File Menu
- Edit Menu
- Actions Menu
- Options Menu
- Help Menu

A) File Menu

File Menu is very much analogous to the file menu belonging to any other application. It allows user to:

- Create new test case, open existing test case, save the current test case.
- Export Test Case As and Export Test Suite As in any of the associated programming language compatible with Selenium RC and WebDriver. It also gives the liberty to the user to prefer amid the available unit testing frameworks like jUnit, TestNG etc. Thus an IDE test case can be exported for a chosen union of programming language, unit testing framework and tool from the selenium package.
- Export Test Case As option exports and converts only the currently opened Selenium IDE test case.
- Export Test Suite As option exports and converts all the test cases associated with the currently opened IDE test suite.
- Close the test case.

The Selenium IDE test cases can be saved into following format:

- HTML format

The Selenium IDE test cases can be exported into following formats/programming languages.

- java (IDE exported in Java)
- rb (IDE exported in Ruby)
- py (IDE exported in Python)
- cs (IDE exported in C#)

Notice that with the forthcoming newer versions of Selenium IDE, the support to formats may expand.

B) Edit Menu Edit menu provides options like Undo, Redo, Cut, Copy, Paste, Delete and Select All which are routinely present in any other edit menu. Amongst them, noteworthy are:

- Insert New Command – Allows user to insert the new command/test step anywhere within the current test case.
- Insert New Comment – Allows user to insert the new comment anywhere within the current test case to describe the subsequent test steps.

Insert New Command

The new command would be inserted above the selected command/test step.

Now the user can insert the actual command action, target and value.

\Insert New Comment

In the same way we can insert comments.
The purple color indicates that the text is representing a comment.
C) Actions Menu
Actions Menu equips the user with the options like:
 Record – Record options fine tunes the Selenium IDE into the recording mode. Thus, any action made by the user on the Firefox browser would be recorded in IDE.

 Play entire test suite – The option plays all the Selenium IDE test cases associated with the current test suite.

 Play current test case – The option plays the current Selenium IDE test case that has been recorded/created by the user.
 Pause / Resume – User can Pause/Resume the test case at any point of time while execution.

 Toggle Breakpoint – User can set one or multiple breakpoint(s) to forcefully break the execution at any particular test step during execution.

 Set / Clear Start Point – User can also set start point at any particular test step for execution. This would enable user to execute the test case from the given start point for the subsequent runs.

 To deal with the page/element loads, the user can set the execution speed from fastest to lowest with respect to the responsiveness of the application under test.

D) Options Menu
Options menu privileges the user to set and practice various settings provided by the Selenium IDE. Options menu is recommended as one of the most important and advantageous menu of the tool.
Options Menu is primarily comprised of the following four components which can be sub-divided into the following:
Options
Selenium IDE Options dialog box
To launch Selenium IDE Options dialog box, follow the steps:
1. Click on Options Menu
2. Click on the Options

A Selenium IDE Options dialog box appears. Refer the following figure.
Selenium IDE Options dialog box aids the user to play with the general settings, available formats, available plug-ins and available locators types and their builders.
Let’s have a look at the few important ones.
General Settings
 Default Timeout Value – Default Timeout Value represents the time (in milliseconds) that selenium would wait for a test step to execute before generating an error. The standard timeout value is 30000 milliseconds i.e. 30 seconds. The user can leverage this feature by changing the default time in cases when the web element takes more/less than the specified time to load.

 Extensions – Selenium IDE supports a wide range of extensions to enhance the capabilities of the core tool thereby multiplying its potential. These user extensions are simply the JavaScript files. They can set by mentioning their absolute path in the text boxes representing extensions in the Options dialog box.

 Remember base URL – Checking this option enables the Selenium IDE to remember the URL every time we launch it. Thus it is advisable to mark it checked. Un-checking this option will leave the base URL field as blank and it will be re-filled only when we launch another URL on the browser.

 Record assertTitle automatically – Checking this field inserts the assertTitle command automatically along with the target value for every visited web page.

 Enable experimental features – Checking this field for the first time imports the various available formats into the Selenium IDE.

Formats

Formats tab displays all the available formats with selenium IDE. User is levied with the choice to enable and disable any of the formats. Refer the following figure.

Selenium IDE Plugins

Plug-ins tab displays the supported Firefox plug-ins installed on our instance of Selenium IDE. There are a number of plug-ins available to cater different needs, thus we can install these add-ons like we do other plug-ins. One of the recently introduced plug-in is "File Logging". In the end of this tutorial, we will witness how to install and use this plug-in.

With the standard distribution, Selenium IDE comes with a cluster of following plug-ins:

 Selenium IDE: Ruby Formatters

 Selenium IDE: Python Formatters

 Selenium IDE: Java Formatters

 Selenium IDE: C# Formatters

These formatters are responsible to convert the HTML test cases into the desired programming formats.

Locator Builders
Locator builders allow us to prioritize the order of locator types that are generated while recording the user actions. Locators are the set of standards by which we uniquely identify a web element on a web page.
Formats
Formats option allows user to convert the Selenium IDE test case (selenese commands) into desired format. E) Help Menu
As Selenium has a wide community and user base, thus various documentations, release notes, guides etc. are handily available. Thus, the help menu lists down official documentation and release notes to help the user.
#2. Base URL Bar
Base URL bar is principally same as that of an address bar. It remembers the previously visited websites so that the navigation becomes easy later on. Now, whenever the user uses "open" command of Selenium IDE without a target value, the base URL would be launched on to the browser.
Accessing relative paths
To access relative paths, user simply needs to enter a target value like "/download" along with the "open" command. Thus, the base URL appended with "/downloads" (http://docs.seleniumhq.org/resources) would be launched on to the browser. The same is evident in the above depiction.
#3. Toolbar
Toolbar provides us varied options pertinent to the recording and execution of the test case.
 Playback Speed – This option allows user to control the test case execution speed from fast to slow.

 Play test suite – This option allows user to execute all the test cases belonging to the current test suite sequentially.

 Play test case – This option allows user to execute the currently selected test case.

 Pause – This option allows user to pause the current execution.

 Step – This option allows user to step into the test step.
 Rollup– This option allows user to combine multiple test steps to act like a single command.

 Record – This option allows user to start/stop the recording of user actions. The hollow red ball indicates the start of the recording session whereas the solid red ball indicates the end of the recording session. By default, the Selenium IDE opens in the recording mode.

#4. Editor
Editor is a section where IDE records a test case. Each and every user action is recorded in the editor in the same order in which they are performed.
The editor in IDE has two views, namely:
1) Table View
It is the default view provided by Selenium IDE. The test case is represented in the tabular format. Each user action in the table view is a consolidation of "Command", "Target" and "Value" where command, target and value refers to user action, web element with the unique identification and test data correspondingly. Besides recording it also allows user to insert, create and edit new Selenese commands with the help of the editor form present in the bottom.

2) Source View The test case is represented in the HTML format. Each test step is considered be a row <tr> which is a combination of command, target and value in the separate columns <td>. Like any HTML document, more rows and columns can be added to correspond to each Selenese command.
Editor Form lets the user to type any command and the suggestions for the related command would be populated automatically. Select button lets the user to select any web element and its locator would be fetched automatically into the target field. Find button lets the user find the web element on the web page against a defined target. Value is the test input data entered into the targets with which we want to test the scenario.
#5. Test case pane At the instance we open Selenium IDE interface, we see a left container titled “Test case” containing an untitled test case. Thus, this left container is entitled as Test case pane.
Test case pane contains all the test cases that are recorded by IDE. The tool has a capability of opening more than one test case at the same time under test case pane and the user can easily shuffle between the test cases. The test steps of these test cases are organized in the editor section.
Selenium IDE has a color coding ingredient for reporting purpose. After the execution, the test case in marked either in “red” or “green” color.

- Red color symbolizes the unsuccessful run i.e. failure of the test case.
- Green color symbolizes the successful run of the test case
- It also layouts the summary of the total number of test cases executed with the number of failed test cases.
- If we execute a test suite, all the associated test cases would be listed in the test case pane. Upon execution, the above color codes would be rendered accordingly.

#6. Log Pane Log pane gives the insight about current execution in the form of messages along with the log level in the real time. Thus, log messages enable a user to debug the issues in case of test case execution failures.
The printing methods / log levels used for generating logs are:

- Error – Error message gives information about the test step failure. It may be generated in the cases when element is not found, page is not loaded, verification/assertion fails etc.
- Warn – Warning message gives information about unexpected conditions.
- Info – Info message gives information about current test step execution.
- Debug – Debug messages gives information about the technicalities in the backdrop about the current test step.

Logs can be filtered with the help of a drop down located at the top-right corner of the footer beside the clear button. Clear button erases all the log messages generated in the current or previous run.
Generating Logs in an external medium
Recently introduced “File Logging” plug-in enables the user to save log messages into an external file. File Logging can be plugged in to IDE like any other plug-in. Upon installation, it can be found as a tab named “File Logging” in the footer beside the Clear button.
Reference Pane
Reference Pane gives the brief description about the currently selected Selenese command along with its argument details.
UI-Element Pane

UI – Element Pane allows Selenium user to use JavaScript Object Notation acronym as JSON to access the page elements. More on this can be found in UI-Element Documentation under Help Menu.

Rollup Pane

Rollup Pane allows the user to roll up or combine multiple test steps to constitute a single command termed as “rollup”. The rollup in turn can be called multiple times across the test case.