FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Software Testing

A Craftsman's Approach

Fourth Edition

Contents

Chapter 1

A Perspective on Testing

Why do we test? The two main reasons are to make a judgment about quality or acceptability and to discover problems. We test because we know that we are fallible—this is especially true in the domain of software and software-controlled systems. The goal of this chapter is to create a framework within which we can examine software testing.

1.1 Basic Definitions

Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers. The International Software Testing Qualification Board (ISTQB) has an extensive glossary of testing terms (see the website http://www.istqb.org/downloads/glossary.html). The terminology here (and throughout this book) is compatible with the ISTQB definitions, and they, in turn, are compatible with the standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society (IEEE, 1983). To get started, here is a useful progression of terms.

Error—People make errors. A good synonym is *mistake*. When people make mistakes while coding, we call these mistakes *bugs*. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

Fault—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, Unified Modeling Language diagrams, hierarchy charts, and source code. *Defect* (see the ISTQB Glossary) is a good synonym for fault, as is *bug*. Faults can be elusive. An error of omission results in a fault in which something is missing that should be present in the representation. This suggests a useful refinement; we might speak of faults of commission and faults of omission. 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 the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition

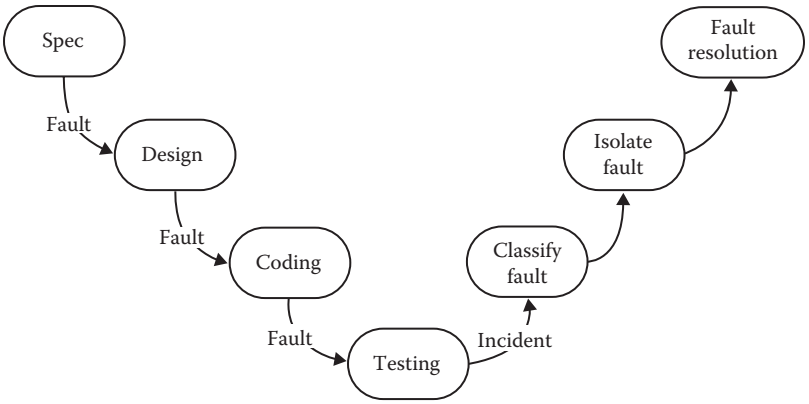

Figure 1.1 A testing life cycle.

relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or perhaps do not execute for a long time? Reviews (see Chapter 22) prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.

Incident—When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

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

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

Figure 1.1 portrays a life cycle model for testing. Notice that, in the development phases, three opportunities arise for errors to be made, resulting in faults that may propagate through the remainder of the development process. The fault resolution step is another opportunity for errors (and new faults). When a fix causes formerly correct software to misbehave, the fix is deficient. We will revisit this when we discuss regression testing.

From this sequence of terms, we see that test cases occupy a central position in testing. The process of testing can be subdivided into separate steps: test planning, test case development, running test cases, and evaluating test results. The focus of this book is how to identify useful sets of test cases.

1.2 Test Cases

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

The output portion of a test case is frequently overlooked, which is unfortunate because this is often the hard part. Suppose, for example, you were testing software that determines an optimal route for an aircraft, given certain Federal Aviation Administration air corridor constraints and the weather data for a flight day. How would you know what the optimal route really is? Various responses can address this problem. The academic response is to postulate the existence of an oracle who "knows all the answers." One industrial response to this problem is known as reference testing, where the system is tested in the presence of expert users. These experts make judgments as to whether outputs of an executed set of test case inputs are acceptable.

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

1.3 Insights from a Venn Diagram

Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code-based view common to software (and system) developers. A quick distinction is that the code-based view focuses on what it *is* and the behavioral view considers what it *does*. One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers; the emphasis is therefore on code-based, instead of behavioral, information. In this section, we develop a simple Venn diagram that clarifies several nagging questions about testing.

Consider a universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. Figure 1.2 shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S and all those behaviors actually programmed are in P. With this diagram, we can see more clearly the problems that confront a tester. What if certain specified behaviors have not been programmed? In our earlier terminology, these are faults of omission. Similarly, what if certain programmed (implemented) behaviors have not been specified? These correspond to faults of commission and to errors that occurred after the specification was complete. The intersection of S and P (the football-shaped region) is the "correct" portion, that is, behaviors that are both specified and implemented. A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented. (As an aside, note that "correctness" only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

The new circle in Figure 1.3 is for test cases. Notice the slight discrepancy with our universe of discourse and the set of program behaviors. Because a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among sets S, P, and T. There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).

Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7).

Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors,

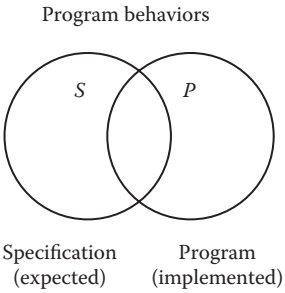

Figure 1.2 Specified and implemented program behaviors.

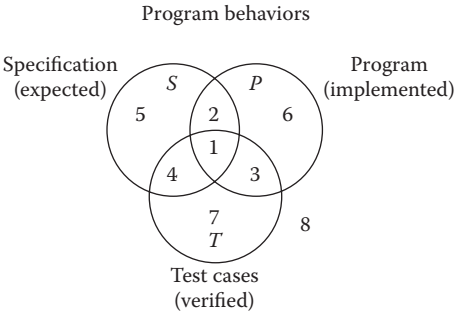

Figure 1.3 Specified, implemented, and tested behaviors.

some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur. (In my experience, good testers often postulate test cases of this latter type. This is a fine reason to have good testers participate in specification and design reviews.)

We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) as large as possible? Another approach is to ask how the test cases in set T are identified. The short answer is that test cases are identified by a testing method. This framework gives us a way to compare the effectiveness of diverse testing methods, as we shall see in Chapter 10.

1.4 Identifying Test Cases

Two fundamental approaches are used to identify test cases; traditionally, these have been called functional and structural testing. Specification-based and code-based are more descriptive names, and they will be used here. Both approaches have several distinct test case identification methods; they are generally just called testing methods. They are methodical in the sense that two testers following the same "method" will devise very similar (equivalent?) test cases.

1.4.1 Specification-Based Testing

The reason that specification-based testing was originally called "functional testing" is that any program can be considered to be a function that maps values from its input domain to values in its output range. (Function, domain, and range are defined in Chapter 3.) This notion is commonly used in engineering, when systems are considered to be black boxes. This led to another synonymous term—black box testing, in which the content (implementation) of the black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs (see Figure 1.4). In *Zen and the Art of Motorcycle Maintenance*, Robert Pirsig refers to this as "romantic" comprehension (Pirsig, 1973). Many times, we operate very effectively with black box knowledge; in fact, this is central to object orientation. As an example, most people successfully operate automobiles with only black box knowledge.

With the specification-based approach to test case identification, the only information used is the specification of the software. Therefore, the 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; and (2) test case development can occur in parallel with the implementation, thereby reducing the overall project development interval. On the negative side, specification-based test cases frequently suffer from two problems: significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software.

Figure 1.5 shows the results of test cases identified by two specification-based methods. Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because specification-based methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified. In Chapter 8, we will see direct comparisons of test cases generated by various specification-based methods for the examples defined in Chapter 2.

In Chapters 5 through 7, we will examine the mainline approaches to specification-based testing, including boundary value analysis, robustness testing, worst-case analysis, special value testing, input (domain) equivalence classes, output (range) equivalence classes, and decision table-based testing. The common thread running through these techniques is that all are based on

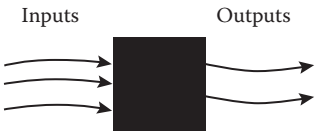

Figure 1.4 Engineer's black box.

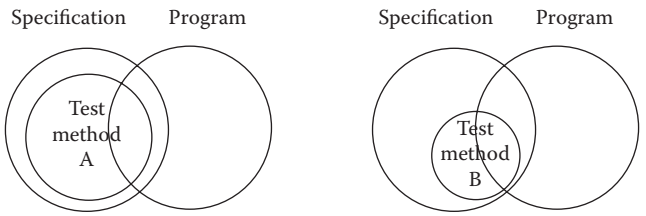

Figure 1.5 Comparing specification-based test case identification methods.

definitional information of the item tested. Some of the mathematical background presented in Chapter 3 applies primarily to specification-based approaches.

1.4.2 Code-Based Testing

Code-based testing is the other fundamental approach to test case identification. To contrast it with black box 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 on the basis of how the function is actually implemented.

Code-based testing has been the subject of some fairly strong theories. To really understand code-based testing, familiarity with the concepts of linear graph theory (Chapter 4) is essential. With these concepts, the tester can rigorously describe exactly what is tested. Because of its strong theoretical basis, code-based 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 1.6 shows the results of test cases identified by two code-based 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 code-based 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 code-based 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 code-based test cases is relatively small with respect to the full set of programmed behaviors. In Chapter 10, we will see direct comparisons of test cases generated by various code-based methods.

1.4.3 Specification-Based versus Code-Based Debate

Given the 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.

The Venn diagrams presented earlier yield a strong resolution to this debate. Recall that the goal of both approaches is to identify test cases (Figure 1.7). Specification-based testing uses only the specification to identify test cases, while code-based testing uses the program source code (implementation) as the basis of test case identification. Later chapters will establish that

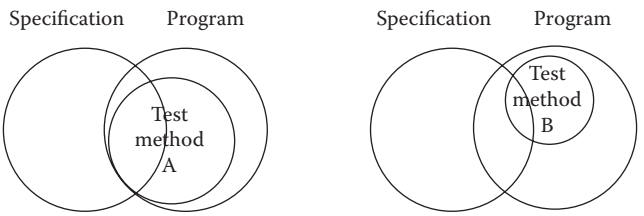

Figure 1.6 Comparing code-based test case identification methods.

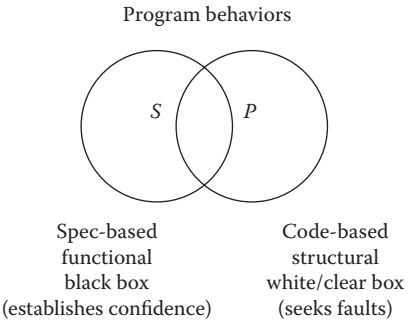

Figure 1.7 Sources of test cases.

neither approach by itself is sufficient. Consider program behaviors: if all specified behaviors have not been implemented, code-based 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 specification-based 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 specification-based testing and the measurement of code-based testing. Earlier, we asserted that specification-based testing often suffers from twin problems of redundancies and gaps. When specification-based test cases are executed in combination with code-based test coverage metrics, both of these problems can be recognized and resolved.

The Venn diagram view of testing provides one final insight. What is the relationship between set T of test cases and sets S and P of specified and implemented behaviors? Clearly, the test cases in set 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.

1.5 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, whereas 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 (fault) occurrence: one time only, intermittent, recurring, or repeatable.

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 Tables 1.1 through 1.5; most of these are from the IEEE standard but I have added some of my favorites.

Since the primary purpose of a software review is to find faults, review checklists (see Chapter 22) are another good source of fault classifications. Karl Wiegers has an excellent set of checklists on his website: http://www.processimpact.com/pr_goodies.shtml.

Table 1.1 Input/Output Faults

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

Table 1.2 Logic Faults

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

Table 1.3 Computation Faults

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

Table 1.4 Interface Faults

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

Table 1.5 Data Faults

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

1.6 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. A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing—system, integration, and unit testing.

A practical relationship exists between levels of testing versus specification-based and code-based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level. This is generally true; however, 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 code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Chapters 11 through 17 to support code-based testing at the integration and system levels for both traditional and object-oriented software.

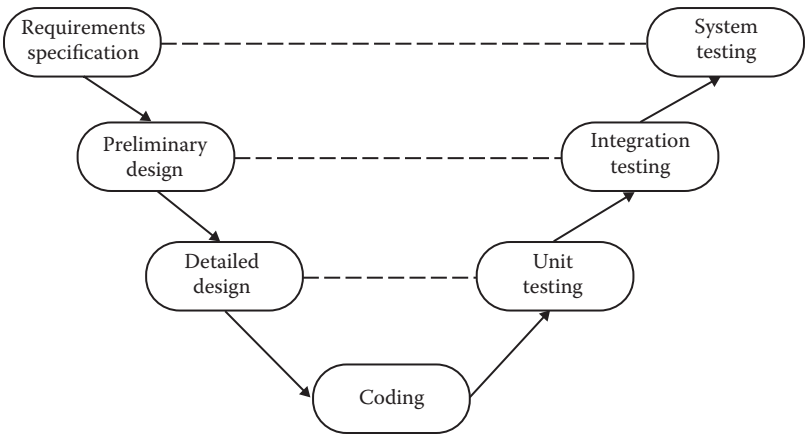

Figure 1.8 Levels of abstraction and testing in waterfall model.

EXERCISES

1. Make a Venn diagram that reflects a part of the following statement: "… we have left undone that which we ought to have done, and we have done that which we ought not to have done …"
2. Make a Venn diagram that reflects the essence of Reinhold Niebuhr's "Serenity Prayer":
   God, grant me the serenity to accept the things I cannot change,
   Courage to change the things I can,
   And wisdom to know the difference.
3. Describe each of the eight regions in Figure 1.3. Can you recall examples of these in software you have written?
4. One of the tales of software lore describes a disgruntled employee who writes a payroll program that contains logic that checks for the employee's identification number before producing paychecks. If the employee is ever terminated, the program creates havoc. Discuss this situation in terms of the error, fault, and failure pattern, and decide which form of testing would be appropriate.

References

IEEE Computer Society, *IEEE Standard Glossary of Software Engineering Terminology*, 1983, ANSI/IEEE Std 729-1983.

IEEE Computer Society, *IEEE Standard Classification for Software Anomalies*, 1993, IEEE Std 1044-1993.

Pirsig, R. M., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, New York, 1973.

Chapter 2

Examples

Three examples will be used throughout in Chapters 5 through 9 to illustrate the various unit testing methods: the triangle problem (a venerable example in testing circles); a logically complex function, NextDate; and an example that typifies 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 higher levels of testing in Chapters 11 through 17 uses four other examples: a simplified version of an automated teller machine (ATM), known here as the simple ATM system (SATM); the currency converter, an event-driven application typical of graphical user interface (GUI) applications; and the windshield wiper control device from the Saturn™ automobile. The last example, a garage door controller, illustrates some of the issues of "systems of systems."

For the purposes of code-based testing, pseudocode implementations of the three unit-level examples are given in this chapter. System-level descriptions of the SATM system, the currency converter, the Saturn windshield wiper system, and the garage door controller are given in Chapters 11 through 17. These applications are modeled with finite-state machines, variations of event-driven petri nets, selected StateCharts, and with the Universal Modeling Language (UML).

2.1 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).

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	‘ <text>
Data structure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
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>
	  Then <then clause>
	  Else <else clause>
	EndIf

Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> … Case *n*: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end> <loop body> EndFor
Pretest repetition	While <condition> <loop body> EndWhile

(*continued*)

Table 2.1 Generalized Pseudocode (Continued)

Language Element	*Generalized Pseudocode Construct*
Posttest repetition	Do <loop body> Until <condition>
Procedure definition (similarly for functions and o–o methods)	<procedure name> (Input: <list of variables>; Output: <list of variables>) <body> End <procedure name>
Interunit communication	Call <procedure name> (<list of variables>; <list of variables>)
Class/Object definition	<name> (<attribute 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> <unit list> End<program name>

2.2 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 and Richardson (1983, 1984), Chellappa (1987), and Hetzel (1988). There are others, but this list makes the point.

2.2.1 Problem Statement

Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. 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. $1 \le a \le 200$	c4. $a < b + c$
c2. $1 \le b \le 200$	c5. $b < a + c$
c3. $1 \le c \le 200$	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 c1, 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 c4, c5, and c6, one of four mutually exclusive outputs is given:

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

2.2.2 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 in Chapter 5.

2.2.3 Traditional Implementation

The traditional implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. Figure 2.2 is a flowchart for the improved version. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) pseudocode program given next. (These numbers correspond exactly to those in Pressman [1982].) This implementation shows its age; a better implementation is given in Section 2.2.4.

The variable "match" is used to record equality among pairs of the sides. A classic intricacy of the FORTRAN style is connected with the variable "match": notice that all three tests for the triangle inequality do not occur. If two sides are equal, say a and c, it is only necessary to compare a + c with b. (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). We will find this version useful later when we discuss infeasible program execution paths. That is the best reason for perpetuating this version. Notice that six ways are used to reach the NotATriangle box (12.1–12.6), and three ways are used to reach the Isosceles box (15.1–15.3).

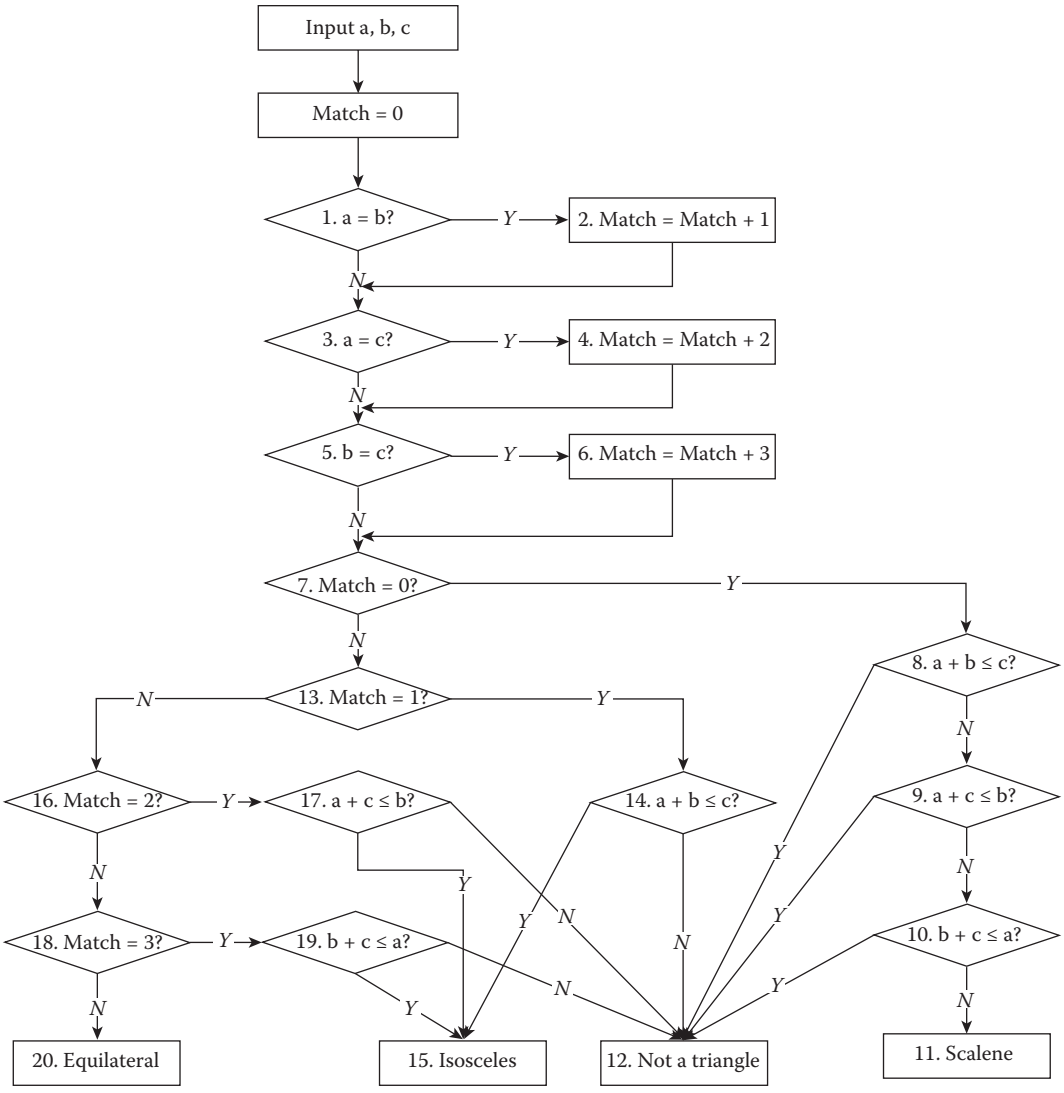

Figure 2.1 Flowchart for traditional triangle program implementation.

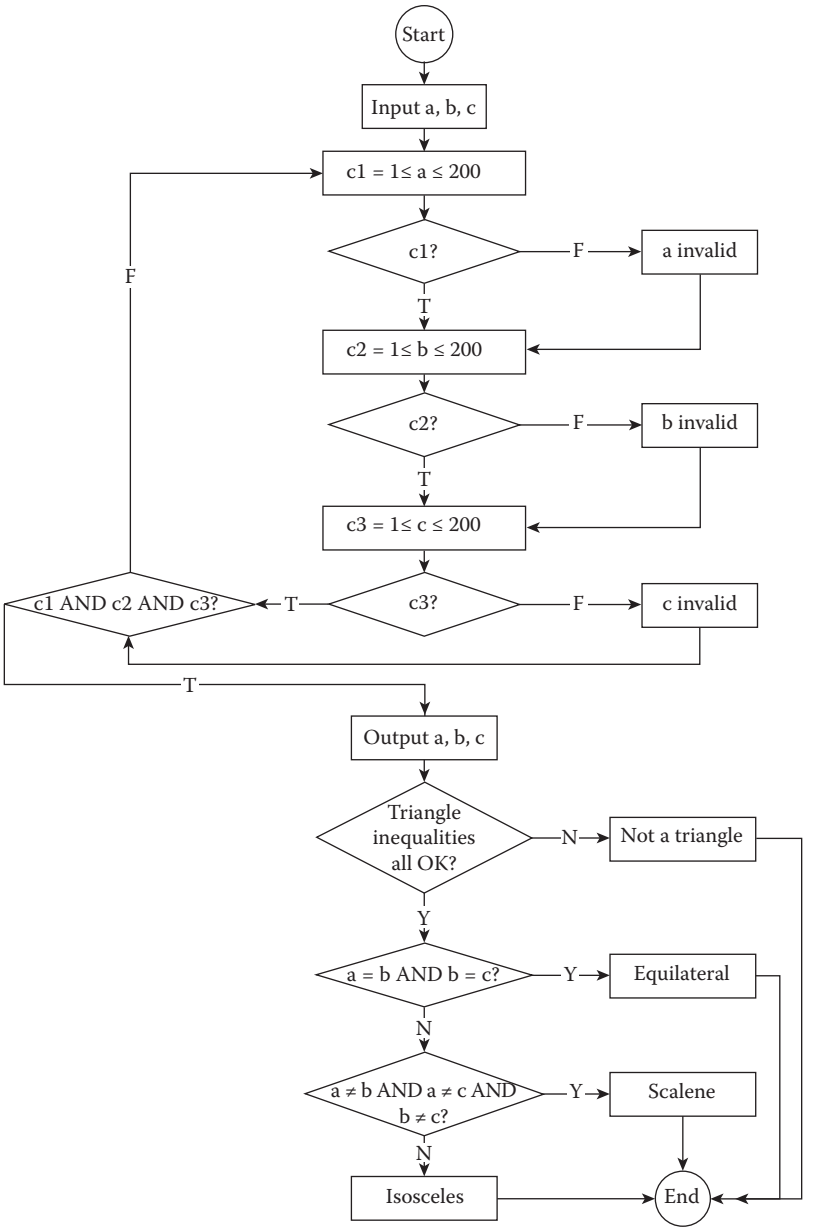

Figure 2.2 Flowchart for improved triangle program implementation.

The pseudocode for this is given next.

```
Program triangle1 'Fortran-like version
'
Dim a, b, c, match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a, b, c)
```

```
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
match = 0
If a = b                                                    '(1)
   Then match = match + 1                                   '(2)
EndIf
If a = c                                                    '(3)
   Then match = match + 2                                   '(4)
EndIf
If b = c                                                    '(5)
   Then match = match + 3                                   '(6)
EndIf
If match = 0                                                '(7)
   Then If (a + b)≤ c                                       '(8)
            Then Output("NotATriangle")                     '(12.1)
            Else If (b + c) ≤ a                             '(9)
                     Then Output("NotATriangle")            '(12.2)
                     Else If (a + c) ≤ b                    '(10)
                              Then Output("NotATriangle")   '(12.3)
                              Else Output ("Scalene")       '(11)
                          EndIf
                EndIf
          EndIf
   Else If match = 1                                        '(13)
            Then If (a + c) ≤ b                             '(14)
                     Then Output("NotATriangle")            '(12.4)
                     Else Output ("Isosceles")              '(15.1)
                EndIf
            Else If match=2                                 '(16)
                     Then If (a + c) ≤ b
                              Then Output("NotATriangle")   (12.5)
                              Else Output ("Isosceles")     '(15.2)
                          EndIf
                     Else If match = 3                      '(18)
                          Then If (b + c) ≤ a  '(19)
                              Then Output("NotATriangle")   '(12.6)
                              Else Output ("Isosceles")     '(15.3)
                          EndIf
                     Else Output ("Equilateral")            '(20)
                EndIf
           EndIf
      EndIf
EndIf
'
End Triangle1
```

2.2.4 Structured Implementations

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
```

Third version

```
Program triangle3'
Dim a, b, c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'Step 1: Get Input
Do
    Output("Enter 3 integers which are sides of a triangle")
    Input(a, b, c)
    c1 = (1 ≤ a) AND (a ≤ 300)
    c2 = (1 ≤ b) AND (b ≤ 300)
    c3 = (1 ≤ c) AND (c ≤ 300)
    If NOT(c1)
         Then Output("Value of a is not in the range of permitted values")
    EndIf
    If NOT(c2)
         Then Output("Value of b is not in the range of permitted values")
    EndIf
    If NOT(c3)
         ThenOutput("Value of c is not in the range of permitted values")
    EndIf
Until c1 AND c2 AND c3
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
```

```
EndIf
'Step 3: Determine Triangle Type
If IsATriangle
    Then If (a = b) AND (b = c)
            Then Output ("Equilateral")
            Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                    Then Output ("Scalene")
                    Else Output ("Isosceles")
                 EndIf
         EndIf
    Else Output("Not a Triangle")
EndIf
End triangle3
```

2.3 The NextDate Function

The complexity in the triangle program is due to the relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity—logical relationships among the input variables.

2.3.1 Problem Statement

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

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

As we did with the triangle program, we can make our problem statement more specific. 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 c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, "Value of month not in the range 1...12." Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: "Invalid Input Date."

2.3.2 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); thus, 1992, 1996, and 2000 are leap years, while the year 1900 is not a leap year. 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.

2.3.3 Implementations

```
Program NextDate1 'Simple version
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Output ("Enter today's date in the form MM DD YYYY")
Input (month, day, year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
 If day < 31
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = month + 1
 EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
 If day < 30
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = month + 1
 EndIf
Case 3: month Is 12: 'December
 If day < 31
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = 1
    If year = 2012
      Then Output ("2012 is over")
      Else tomorrow.year = year + 1
 EndIf
Case 4: month is 2: 'February
 If day < 28
   Then tomorrowDay = day + 1
   Else
    If day = 28
      Then If ((year is a leap year)
               Then tomorrowDay = 29 'leap year
               Else 'not a leap year
                 tomorrowDay = 1
                 tomorrowMonth = 3
           EndIf
    Else If day = 29
           Then If ((year is a leap year)
                    Then tomorrowDay = 1
```

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

Program NextDate2   Improved version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
'
Do
 Output ("Enter today's date in the form MM DD YYYY")
 Input (month, day, year)
 c1 = (1 ≤ day) AND (day ≤ 31)
 c2 = (1 ≤ month) AND (month ≤ 12)
 c3 = (1812 ≤ year) AND (year ≤ 2012)
 If NOT(c1)
  Then Output("Value of day not in the range 1..31")
 EndIf
 If NOT(c2)
  Then Output("Value of month not in the range 1..12")
 EndIf
 If NOT(c3)
  Then Output("Value of year not in the range 1812..2012")
 EndIf
Until c1 AND c2 AND c2

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

```
 If day < 31
    Then tomorrowDay = day + 1
    Else
     tomorrowDay = 1
     tomorrowMonth = 1
     If year = 2012
        Then Output ("Invalid Input Date")
        Else tomorrow.year = year + 1
     EndIf
 EndIf
Case 4: month is 2: 'February
 If day < 28
    Then tomorrowDay = day + 1
    Else
     If day = 28
        Then
         If (year is a leap year)
            Then tomorrowDay = 29 'leap day
            Else 'not a leap year
              tomorrowDay = 1
              tomorrowMonth = 3
         EndIf
        Else
         If day = 29
            Then
              If (year is a leap year)
                 Then tomorrowDay = 1
                      tomorrowMonth = 3
                 Else
                  If day > 29
                     Then Output("Invalid Input Date")
                  EndIf
              EndIf
         EndIf
     EndIf
 EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate2
```

2.4 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. Our main use of this example will be in our discussion of data flow and slice-based testing.

2.4.1 Problem Statement

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to

sell at least one lock, one stock, and one barrel (but not necessarily one complete rifle) per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. 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.

2.4.2 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 US 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.

2.4.3 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 (Figure 2.3).

The ATM described here is minimal, yet it contains an interesting variety of functionality and interactions that typify the client side of client–server systems.

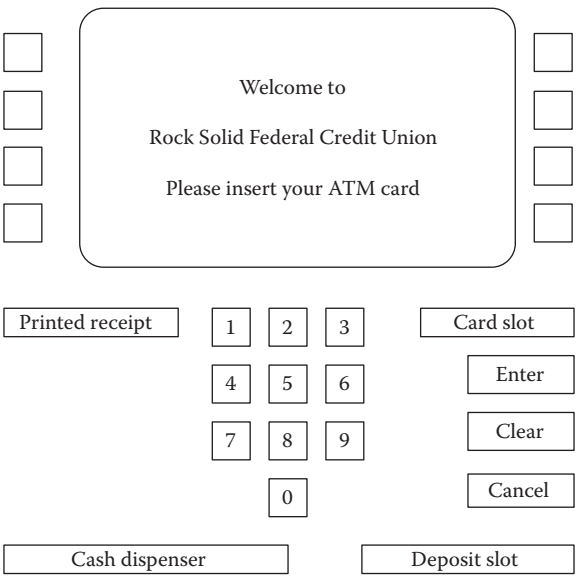

Figure 2.3 SATM terminal.

2.5.1 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. For simplicity, these transactions can only be done on a checking account.

When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the 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.

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

On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount. If a problem occurs with the deposit envelope slot, the

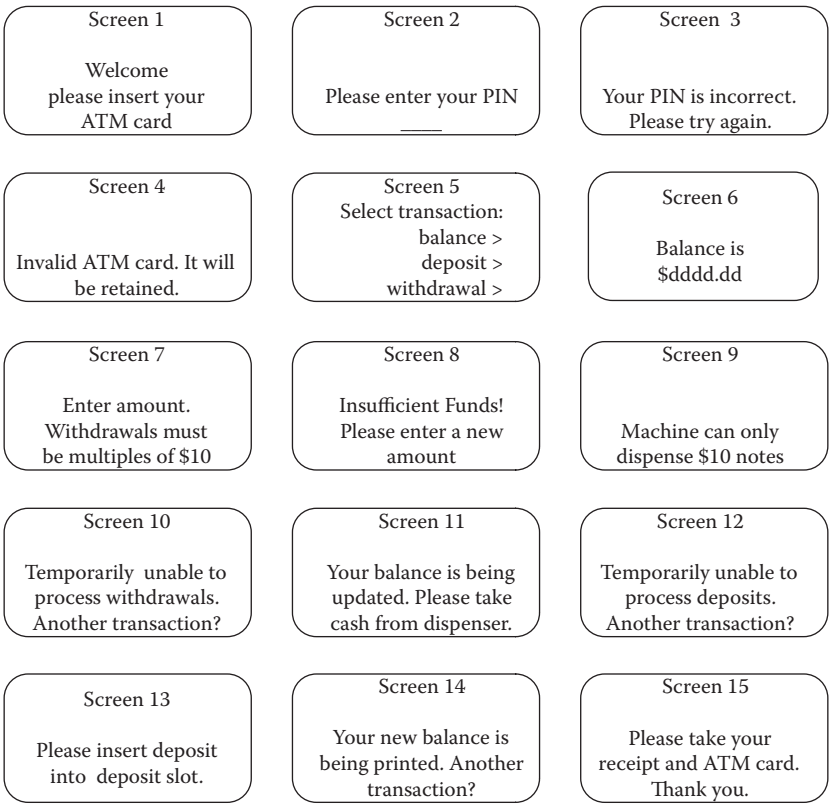

Figure 2.4 SATM screens.

system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14.

If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14.

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

2.5.2 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 customers from taking out more than their actual balance if they go 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.

2.6 The Currency Converter

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

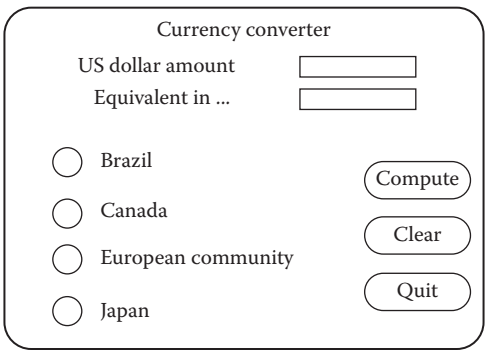

Figure 2.5 Currency converter graphical user interface.

The application converts US dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label; 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 US dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, or the Quit button. Clicking on the Compute button results in the conversion of the US dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the US dollar amount, and the equivalent currency amount and the associated label. Clicking on the Quit button ends the application. This example nicely illustrates a description with UML and an object-oriented implementation, which we will use in Chapter 15.

2.7 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

2.8 Garage Door Opener

A system to open a garage door is composed of several components: a drive motor, a drive chain, the garage door wheel tracks, a lamp, and an electronic controller. This much of the system is powered by commercial 110 V electricity. Several devices communicate with the garage door controller—a wireless keypad (usually in an automobile), a digit keypad on the outside of the garage door, and a wall-mounted button. In addition, there are two safety features, a laser beam near the floor and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet), the door immediately stops, and then reverses direction until the door is fully open. If the door encounters an obstacle while it is closing (say a child's tricycle left in the path of the door), the door stops and reverses direction until it is fully open. There is a third way to stop a door in motion, either when it is closing or opening. A signal from any of the three devices (wireless keypad, digit keypad, or wall-mounted control button). The response to any of these signals is different—the door stops in place. A subsequent signal from any of the devices starts the door in the same direction as when it was stopped. Finally, there are sensors that detect when the door has moved to one of the extreme positions, either fully open

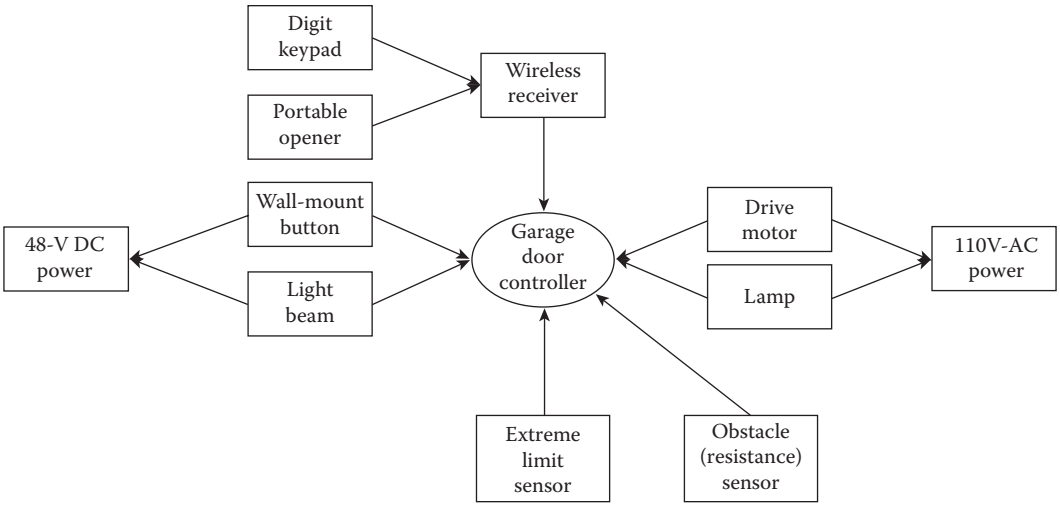

Figure 2.6 SysML diagram of garage door controller.

or fully closed. When the door is in motion, the lamp is lit, and remains lit for approximately 30 seconds after the door reaches one of the extreme positions.

The three signaling devices and the safety features are optional additions to the basic garage door opener. This example will be used in Chapter 17 in the discussion of systems of systems. For now, a SysML context diagram of the garage door opener is given in Figure 2.6.

EXERCISES

1. Revisit the traditional triangle program flowchart in Figure 2.1. Can the variable match ever have the value of 4? Of 5? Is it ever possible to "execute" the following sequence of numbered boxes: 1, 2, 5, 6?
2. Recall the discussion from Chapter 1 about the relationship between the specification and the implementation of a program. If you study the implementation of NextDate carefully, you will see a problem. Look at the CASE clause for 30-day months (4, 6, 9, 11). There is no special action for day = 31. Discuss whether this implementation is correct. Repeat this discussion for the treatment of values of day 29 in the CASE clause for February.
3. In Chapter 1, we mentioned that part of a test case is the expected output. What would you use as the expected output for a NextDate test case of June 31, 1812? Why?
4. One common addition to the triangle problem is to check for right triangles. Three sides constitute a right triangle if the Pythagorean relationship is satisfied: $c^2 = a^2 + b^2$. This change makes it convenient to require that the sides be presented in increasing order, that is, $a \le b \le c$. Extend the Triangle3 program to include the right triangle feature. We will use this extension in later exercises.
5. What will the Triangle2 program do for the sides –3, –3, 5? Discuss this in terms of the considerations we made in Chapter 1.
6. The function YesterDate is the inverse of NextDate. Given a month, day, year, YesterDate returns the date of the day before. Develop a program in your favorite language (or our generalized pseudocode) for YesterDate. We will also use this as a continuing exercise.

7. Part of the art of GUI design is to prevent user input errors. Event-driven applications are particularly vulnerable to input errors because events can occur in any order. As the given definition stands, a user could enter a US dollar amount and then click on the compute button without selecting a country. Similarly, a user could select a country and then click on the compute button without inputting a dollar amount. GUI designers use the concept of "forced navigation" to avoid such situations. In Visual Basic, this can be done using the visibility properties of various controls. Discuss how you could do this.
8. The CRC Press website (http://www.crcpress.com/product/isbn/9781466560680) contains some software supplements for this book. There is a series of exercises that I use in my graduate class in software testing; the first part of a continuing exercise is to use the naive.xls (runs in most versions of Microsoft Excel) program to test the triangle, NextDate, and commission problems. The spreadsheet lets you postulate test cases and then run them simply by clicking on the "Run Test Cases" button. As a start to becoming a testing craftsperson, use naive.xls to test our three examples in an intuitive (hence "naive") way. There are faults inserted into each program. If (when) you find failures, try to hypothesize the underlying fault. Keep your results for comparison to ideas in Chapters 5, 6, and 9.

References

Brown, J.R. and Lipov, M., Testing for software reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, April 1975, pp. 518–527.

Chellappa, M., Nontraversible paths in a program, *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 6, June 1987, pp. 751–756.

Clarke, L.A. and Richardson, D.J., The application of error sensitive strategies to debugging, *ACM SIGSOFT Software Engineering Notes,* Vol. 8, No. 4, August 1983.

Clarke, L.A. and Richardson, D.J., A reply to Foster's comment on "The Application of Error Sensitive Strategies to Debugging," *ACM SIGSOFT Software Engineering Notes,* Vol. 9, No. 1, January 1984.

Gruenberger, F., Program testing, the historical perspective, in *Program Test Methods,* William C. Hetzel, Ed., Prentice-Hall, New York, 1973, pp. 11–14.

Hetzel, Bill, *The Complete Guide to Software Testing,* 2nd ed., QED Information Sciences, Inc., Wellesley, MA, 1988.

Inglis, Stuart J., *Planets, Stars, and Galaxies*, 4th Ed., John Wiley & Sons, New York, 1961.

Myers, G.J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.