Chapter 2

Examples

Three examples will be used throughout Parts II and III 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 in Part V.

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 are given in Part IV. 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), in Part V.

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>
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> (<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>

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 (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 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.	$1 \le a \le 200$	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 c1, c2, and c3, 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. 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).) I do not really like this implementation very much, so a more structured implementation is given in Section 2.2.4.

```
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)
```

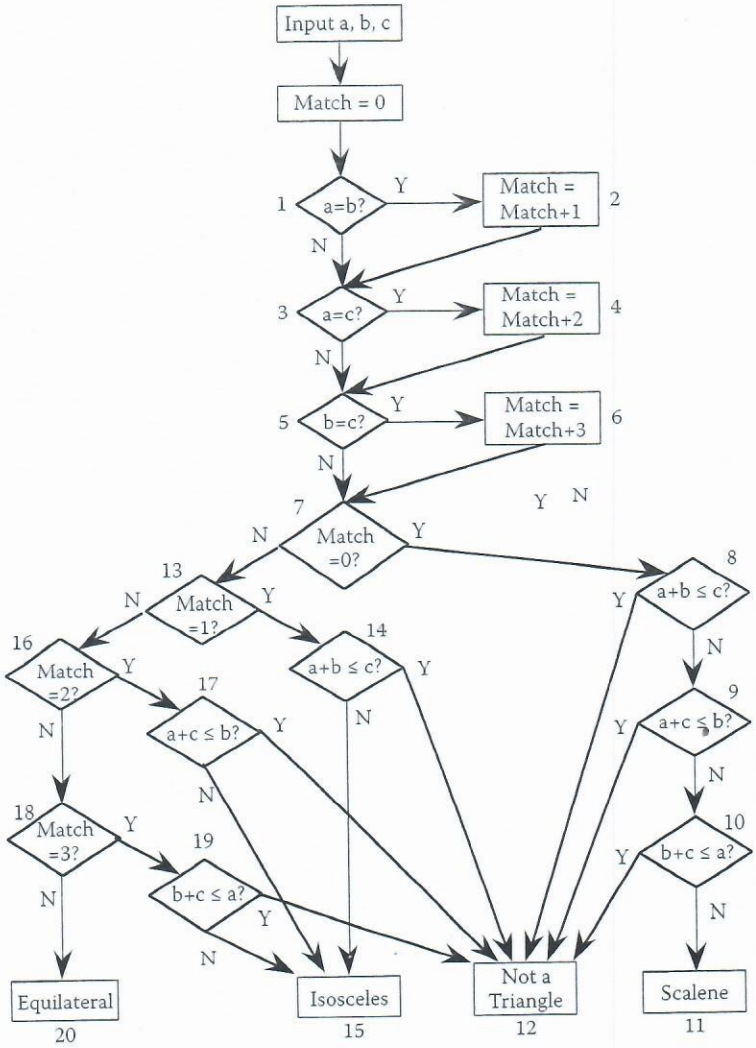

Figure 2.1 Flowchart for the traditional triangle program implementation.

```
If a = c                                                    '(3)
   Then match = match + 2                                   '(4)
EndIf
If b = c                                                    '(5)
   Then match = match + 3                                   '(6)
EndIf
If match = 0                                                '(7)
   Then   If (a+b)<=c                                       '(8)
      Then     Output ("NotATriangle")                      '(12.1)
      Else   If (b+c)<=a                                    '(9)
         Then   Output ("NotATriangle")                     '(12.2)
```

```
        Else      If (a+c)<=b                                    '(10)
           Then      Output ("NotATriangle")                    '(12.3)
           Else      Output ("Scalene")                          '(11)
                  EndIf
            EndIf
          EndIf
Else  If match=1                                                 '(13)
    Then     If (a+c) <=b                                        '(14)
         Then      Output ("NotATriangle")                      '(12.4)
         Else      Output ("Isosceles")                         '(15.1)
      EndIf
    Else      If match=2                                         '(16)
               Then  If (a+c)<=b
                       Then  Output ("NotATriangle")           '(12.5)
                       Else  Output ("Isosceles")              '(15.2)
                     EndIf
               Else  If match=3                                  '(18)
                       Then  If (b+c)<=a                         '(19)
                              Then  Output ("NotATriangle") '(12.6)
                              Else  Output ("Isosceles") ' '(15.3)
                             EndIf
                       Else  Output ("Equilateral")             '(20)
                     EndIf
              EndIf
      EndIf
EndIf
'
End Triangle1
```

The 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). We will find this version useful later in Part III when we discuss infeasible program execution paths. That is the only reason for perpetuating this version.

Notice that six ways are used to reach the NotATriangle box (12.1 to 12.6), and three ways are used to reach the Isosceles box (15.1 to 15.3).

2.2.4 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

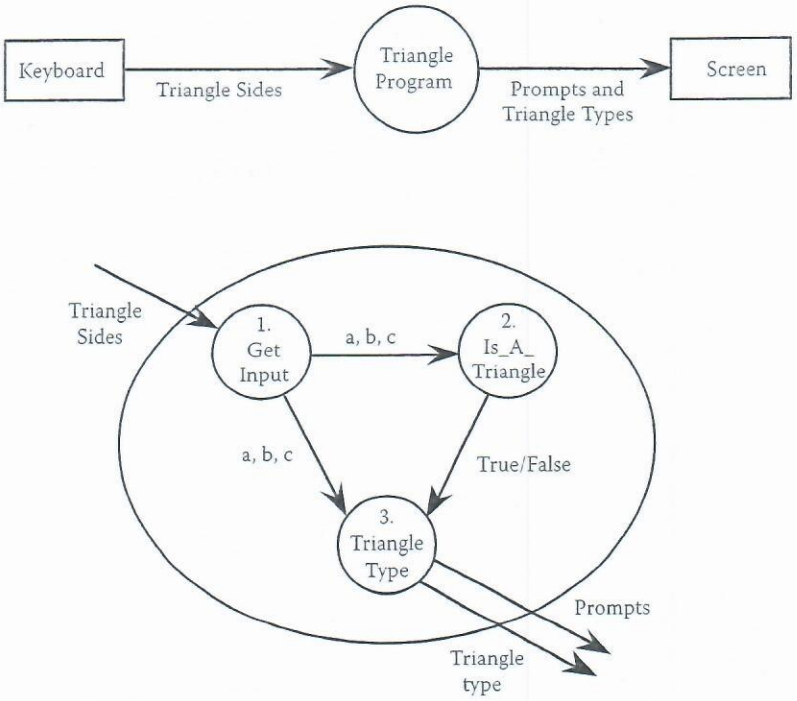

Figure 2.2 Dataflow diagram for a structured triangle program implementation.

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

```
      Then   If (a = b) AND (b = c)
               Then Output ("Equilateral")
               Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                          Then    Output ("Scalene")
                          Else    Output ("Isosceles")
                       EndIf
             EndIf
      Else      Output ("Not a Triangle")
EndIf
'
End triangle2

Program triangle3 'Structured programming version of improved
specification
'
Dim a,b,c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'
'Step 1: Get Input
Do
      Output("Enter 3 integers which are sides of a triangle")
      Input(a,b,c)
      c1 = (1 <= a) AND (a <= 200)
      c2 = (1 <= b) AND (b <= 200)
      c3 = (1 <= c) AND (c <= 200)
      If NOT(c1)
        Then  Output("Value of a is not in the range of
        permitted values")
      EndIf
      If NOT (c2)
        Then  Output("Value of b is not in the range of
        permitted values")
      EndIf
      If NOT(c3)
        Then  Output ("Value of c is not in the range of
        permitted values")
      EndIf
Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < (b + c)) AND (b < (a + c)) AND (c < (a + b))
      Then IsATriangle = True
      Else IsATriangle = False
```

```
'
'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 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, 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:

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 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 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), 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.

2.3.3 Implementation

```
Program NextDate1       'Simple version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
'
Output ("Enter today's date in the form MM DD YYYY")
Input (month,day,year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
   If day < 31
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = month + 1
   EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
   If day < 30
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = month + 1
   EndIf
Case 3: month Is 12: 'December
   If day < 31
     Then tomorrowDay = day + 1
     Else
       tomorrowDay = 1
       tomorrowMonth = 1
       If year = 2012
          Then Output ("2012 is over")
          Else tomorrow.year = year + 1
       EndIf
   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)
```

```
            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

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)
   EndIf
     Then      Output ("Value of month not in the range 1..12")
   If NOT(c3)
     Then   Output ("Value of year not in the range 1812..2012")
   EndIf
 Until c1 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 = 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
```

```
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.

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 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.

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 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.

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 scop
The automated teller machine described here is a refinement of that in Topper (1993); it contai
an interesting variety of functionality and interactions that typify the client side of client/serv

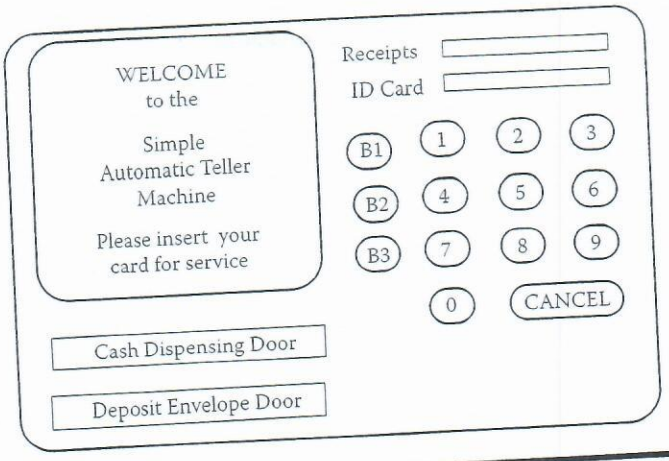

Figure 2.3 The SATM terminal.

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. 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.

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.

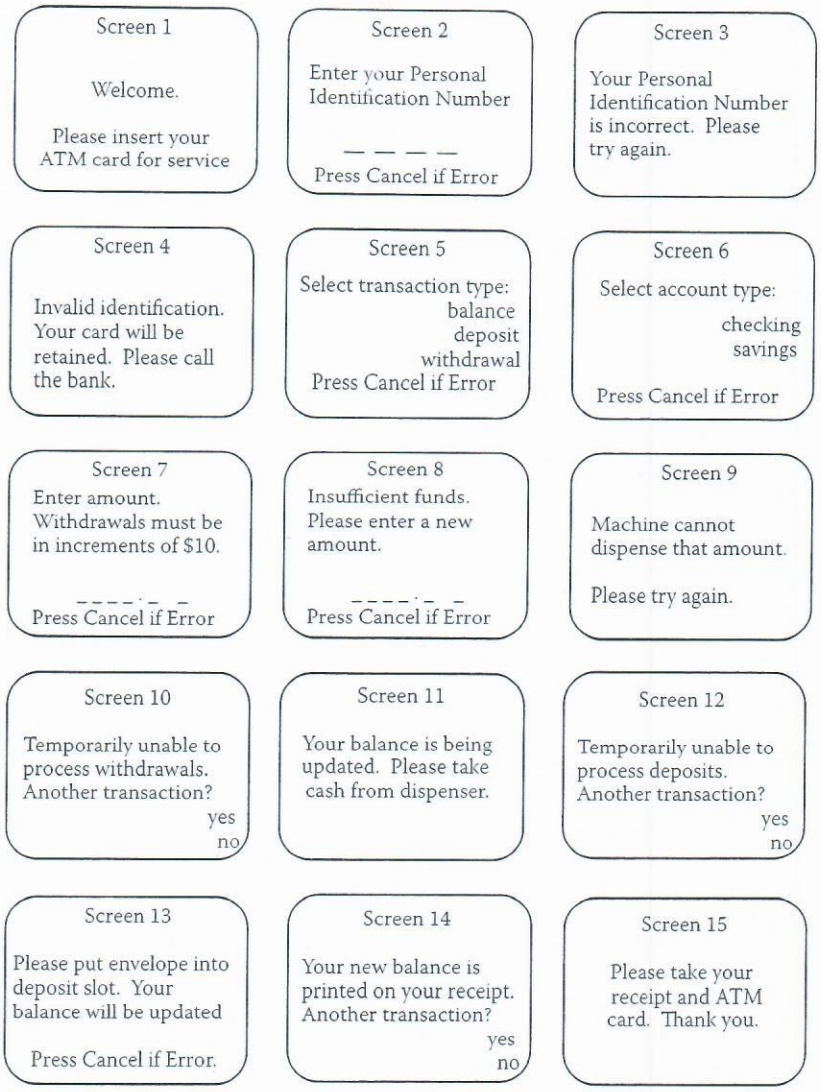

Figure 2.4 SATM screens.

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. 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

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.

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 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.

2.6 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, or the Quit button. Clicking on the Compute button

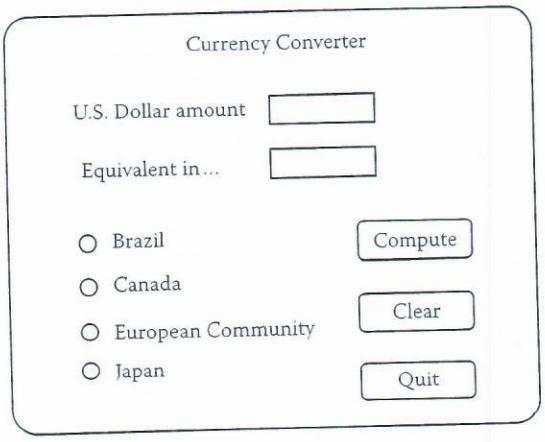

Figure 2.5 Currency converter GUI.

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. This example nicely illustrates a description with UML and an object-oriented implementation we will use in Part V.

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

We will use this example in our discussion of interaction testing in Chapter 15.

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, B., *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.

Topper, A. et al., *Structured Methods: Merging Models, Techniques, and CASE*, McGraw-Hill, New York, 1993.

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