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 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).

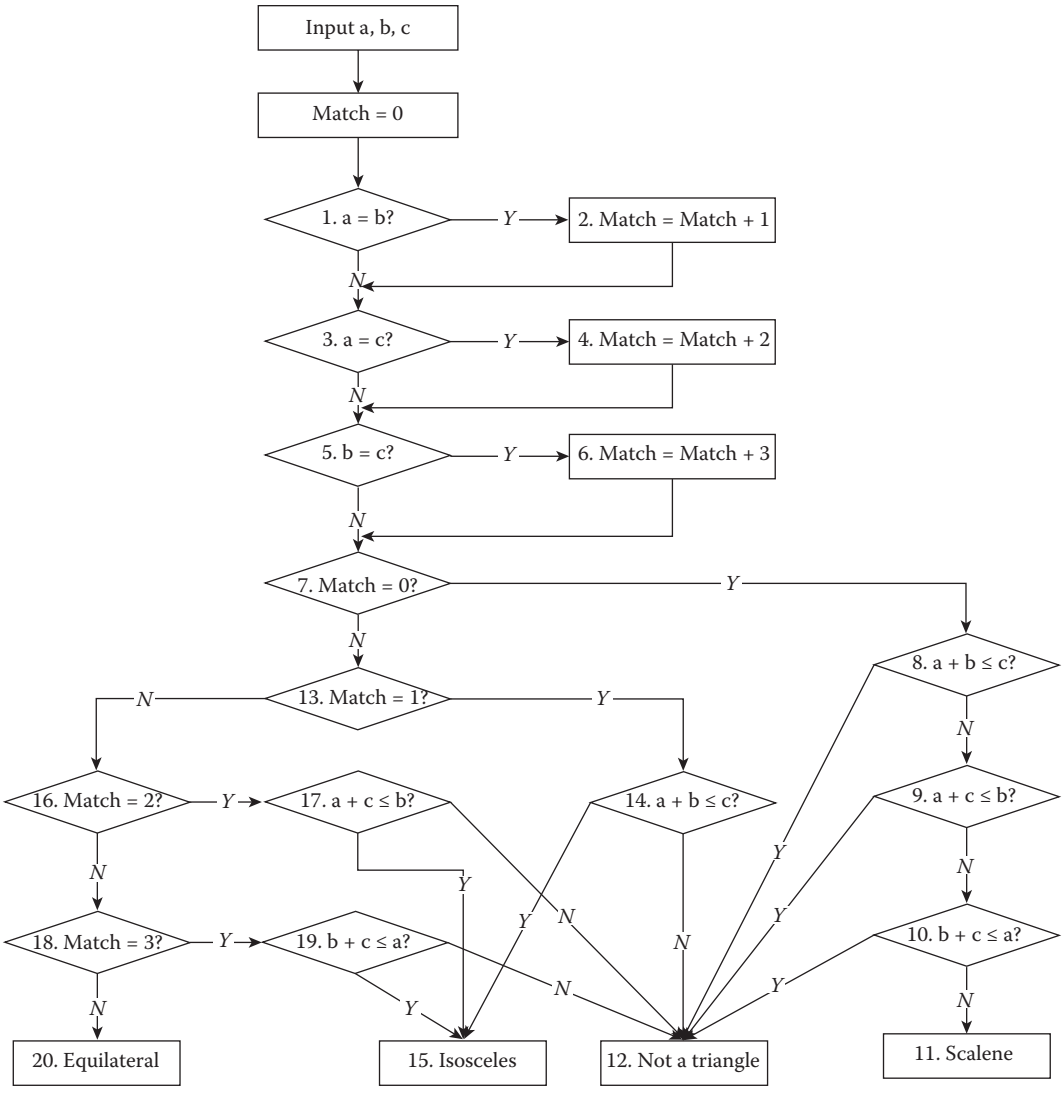

Figure 2.1 Flowchart for traditional triangle program implementation.

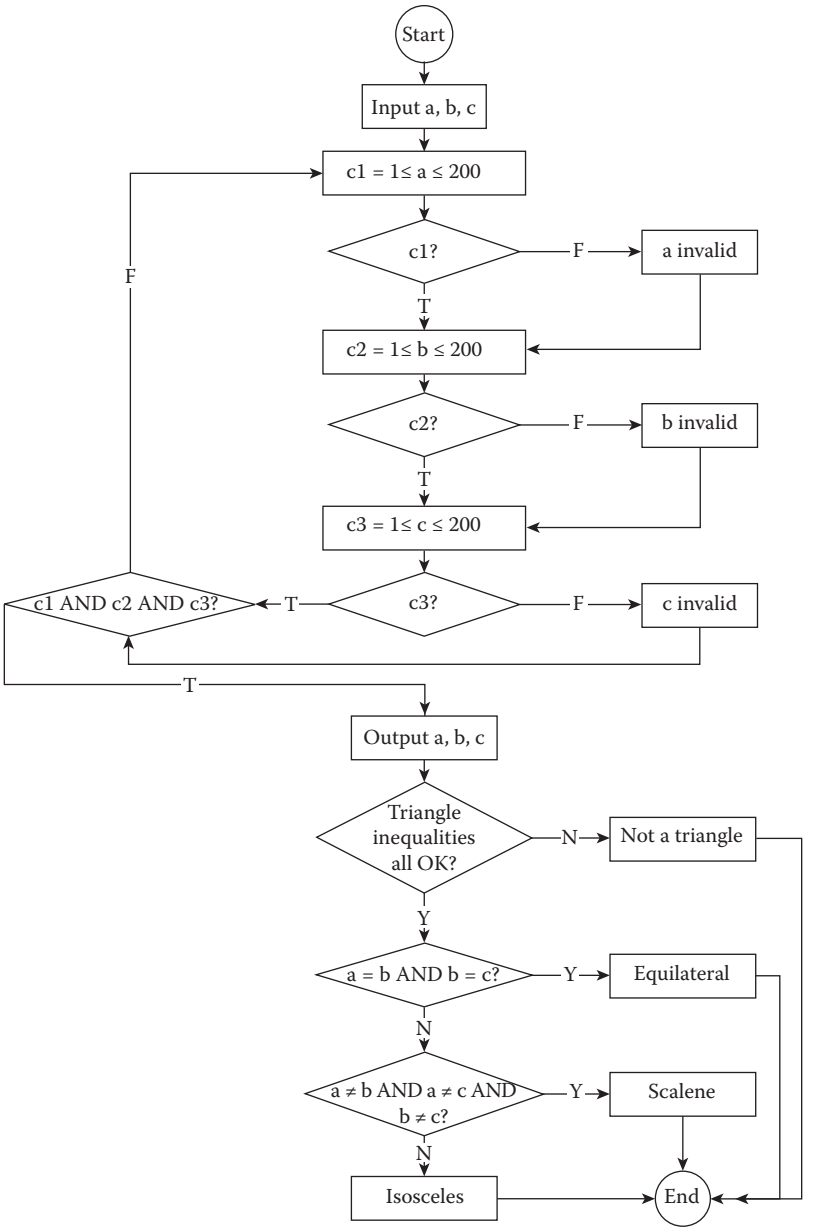

Figure 2.2 Flowchart for improved triangle program implementation.

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.

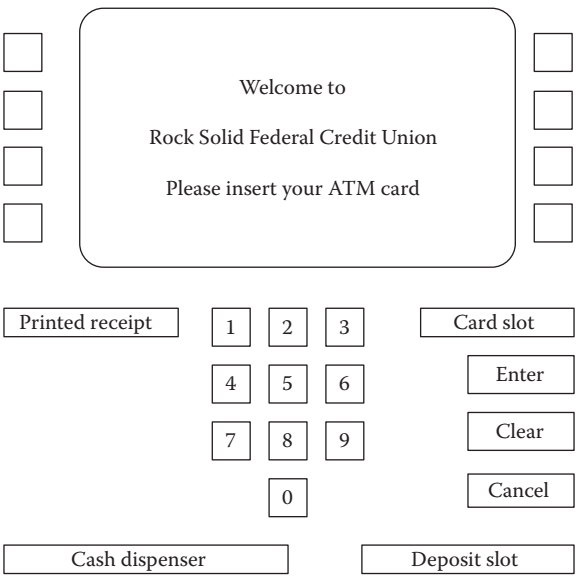

Figure 2.3 SATM terminal.

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

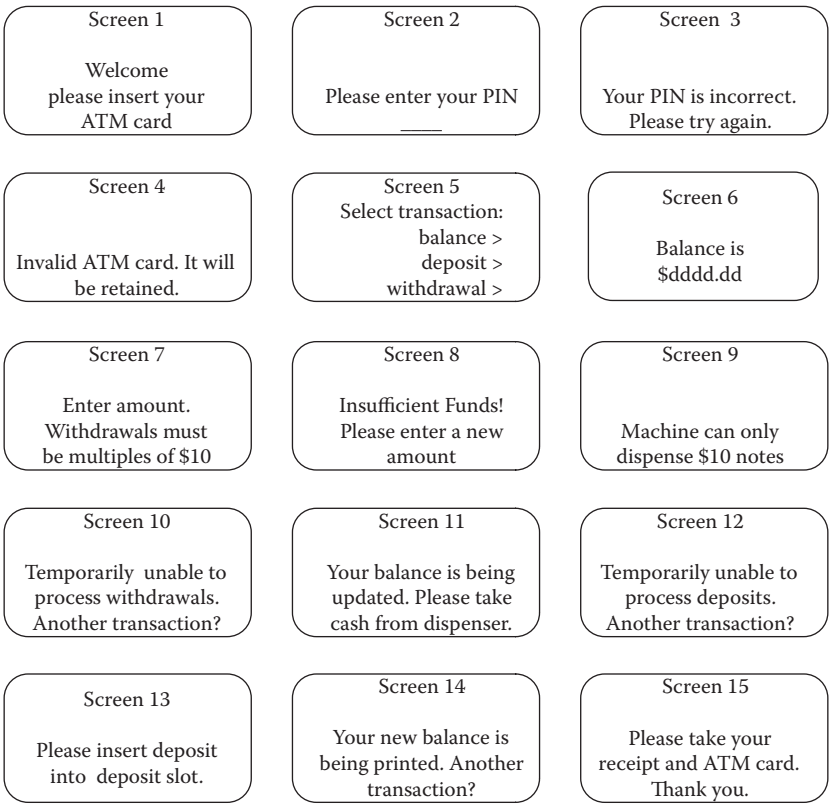

Figure 2.4 SATM screens.

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.

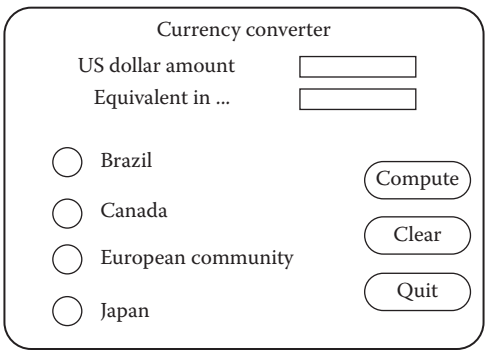

Figure 2.5 Currency converter graphical user interface.

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

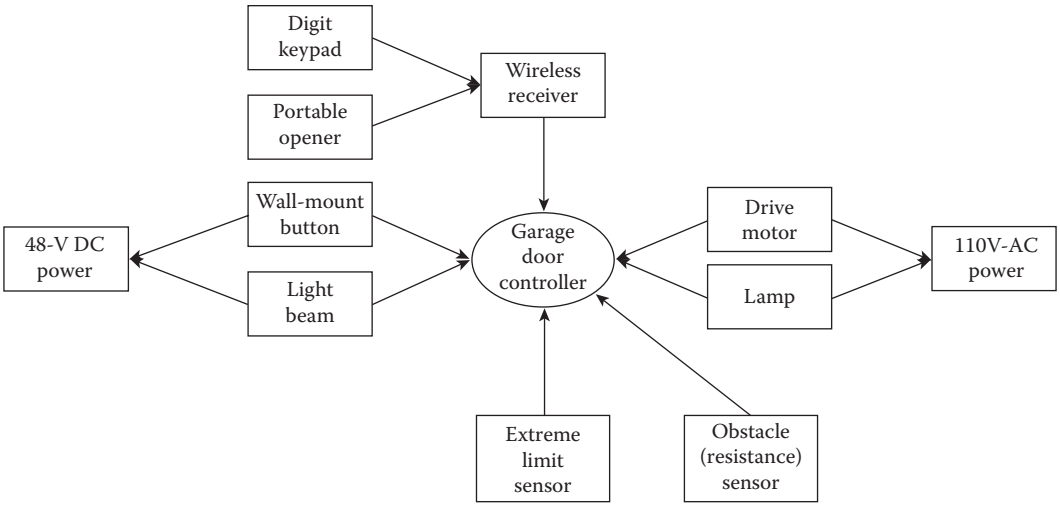

Figure 2.6 SysML diagram of garage door controller.

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.

Chapter 11

Life Cycle–Based Testing

In this chapter, we begin with various models of the software development life cycle in terms of the implications these life cycles have for testing. We took a general view in Chapter 1, where we identified three levels (unit, integration, and system) in terms of symmetries in the waterfall model of software development. This view has been relatively successful for decades, and these levels persist; however, the advent of alternative life cycle models mandates a deeper look at these views of testing. We begin with the traditional waterfall model, mostly because it is widely understood and is a reference framework for the more recent models. Then we look at derivatives of the waterfall model, and finally some mainline agile variations.

We also make a major shift in our thinking. We are more concerned with how to represent the item tested because the representation may limit our ability to identify test cases. Take a look at the papers presented at the leading conferences (professional or academic) on software testing—you will find nearly as many presentations on specification models and techniques as on testing techniques. Model-Based Testing (MBT) is the meeting place of software modeling and testing at all levels.

11.1 Traditional Waterfall Testing

The traditional model of software development is the waterfall model, which is illustrated in Figure 11.1. It is sometimes drawn as a V as in Figure 11.2 to emphasize how the basic levels of testing reflect the early waterfall phases. (In ISTQB circles, this is known as the "V-Model.") In this view, information produced in one of the development phases constitutes the basis for test case identification at that level. Nothing controversial here: we certainly would hope that system test cases are clearly correlated with the requirements specification, and that unit test cases are derived from the detailed design of the unit. On the upper left side of the waterfall, the tight what/how cycles are important. They underscore the fact that the predecessor phase defines what is to be done in the successor phase. When complete, the successor phase states how it accomplishes "what" was to be done. These are also ideal points at which to conduct software reviews (see Chapter 22). Some humorists assert that these phases are the fault creation phases, and those on the right are the fault detection phases.

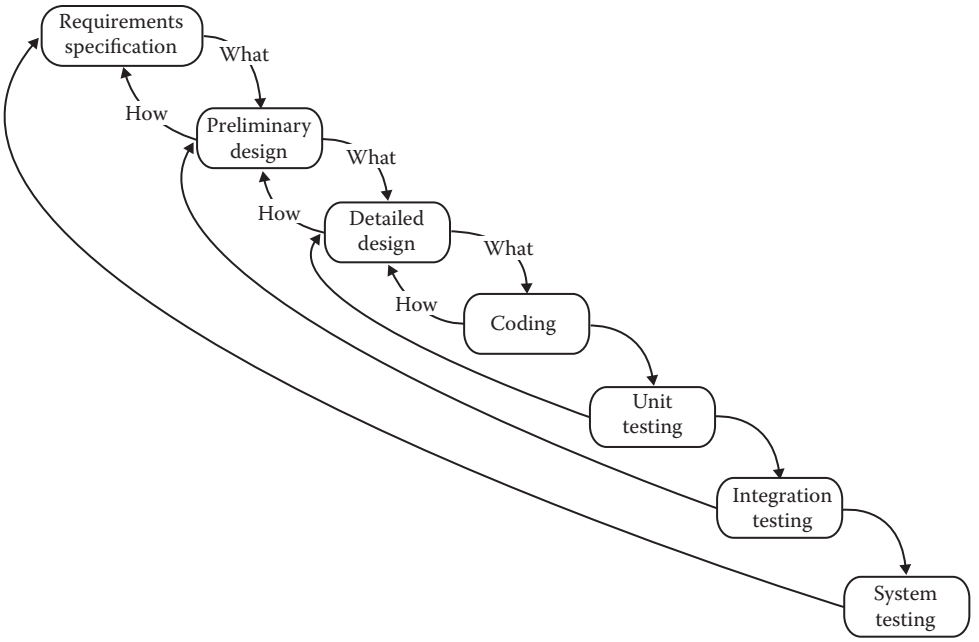

Figure 11.1 The waterfall life cycle.

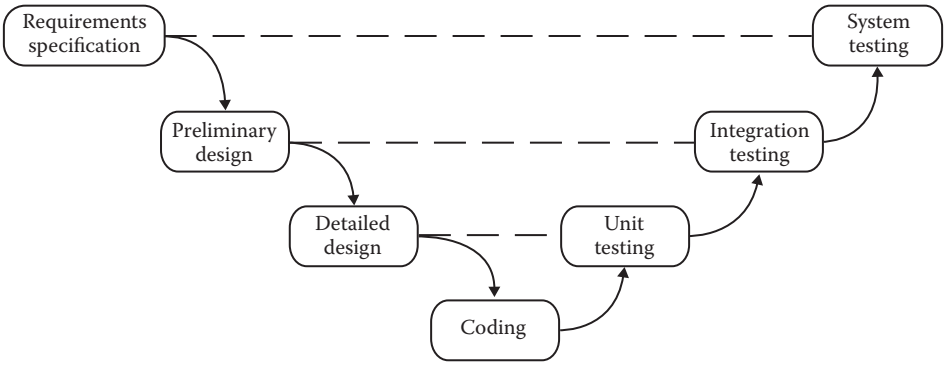

Figure 11.2 The waterfall life cycle as the V-Model.

Two observations: a clear presumption of functional testing is used here, and an implied bottom–up testing order is used. Here, "bottom–up" refers to levels of abstraction—unit first, then integration, and finally, system testing. In Chapter 13, bottom–up also refers to a choice of orders in which units are integrated (and tested).

Of the three main levels of testing (unit, integration, and system), unit testing is best understood. Chapters 5 through 10 are directed at the testing theory and techniques applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom–up approach sheds some insight: test the individual components, and then integrate these into subsystems until the entire system is tested. System testing should be something that the

customer (or user) understands, and it often borders on customer acceptance testing. Generally, system testing is functional instead of structural; this is mostly due to the lack of higher-level structural notations.

11.1.1 Waterfall Testing

The waterfall model is closely associated with top–down development and design by functional decomposition. The end result of preliminary design is a functional decomposition of the entire system into a tree-like structure of functional components. With such a decomposition, top–down integration would begin with the main program, checking the calls to the next-level units, and so on until the leaves of the decomposition tree are reached. At each point, lower-level units are replaced by stubs—throwaway code that replicates what the lower-level units would do when called. Bottom–up integration is the opposite sequence, starting with the leaf units and working up toward the main program. In bottom–up integration, units at higher levels are replaced by drivers (another form of throwaway code) that emulate the procedure calls. The "big bang" approach simply puts all the units together at once, with no stubs or drivers. Whichever approach is taken, the goal of traditional integration testing is to integrate previously tested units with respect to the functional decomposition tree. Although this describes integration testing as a process, discussions of this type offer little information about the methods or techniques. We return to this in Chapter 13.

11.1.2 Pros and Cons of the Waterfall Model

In its history since the first publication in 1968, the waterfall model has been analyzed and critiqued repeatedly. The earliest compendium was by Agresti (1986), which stands as a good source. Agresti observes that

- The framework fits well with hierarchical management structures.
- The phases have clearly defined end products (exit criteria), which in turn are convenient for project management.
- The detailed design phase marks the starting point where individuals responsible for units can work in parallel, thereby shortening the overall project development interval.

More importantly, Agresti highlights major limitations of the waterfall model. We shall see that these limitations are answered by the derived life cycle models. He observes that

- There is a very long feedback cycle between requirements specification and system testing, in which the customer is absent.
- The model emphasizes analysis to the near exclusion of synthesis, which first occurs at the point of integration testing.
- Massive parallel development at the unit level may not be sustainable with staffing limitations.
- Most important, "perfect foresight" is required because any faults or omissions at the requirements level will penetrate through the remaining life cycle phases.

The "omission" part was particularly troubling to the early waterfall developers. As a result, nearly all of the papers of requirements specification demanded consistency, completeness, and clarity. Consistency is impossible to demonstrate for most requirements specification techniques

(decision tables are an exception), and the need for clarity is obvious. The interesting part is completeness—all of the successor life cycles assume incompleteness, and depend on some form of iteration to gradually arrive at "completeness."

11.2 Testing in Iterative Life Cycles

Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model just mentioned. Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on iteration and composition. Decomposition is a perfect fit both to the top–down progression of the waterfall model and to the bottom–up testing order, but it relies on one of the major weaknesses of waterfall development cited by Agresti (1986)—the need for "perfect foresight." Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, no opportunity is available for feedback from the customer. Composition, on the other hand, is closer to the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions.

A very nice analogy can be applied to positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's *David*: start with a piece of marble, and simply chip away all non-*David*. Positive sculpture is often done with a pliable medium, such as wax. The central shape is approximated, and then wax is either added or removed until the desired shape is attained. The wax original is then cast in plaster. Once the plaster hardens, the wax is melted out, and the plaster "negative" is used as a mold for molten bronze. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away and restarted. (A museum in Florence, Italy, contains half a dozen such false starts to the *David*.) With positive sculpture, the erroneous part is simply removed and replaced. We will see this is the defining essence of the agile life cycle models. The centrality of composition in the alternative models has a major implication for integration testing.

11.2.1 Waterfall Spin-Offs

There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the spiral model (Boehm, 1988). Each of these involves a series of increments or builds as shown in Figure 11.3. It is important to keep preliminary design as an integral phase rather than to try to amortize such high-level design across a series of builds. (To do so usually results in unfortunate consequences of design choices made during the early builds that are regrettable in later builds.) This single design step cannot be done in the evolutionary and spiral models. This is also a major limitation of the bottom–up agile methods.

Within a build, the normal waterfall phases from detailed design through testing occur with one important difference: system testing is split into two steps—regression and progression testing. The main impact of the series of builds is that regression testing becomes necessary. The goal of regression testing is to ensure that things that worked correctly in the previous build still work with the newly added code. Regression testing can either precede or follow integration testing, or possibly occur in both places. Progression testing assumes that regression testing was successful and that the new functionality can be tested. (We like to think that the addition of new code represents progress, not a regression.) Regression testing is an absolute necessity in a series of builds

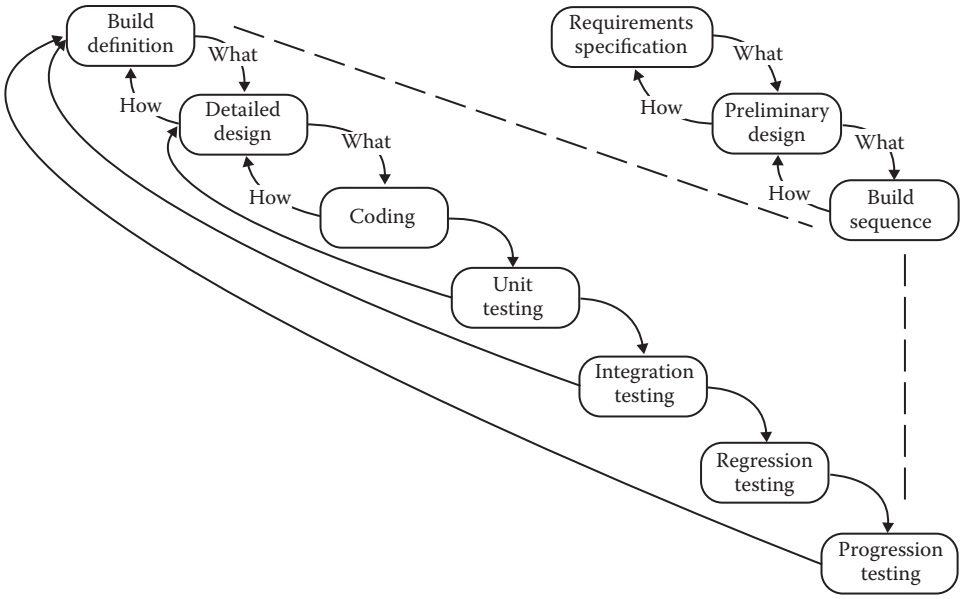

Figure 11.3 Iterative development.

because of the well-known ripple effect of changes to an existing system. (The industrial average is that one change in five introduces a new fault.)

Evolutionary development is best summarized as client-based iteration. In this spin-off, a small initial version of a product is given to users who then suggest additional features. This is particularly helpful in applications for which time-to-market is a priority. The initial version might capture a segment of the target market, and then that segment is "locked in" to future evolutionary versions. When these customers have a sense that they are "being heard," they tend to be more invested in the evolving product.

Barry Boehm's spiral model has some of the flavor of the evolutionary model. The biggest difference is that the increments are determined more on the basis of risk rather than on client suggestions. The spiral is superimposed on an *x*–*y* coordinate plane, with the upper left quadrant referring to determining objectives, the upper right to risk analysis, the lower right refers to development (and test), and the lower left is for planning the next iteration. These four phases—determine objectives, analyze risk, develop and test, and next iteration planning—are repeated in an evolutionary way. At each evolutionary step, the spiral enlarges.

There are two views of regression testing: one is to simply repeat the tests from the previous iteration; the other is to devise a smaller set of test cases specifically focused on finding affected faults. Repeating a full set of previous integration tests is fine in an automated testing environment, but is undesirable in a more manual environment. The expectation of test case failure is (or should be) lower for regression testing compared to that for progression testing. As a guideline, regression tests might fail in only 5% of the repeated progression tests. This may increase to 20% for progression tests. If regression tests are performed manually, there is an interesting term for special regression test cases: Soap Opera Tests. The idea is to have long, complex regression tests, akin to the complicated plot lines in television soap operas. A soap opera test case could fail in many ways, whereas a progression test case should fail for only a very few reasons. If a soap opera test case fails, clearly more focused testing is required to localize the fault. We will see this again in Chapter 20 on all-pairs testing.

The differences among the three spin-off models are due to how the builds are identified. In incremental development, the motivation for separate builds is usually to flatten the staff profile. With pure waterfall development, there can be a huge bulge of personnel for the phases from detailed design through unit testing. Many organizations cannot support such rapid staff fluctuations, so the system is divided into builds that can be supported by existing personnel. In evolutionary development, the presumption of a build sequence is still made, but only the first build is defined. On the basis of that, later builds are identified, usually in response to priorities set by the customer/user, so the system evolves to meet the changing needs of the user. This foreshadows the customer-driven tenet of the agile methods. The spiral model is a combination of rapid prototyping and evolutionary development, in which a build is defined first in terms of rapid prototyping and then is subjected to a go/no-go decision based on technology-related risk factors. From this, we see that keeping preliminary design as an integral step is difficult for the evolutionary and spiral models. To the extent that this cannot be maintained as an integral activity, integration testing is negatively affected. System testing is not affected.

Because a build is a set of deliverable end-user functionality, one advantage common to all these spin-off models is that they provide earlier synthesis. This also results in earlier customer feedback, so two of the deficiencies of waterfall development are mitigated. The next section describes two approaches to deal with the "perfect foresight" problem.

11.2.2 Specification-Based Life Cycle Models

When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. Barry Boehm jokes when he describes the customer who says "I don't know what I want, but I'll recognize it when I see it." The rapid prototyping life cycle (Figure 11.4)

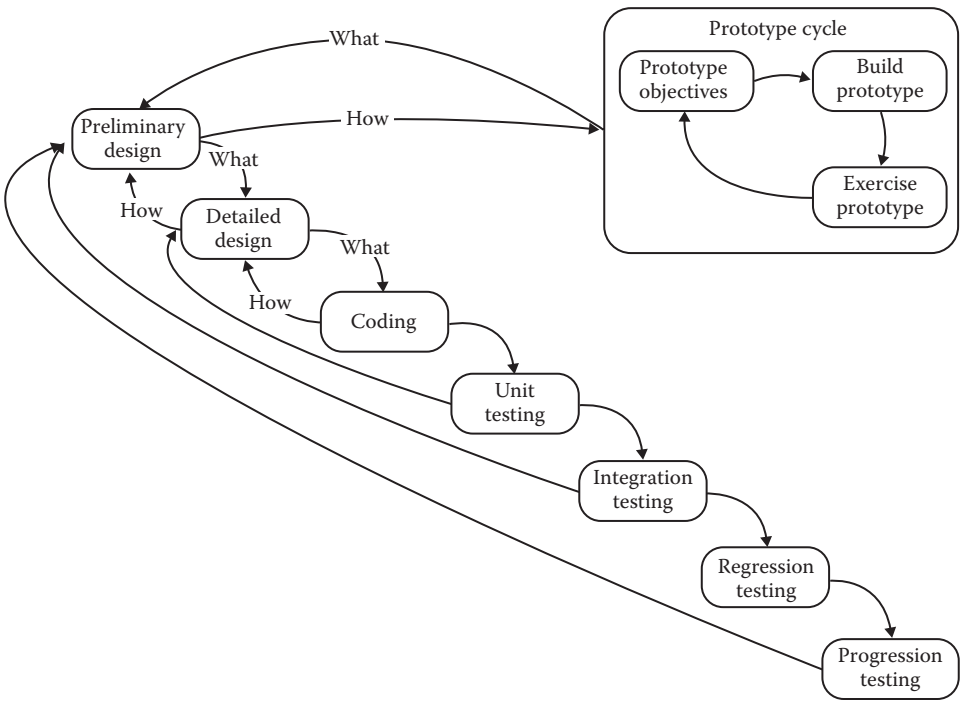

Figure 11.4 Rapid prototyping life cycle.

deals with this by providing the "look and feel" of a system. Thus, in a sense, customers can recognize what they "see." In turn, this drastically reduces the specification-to-customer feedback loop by producing very early synthesis. Rather than build a final system, a "quick and dirty" prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used. The agile life cycles are the extreme of this pattern.

Rapid prototyping has no new implications for integration testing; however, it has very interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycles as information-gathering activities and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototypes, define these as scenarios that are important to the customer, and then use these as system test cases. These could be precursors to the user stories of the agile life cycles. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate because most customers do not care about the structure, and they do care about the behavior.

Executable specifications (Figure 11.5) are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines, StateCharts, or Petri nets). The customer then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model. The executable models are, or can be, quite complex. This is an understatement for the full-blown version of StateCharts. Building an executable model requires expertise, and executing it requires an engine. Executable

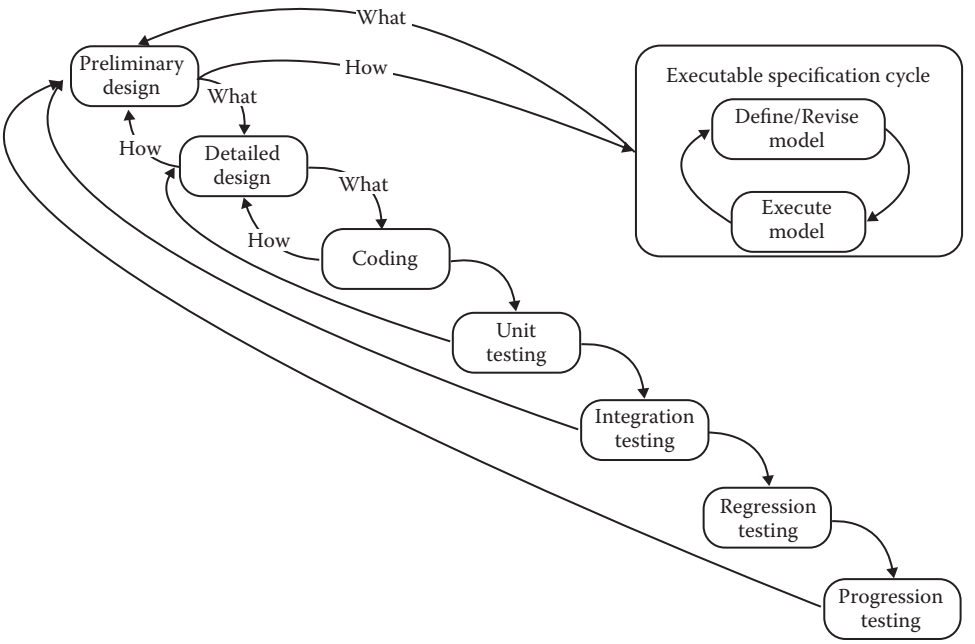

Figure 11.5 Executable specification.

specification is best applied to event-driven systems, particularly when the events can arrive in different orders. David Harel, the creator of StateCharts, refers to such systems as "reactive" (Harel, 1988) because they react to external events. As with rapid prototyping, the purpose of an executable specification is to let the customer experience scenarios of intended behavior. Another similarity is that executable models might have to be revised on the basis of customer feedback. One side benefit is that a good engine for an executable model will support the capture of "interesting" system transactions, and it is often a nearly mechanical process to convert these into true system test cases. If this is done carefully, system testing can be traced directly back to the requirements.

Once again, this life cycle has no implications for integration testing. One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. We will see this in Chapter 14. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Here is another important distinction: when system testing is based on an executable specification, we have an interesting form of structural testing at the system level. Finally, as we saw with rapid prototyping, the executable specification step can be combined with any of the iterative life cycle models.

11.3 Agile Testing

The *Agile Manifesto* (http://agilemanifesto.org/) was written by 17 consultants, the Agile Alliance, in February 2001. It has been translated into 42 languages and has drastically changed the software development world. The underlying characteristics of all agile life cycles are

- Customer-driven
- Bottom–up development
- Flexibility with respect to changing requirements
- Early delivery of fully functional components

These are sketched in Figure 11.6. Customers express their expectations in terms of user stories, which are taken as the requirements for very short iterations of design–code–test. When does an

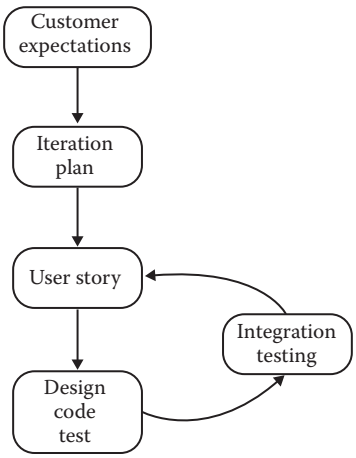

Figure 11.6 Generic agile life cycle.

agile project end? When the customer has no more user stories. Looking back at the iterative models, we see the progenitors of agility, especially in Barry Boehm's spiral model. Various websites will list as few as 3 to as many as 40 variations of agile software development. Here we look at three major ones, and focus on how they deal with testing.

11.3.1 Extreme Programming

Extreme Programming (XP) was first applied to a project (in a documented way) in 1996 by Kent Beck (http://www.extremeprogramming.org/) while he was at Chrysler Corporation. The clear success of the project, even though it was a revision of an earlier version, led to his book (Beck, 2004). The main aspects of XP are captured in Figure 11.7. It is clearly customer-driven, as shown by the position of user stories driving both a release plan and system testing. The release plan defines a sequence of iterations, each of which delivers a small working component. One distinction of XP is the emphasis on paired programming, in which a pair of developers work closely together, often sharing a single development computer and keyboard. One person works at the code level, while the other takes a slightly higher view. In a sense, the pair is conducting a continuous review. In Chapter 22, we will see that this is better described as a continuous code walk-through. There are many similarities to the basic iterative life cycle shown in Figure 11.3. One important difference is that there is no overall preliminary design phase. Why? Because this is a bottom–up process. If XP were truly driven by a sequence of user stories, it is hard to imagine what can occur in the release plan phase.

11.3.2 Test-Driven Development

Test-driven development (TDD) is the extreme case of agility. It is driven by a sequence of user stories, as shown in Figure 11.8. A user story can be decomposed into several tasks, and this is where the big difference occurs. Before any code is written for a task, the developer decides how it will be tested. The tests become the specification. The next step is curious—the tests are run on nonexistent code. Naturally, they fail, but this leads to the best feature of TDD—greatly simplified

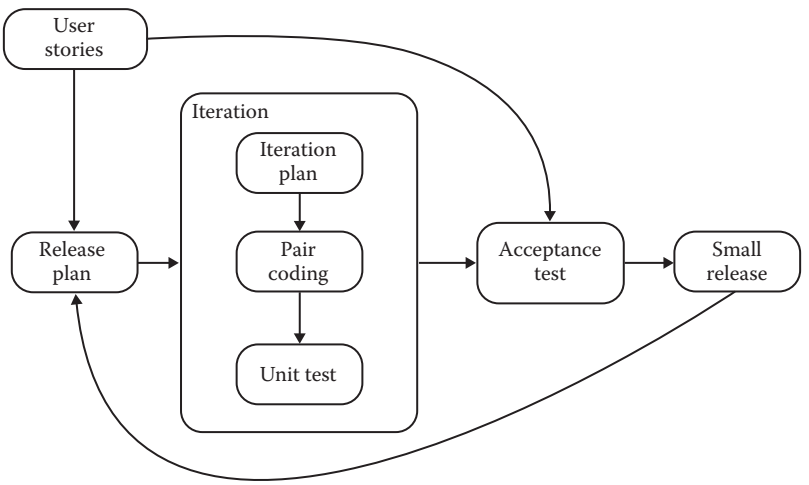

Figure 11.7 The Extreme Programming life cycle.

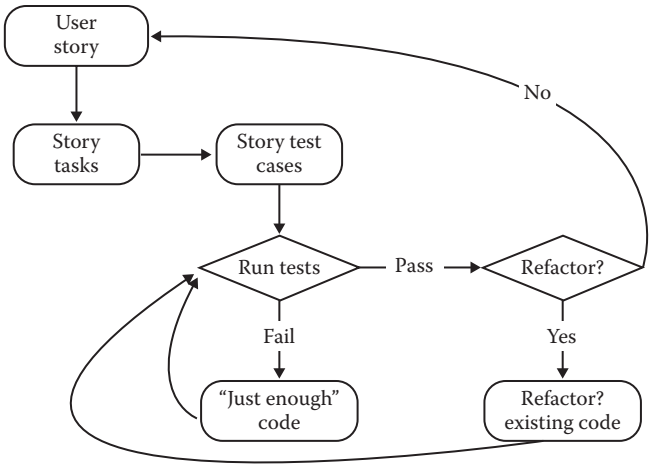

Figure 11.8 Test-driven development life cycle.

fault isolation. Once the tests have been run (and failed), the developer writes just enough code to make the tests pass, and the tests are rerun. If any test fails, the developer goes back to the code and makes a necessary change. Once all the tests pass, the next user story is implemented. Occasionally, the developer may decide to refactor the existing code. The cleaned–up code is then subjected to the full set of existing test cases, which is very close to the idea of regression testing. For TDD to be practical, it must be done in an environment that supports automated testing, typically with a member of the nUnit family of automated test environments. (We will have an example of this in Chapter 19.)

Testing in TDD is interesting. Since the story-level test cases drive the coding, they ARE the specification, so in a sense, TDD uses specification-based testing. But since the code is deliberately as close as possible to the test cases, we could argue that it is also code-based testing. There are two problems with TDD. The first is common to all agile flavors—the bottom–up approach prohibits a single, high-level design step. User stories that arrive late in the sequence may obviate earlier design choices. Then refactoring would have to also occur at the design level, rather than just at the code level. The agile community is very passionate about the claim that repeated refactoring results in an elegant design. Given one of the premises of agile development, namely that the customer is not sure of what is needed, or equivalently, rapidly changing requirements, refactoring at both the code and design levels seems the only way to end up with an elegant design. This is an inevitable constraint on bottom–up development.

The second problem is that all developers make mistakes—that is much of the reason we test in the first place. But consider: what makes us think that the TDD developer is perfect at devising the test cases that drive the development? Even worse: what if late user stories are inconsistent with earlier ones? A final limitation of TDD is there is no place in the life cycle for a cross-check at the user story level.

11.3.3 Scrum

Scrum is probably the most frequently used of all the agile life cycles. There is a pervading emphasis on the team members and teamwork. The name comes from the rugby maneuver in which the

opposing teams are locked together and try to hook the football back to their respective sides. A rugby scrum requires organized teamwork—hence, the name for the software process.

The quick view of Scrum (the development life cycle) is that it is mostly new names for old ideas. This is particularly true about the accepted Scrum vocabulary. Three examples: roles, ceremonies, and artifacts. In common parlance, Scrum roles refer to project participants; the ceremonies are just meetings, the artifacts are work products. Scrum projects have Scrum masters (who act like traditional supervisors with less administrative power). Product owners are the customers of old, and the Scrum team is a development team. Figure 11.9 is adapted from the "official" Scrum literature, the Scrum Alliance (http://www.scrumalliance.org/learn_about_scrum). Think about the activities in terms of the iterative life cycle in Figure 11.3. The traditional iterations become "sprints," which last from 2 to 4 weeks. In a sprint, there is a daily stand-up meeting of the Scrum team to focus on what happened the preceding day and what needs to be done in the new day. Then there is a short burst of design–code–test followed by an integration of the team's work at the end of the day. This is the agile part—a daily build that contributes to a sprint-level work product in a short interval. The biggest differences between Scrum and the traditional view of iterative development are the special vocabulary and the duration of the iterations.

Testing in the Scrum life cycle occurs at two levels—the unit level at each day's end, and the integration level of the small release at the end of a sprint. Selection of the Sprint backlog from the product backlog is done by the product owner (the customer), which corresponds roughly to a requirements step. Sprint definition looks a lot like preliminary design because this is the point where the Scrum team identifies the sequence and contents of individual sprints. The bottom line? Scrum has two distinct levels of testing—unit and integration/system. Why "integration/system?" The small release is a deliverable product usable by the product owner, so it is clearly a system-level work product. But this is the point where all of the development work is integrated for the first time.

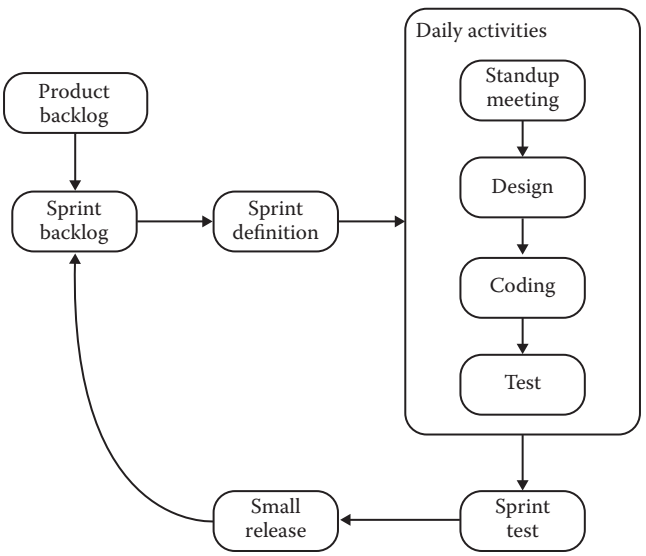

Figure 11.9 The Scrum life cycle.

11.4 Agile Model–Driven Development

My German friend Georg is a PhD mathematician, a software developer, and a Go player. For several months, we had an e-mail-based discussion about agile development. At one point, Georg asked if I play the oriental game Go. I do not, but he replied that, to be a successful Go player, one needs both strategy and tactics. A deficiency in either one puts a Go player at a disadvantage. In the software development realm, he equates strategy with an overall design, and tactics as unit-level development. His take on the flavors of agile development is that the strategy part is missing, and this leads us to a compromise between the agile world and the traditional views of software development. We first look at Agile Model–Driven Development (AMDD) popularized by Scott Ambler. This is followed by my mild reorganization of Ambler's work, named here as Model–Driven Agile Development (MDAD).

11.4.1 Agile Model–Driven Development

The agile part of AMDD is the modeling step. Ambler's advice is to model just enough for the current user story, and then implement it with TDD. The big difference between AMDD and any of the agile life cycles is that there is a distinct design step. (The agilists usually express their distaste/disdain for modeling by calling it the "Big Design Up Front" and abbreviate it as simply the BDUF.) See Figure 11.10.

Ambler's contribution is the recognition that design does indeed have a place in agile development. As this was being written, there was a protracted discussion on LinkedIn started by the question "Is there any room for design in agile software development?" Most of the thread affirms the need for design in any agile life cycle. Despite all this, there seems to be no room in AMDD for integration/system testing.

11.4.2 Model–Driven Agile Development

Model–driven agile development (MDAD) is my proposal for a compromise between the traditional and the agile worlds. It is stimulated by Georg's view of the need for both strategy and tactics, hence the compromise. How does MDAD differ from iterative development? MDAD recommends test-driven development as the tactic and it uses Ambler's view of short iterations.

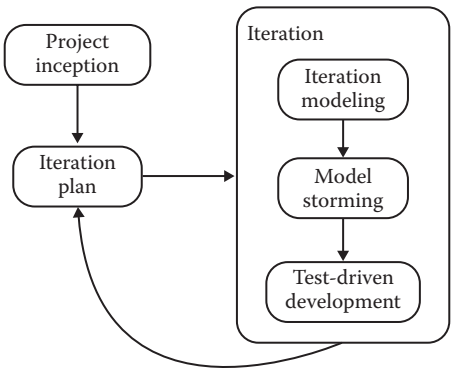

Figure 11.10 The agile model–driven development life cycle.

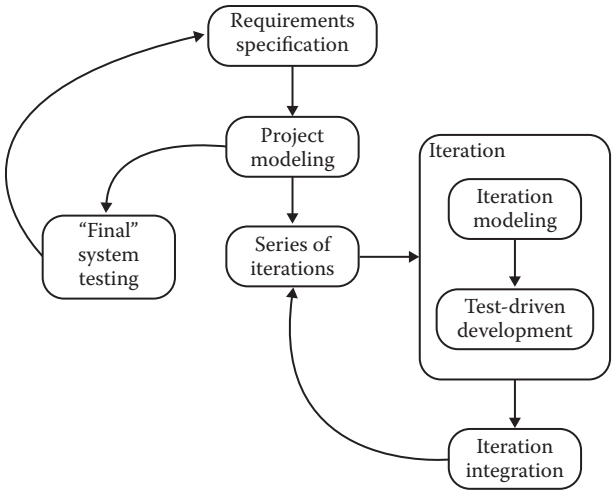

Figure 11.11 The model–driven agile development life cycle.

The strategy part is the emphasis on an overall model, which in turn, supports MBT. In MDAD, the three levels of testing, unit, integration, and system, are present (Figure 11.11).

References

Agresti, W.W., *New Paradigms for Software Development,* IEEE Computer Society Press, Washington, DC, 1986.

Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison Wesley, Boston, 2004.

Boehm, B.W., A spiral model for software development and enhancement, *IEEE Computer*, Vol. 21, No. 6, May 1988, pp. 61–72.

Harel, D., On visual formalisms, *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 514–530.

Chapter 13

Integration Testing

In September 1999, the Mars Climate Orbiter mission failed after successfully traveling 416 million miles in 41 weeks. It disappeared just as it was to begin orbiting Mars. The fault should have been revealed by integration testing: Lockheed Martin Astronautics used acceleration data in English units (pounds), while the Jet Propulsion Laboratory did its calculations with metric units (newtons). NASA announced a $50,000 project to discover how this could have happened (Fordahl, 1999). They should have read this chapter.

Of the three distinct levels of software testing—unit, integration, and system—integration testing is the least well understood of these; hence in practice, it is the phase most poorly done. This chapter examines two mainline and one less well-known integration testing strategies. They are illustrated with a continuing procedural example, discussed in some detail, and then critiqued with respect to their advantages and disadvantages.

Craftspersons are recognized by two essential characteristics: they have a deep knowledge of the tools of their trade, and they have a similar knowledge of the medium in which they work so that they understand their tools in terms of how they work with the medium. In Chapters 5 through 10, we focused on the tools (techniques) available to the testing craftsperson at the unit level. Our goal there was to understand testing techniques in terms of their advantages and limitations with respect to particular types of software. Here, we continue our emphasis on model-based testing, with the goal of improving the testing craftsperson's judgment through a better understanding of three underlying models. Integration testing for object-oriented software is integrated into this chapter.

13.1 Decomposition-Based Integration

Mainline introductory software engineering texts, for example, Pressman (2005) and Schach (2002), typically present four integration strategies based on the functional decomposition tree of the procedural software: top–down, bottom–up, sandwich, and the vividly named "big bang." Many classic software testing texts echo this approach, Deutsch (1982), Hetzel (1988), Kaner et al. (1993), and Mosley (1993), to name a few. Each of these strategies (except big bang) describes the order in which units are to be integrated. We can dispense with the big bang

approach most easily: in this view of integration, all the units are compiled together and tested at once. The drawback to this is that when (not if!) a failure is observed, few clues are available to help isolate the location(s) of the fault. (Recall the distinction we made in Chapter 1 between faults and failures.)

The functional decomposition tree is the basis for integration testing because it is the main representation, usually derived from final source code, which shows the structural relationship of the system with respect to its units. All these integration orders presume that the units have been separately tested; thus, the goal of decomposition-based integration is to test the interfaces among separately tested units. A functional decomposition tree reflects the lexicological inclusion of units, in terms of the order in which they need to be compiled, to assure the correct referential scope of variables and unit names. In this chapter, our familiar NextDate unit is extended to a main program, Calendar, with procedures and functions. Figure 13.1 contains the functional decomposition tree for the Calendar program. The pseudocode is given in next.

The Calendar program sketched here in pseudocode acquires a date in the form mm, dd, yyyy, and provides the following functional capabilities:

- The date of the next day (our old friend, NextDate)
- The day of the week corresponding to the date (i.e., Monday, Tuesday, …)
- The zodiac sign of the date
- The most recent year in which Memorial Day was celebrated on May 27
- The most recent Friday the 13th

A sketch of the Calendar program is given next, followed by a condensed "skeleton," which is the basis for the functional decomposition in Figure 13.1.

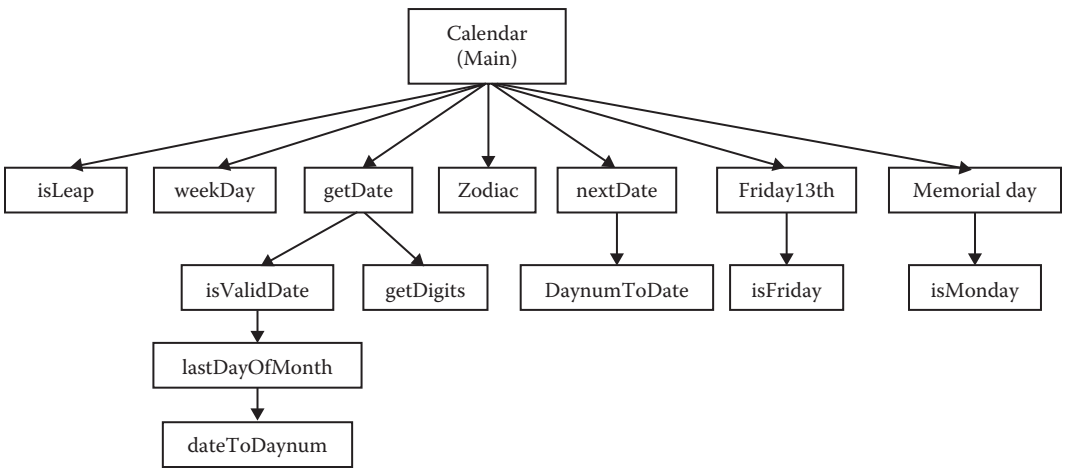

Figure 13.1 Functional decomposition of Calendar program.

Pseudocode for the Calendar Program

```
Main   Calendar
Data Declarations
    mm, dd, yyyy, dayNumber, dayName, zodiacSign
Function isLeap (input yyyy, returns T/F)
    (isLeap is self-contained)
End Function isLeap

Procedure getDate (returns mm, dd, yyyy, dayNumber)
    Function isValidDate (inputs mm, dd, yyyy; returns T/F)
       Function lastDayOfMonth (inputs mm, yyyy, returns 28, 29, 30, or 31)
          lastDayOfMonth body
               (uses isLeap)
          end lastDayOfMonth body
       End Function lastDayOfMonth

       isValidDate body
          (uses lastDayOfMonth)
       end isValidDate body
    End Function isValidDate

    Procedure getDigits(returns mm, dd, yyyy)
       (uses Function isValidDate)
    End Procedure getDigits

    Procedure memorialDay (inputs mm, dd, yyyy; returns yyyy)
       Function isMonday (inputs mm, dd, yyyy; returns T/F)
          (uses weekDay)
       End Function isMonday

       memorialDaybody
          isMonday
       end memorialDay
    End Procedure memorialDay

Procedure friday13th (inputs mm, dd, yyyy; returns mm1, dd1, yyyy1)
       Function isFriday (inputs mm, dd, yyyy; returns T/F)
          (uses weekDay)
       End Function isFriday

    friday13th body
       (uses isFriday)
    end friday13th
End Procedure friday13th

getDate body
    getDigits
    isValidDate
    dateToDayNumber
end getDate body
End Procedure getDate
Procedure nextDate (input daynum, output mm1, dd1, yyyy1)
    Procedure dayNumToDate
```

```
    dayNumToDate body
       (uses isLeap)
    end dayNumToDate body
nextDate body
    dayNumToDate
end nextDate body
End Procedure nextDate

Procedure weekDay (input mm, dd, yyyy; output dayName)
    (uses Zeller's Congruence)
End Procedure weekDay

Procedure zodiac (input dayNumber; output dayName)
    (uses dayNumbers of zodiac cusp dates)
End Procedure zodiac

Main program body
    getDate
    nextDate
    weekDay
    zodiac
    memorialDay
    friday13th
End Main program body
```

Lexicological Inclusion of the Calendar Program

```
Main   Calendar
    Function isLeap
    Procedure weekDay
    Procedure getDate
       Function isValidDate
          Function lastDayOfMonth
       Procedure getDigits
    Procedure memorialDay
          Function isMonday
    Procedure friday13th
       Function isFriday
    Procedure nextDate
       Procedure dayNumToDate
    Procedure zodiac
```

13.1.1 Top–Down Integration

Top–down integration begins with the main program (the root of the tree). Any lower-level unit that is called by the main program appears as a "stub," where stubs are pieces of throwaway code that emulate a called unit. If we performed top–down integration testing for the Calendar program, the first step would be to develop stubs for all the units called by the main program—isLeap, weekDay, getDate, zodiac, nextDate, friday13th, and memorialDay. In a stub for any unit, the tester hard codes in a correct response to the request from the calling/invoking unit. In the stub for zodiac, for example, if the main program calls zodiac with 05, 27, 2012, zodiacStub would return "Gemini." In extreme practice, the response might be "pretend zodiac returned Gemini."

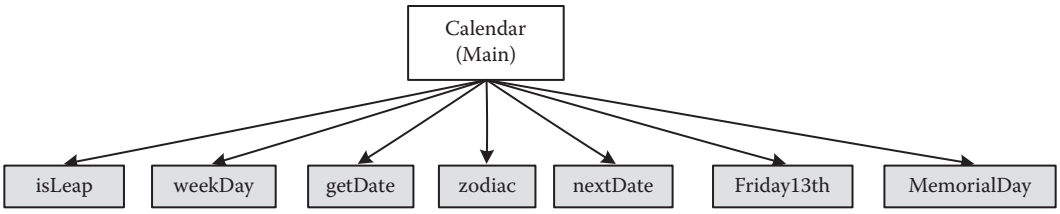

Figure 13.2 First step in top–down integration.

The use of the pretend prefix emphasizes that it is not a real response. In practice, the effort to develop stubs is usually quite significant. There is good reason to consider stub code as part of the software project and maintain it under configuration management. In Figure 13.2, the first step in top–down integration is shown. The gray-shaded units are all stubs. The goal of the first step is to check that the main program functionality is correct.

Once the main program has been tested, we replace one stub at a time, leaving the others as stubs. Figure 13.3 shows the first three steps in the gradual replacement of stubs by actual code. The stub replacement process proceeds in a breadth-first traversal of the decomposition tree until all the stubs have been replaced. (In Figures 13.2 and 13.3, the units below the first level are not shown because they are not needed.)

The "theory" of top–down integration is that, as stubs are replaced one at a time, if there is a problem, it must be with the interface to the most recently replaced stub. (Note that the fault

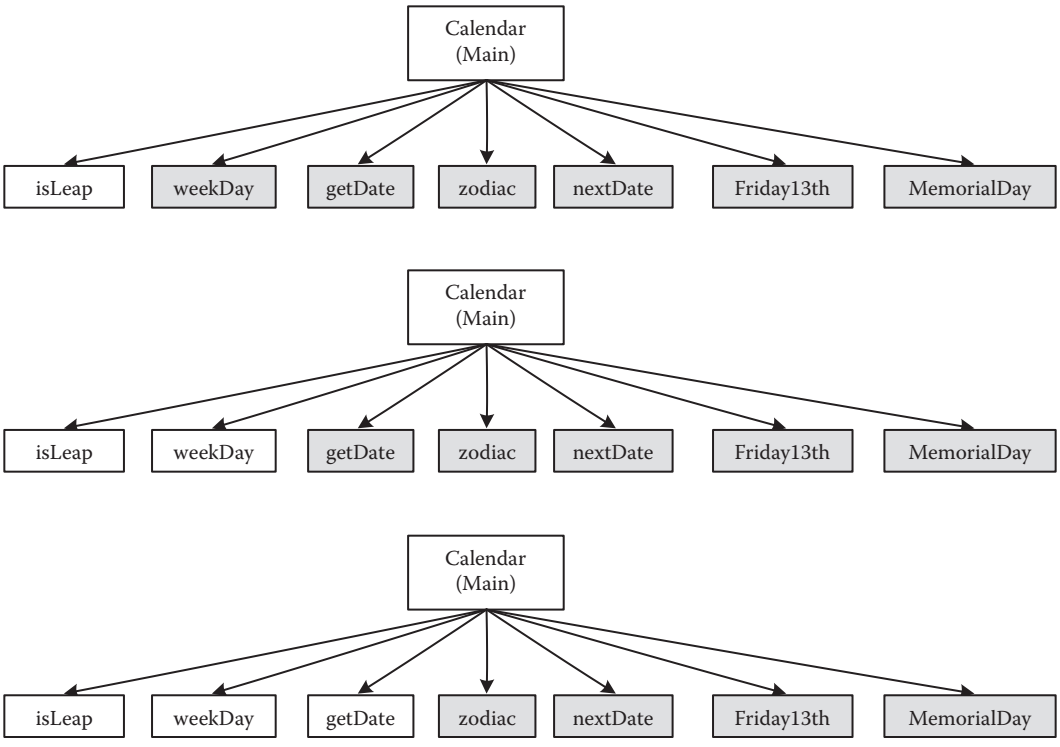

Figure 13.3 Next three steps in top–down integration.

isolation is similar to that of test-driven development.) The problem is that a functional decomposition is deceptive. Because it is derived from the lexicological inclusion required by most compilers, the process generates impossible interfaces. Calendar main never directly refers to either isLeap or weekDay, so those test sessions could not occur.

13.1.2 Bottom–Up Integration

Bottom–up integration is a "mirror image" to the top–down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. (In Figure 13.4, the gray units are drivers.) Bottom–up integration begins with the leaves of the decomposition tree, and use a driver version of the unit that would normally call it to provide it with test cases. (Note the similarity to test driver units at the unit level. As units are tested, the drivers are gradually replaced, until the full decomposition tree has been traversed. Less throwaway code exists in bottom–up integration, but the problem of impossible interfaces persists.

Figure 13.5 shows one case where a unit (zodiac) can be tested with a driver. In this case, the Calendar driver would probably call zodiac with 36 test dates that are the day before a cusp date, the cusp date, and the day after the cusp date. The cusp date for Gemini is May 21, so the driver would call zodiac three times, with May 20, May 21, and May 22. The expected responses would be "Taurus," "Gemini," and "Gemini," respectively. Note how similar this is to the assert mechanism in the jUnit (and related) test environments.

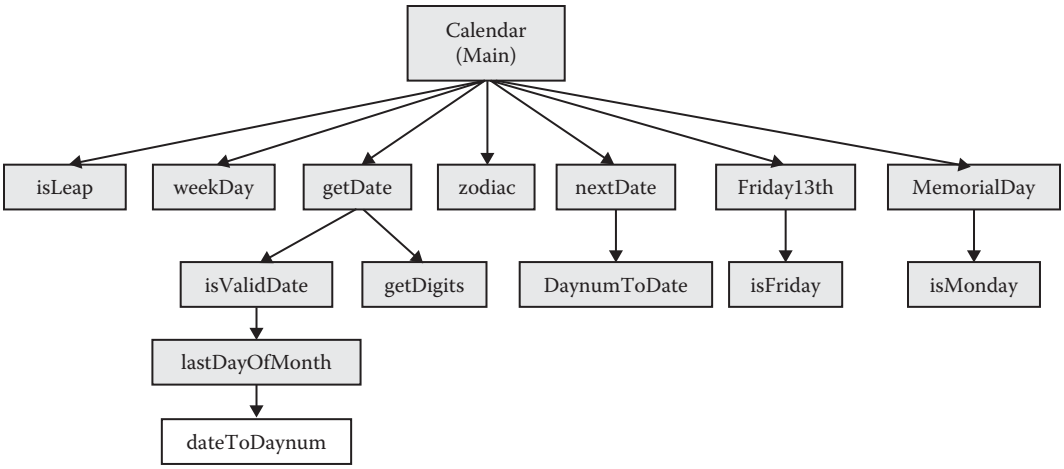

Figure 13.4 First steps in bottom–up integration.

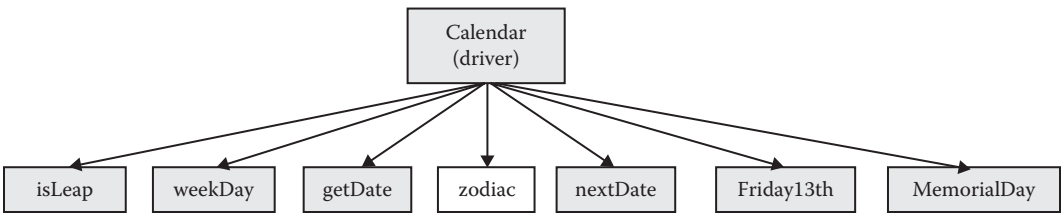

Figure 13.5 Bottom–up integration for zodiac.

13.1.3 Sandwich Integration

Sandwich integration is a combination of top–down and bottom–up integration. If we think about it in terms of the decomposition tree, we are really only doing big bang integration on a subtree (see Figure 13.6). There will be less stub and driver development effort, but this will be offset to some extent by the added difficulty of fault isolation that is a consequence of big bang integration. (We could probably discuss the size of a sandwich, from dainty finger sandwiches to Dagwood-style sandwiches, but not now.)

A sandwich is a full path from the root to leaves of the functional decomposition tree. In Figure 13.6, the set of units is almost semantically coherent, except that isLeap is missing. This set of units could be meaningfully integrated, but test cases at the end of February would not be covered. Also note that the fault isolation capability of the top–down and bottom–up approaches is sacrificed. No stubs nor drivers are needed in sandwich integration.

13.1.4 Pros and Cons

With the exception of big bang integration, the decomposition-based approaches are all intuitively clear. Build with tested components. Whenever a failure is observed, the most recently added unit is suspected. Integration testing progress is easily tracked against the decomposition tree. (If the tree is small, it is a nice touch to shade in nodes as they are successfully integrated.) The top–down and bottom–up terms suggest breadth-first traversals of the decomposition tree, but this is not mandatory. (We could use full-height sandwiches to test the tree in a depth-first manner.)

One of the most frequent objections to functional decomposition and waterfall development is that both are artificial, and both serve the needs of project management more than the needs of software developers. This holds true also for decomposition-based testing. The whole mechanism is that units are integrated with respect to structure; this presumes that correct behavior follows from individually correct units and correct interfaces. (Practitioners know better.) The development effort for stubs or drivers is another drawback to these approaches, and this is compounded by the retesting effort.

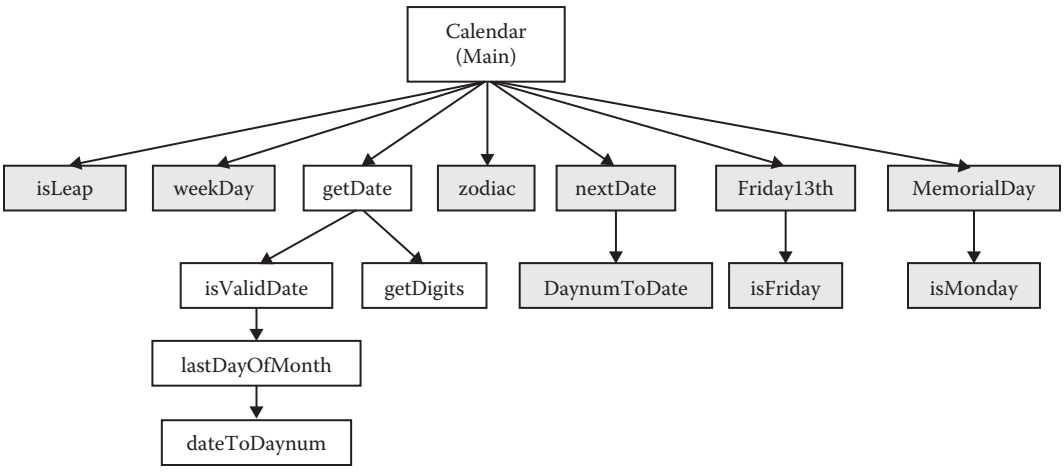

Figure 13.6 Sample sandwich integration.

13.2 Call Graph–Based Integration

One of the drawbacks of decomposition-based integration is that the basis is the functional decomposition tree. We saw that this leads to impossible test pairs. If we use the call graph instead, we resolve this deficiency; we also move in the direction of structural testing. The call graph is developed by considering units to be nodes, and if unit A calls (or uses) unit B, there is an edge from node A to node B. The call graph for the Calendar program is shown in Figure 13.7.

Since edges in the call graph refer to actual execution–time connections, the call graph avoids all the problems we saw in the decomposition tree–based versions of integration. In fact, we could repeat the discussion of Section 13.1 based on stubs and drivers in the units in Figure 13.7. This will work well, and it preserves the fault isolation feature of the decomposition-based approaches. Figure 13.8 shows the first step in call graph–based top–down integration.

The stubs in the first session could operate as follows. When the Calendar main program calls getDateStub, the stub might return May 27, 2013. The zodiacStub would return "Gemini," and so on. Once the main program logic is tested, the stubs would be replaced, as we discussed in Section 13.1. The three strategies of Section 13.1 will all work well when stubs and drivers are based on the call graph rather than the functional decomposition.

We are in a position to enjoy the investment we made in the discussion of graph theory. Because the call graph is a directed graph, why not use it the way we used program graphs? This leads us to two new approaches to integration testing: we will refer to them as pairwise integration and neighborhood integration.

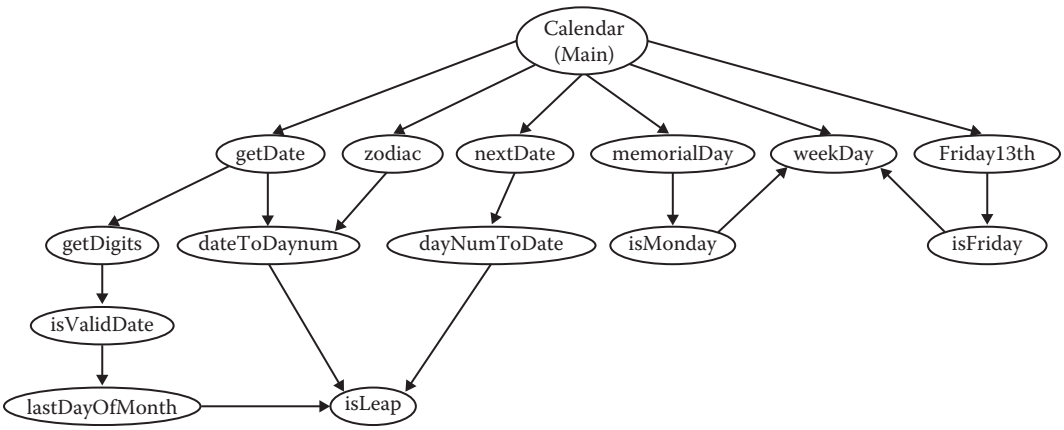

Figure 13.7 Call graph of Calendar program.

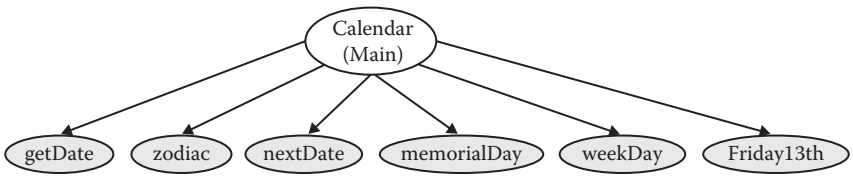

Figure 13.8 Call graph–based top–down integration of Calendar program.

13.2.1 Pairwise Integration

The idea behind pairwise integration is to eliminate the stub/driver development effort. Instead of developing stubs and/or drivers, why not use the actual code? At first, this sounds like big bang integration, but we restrict a session to only a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph. Pairwise integration results in an increased number of integration sessions when a node (unit) is used by two or more other units. In the Calendar example, there would be 15 separate sessions for top–down integration (one for each stub replacement); this increases to 19 sessions for pairwise integration (one for each edge in the call graph). This is offset by a reduction in stub/driver development. Three pairwise integration sessions are shown in Figure 13.9.

The main advantage of pairwise integration is the high degree of fault isolation. If a test fails, the fault must be in one of the two units. The biggest drawback is that, for units involved on several pairs, a fix that works in one pair may not work in another pair. This is yet another example of the testing pendulum discussed in Chapter 10. Call graph integration is slightly better than the decomposition tree–based approach, but both can be removed from the reality of the code being tested.

13.2.2 Neighborhood Integration

We can let the mathematics carry us still further by borrowing the notion of a neighborhood from topology. (This is not too much of a stretch—graph theory is a branch of topology.) The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node. (Technically, this is a neighborhood of radius 1; in larger systems, it makes sense to increase the neighborhood radius.) In a directed graph, this includes all the immediate predecessor nodes and all the immediate successor nodes (notice that these correspond to the set of stubs and drivers of the node). The neighborhoods of getDate, nextDate, Friday13th, and weekDay are shown in Figure 13.10.

The 15 neighborhoods for the Calendar example (based on the call graph in Figure 13.7) are listed in Table 13.1. To make the table simpler, the original unit names are replaced by node numbers (in Figure 13.11), where the numbering is generally breadth first. The juxtaposition and connectivity is preserved in Figure 13.11.

The information in Table 13.1 is given in Table 13.2 as the adjacency matrix for the call graph. The column sums show the indegrees of each node, and the row sums show the outdegrees.

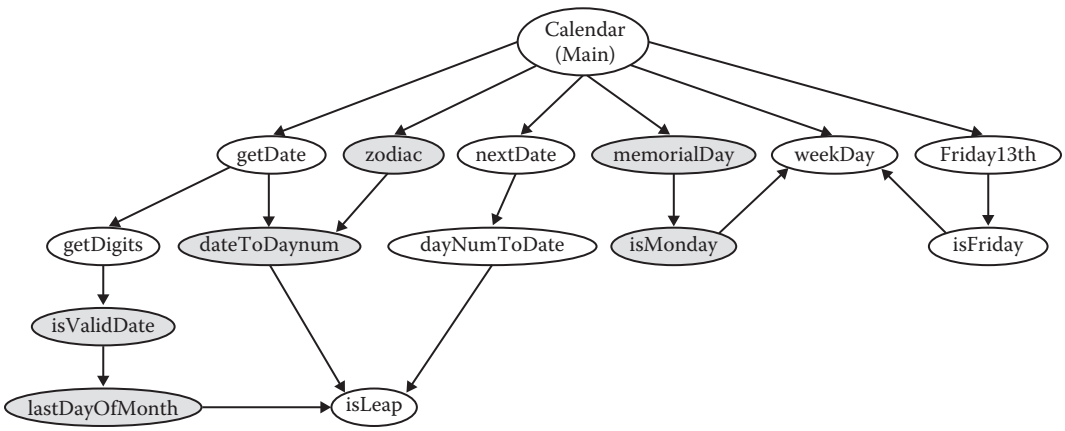

Figure 13.9 Three pairs for pairwise integration.

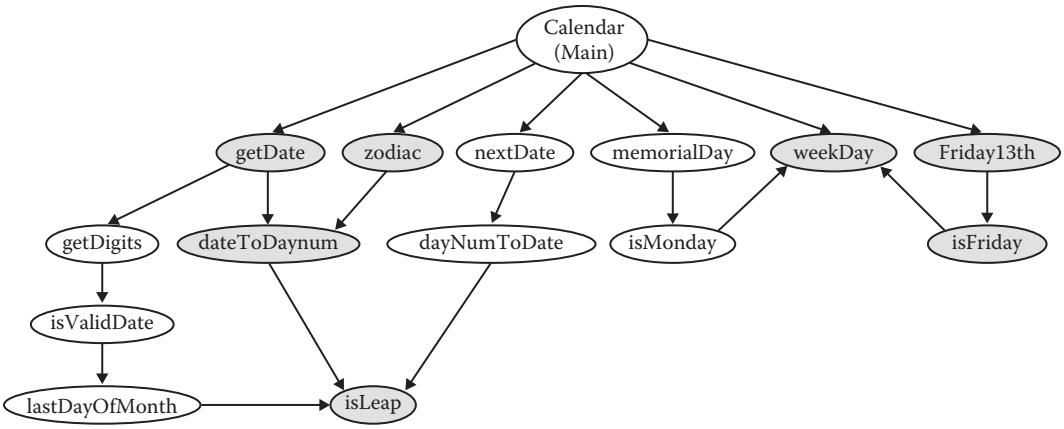

Figure 13.10 Three neighborhoods (of radius 1) for neighborhood integration.

Table 13.1 Neighborhoods of Radius 1 in Calendar Call Graph

Neighborhoods in Calendar Program Call Graph			
Node	*Unit Name*	*Predecessors*	*Successors*
1	Calendar (Main)	(None)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9
4	nextDate	1	10
5	memorialDay	1	11
6	weekday	1, 11, 12	(None)
7	Friday13th	1	12
8	getDigits	2	13
9	dateToDayNum	3	15
10	dayNumToDate	4	15
11	isMonday	5	6
12	isFriday	7	6
13	isValidDate	8	14
14	lastDayOfMonth	13	15
15	isLeap	9, 10, 14	(None)

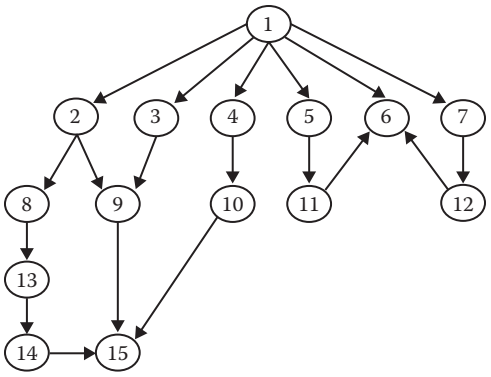

Figure 13.11 Calendar call graph with units replaced by numbers.

Table 13.2 Adjacency Matrix of Calendar Call Graph

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*Row Sum*
1		1	1	1	1	1	1									6
2								1	1							2
3									1							1
4										1						1
5											1					1
6																0
7												1				1
8													1			1
9															1	1
10															1	1
11							1									1
12							1									1
13														1		1
14															1	1
15																0
Column sum	0	1	1	1	1	1	3	1	2	1	1	1	1	1	3	

We can always compute the number of neighborhoods for a given call graph. Each interior node will have one neighborhood, plus one extra in case leaf nodes are connected directly to the root node. (An interior node has a nonzero indegree and a nonzero outdegree.) We have

$$\text{Interior nodes} = \text{nodes} - (\text{source nodes} + \text{sink nodes})$$

$$\text{Neighborhoods} = \text{interior nodes} + \text{source nodes}$$

which combine to

$$\text{Neighborhoods} = \text{nodes} - \text{sink nodes}$$

Neighborhood integration usually yields a reduction in the number of integration test sessions, and it reduces stub and driver development. The end result is that neighborhoods are essentially the sandwiches that we slipped past in the previous section. (It is slightly different because the base information for neighborhoods is the call graph, not the decomposition tree.) What they share with sandwich integration is more significant—neighborhood integration testing has the fault isolation difficulties of "medium bang" integration.

13.2.3 Pros and Cons

The call graph–based integration techniques move away from a purely structural basis toward a behavioral basis; thus, the underlying assumption is an improvement. (See the Testing Pendulum in Chapter 10.) The neighborhood-based techniques also reduce the stub/driver development effort. In addition to these advantages, call graph–based integration matches well with developments characterized by builds and composition. For example, sequences of neighborhoods can be used to define builds. Alternatively, we could allow adjacent neighborhoods to merge (into villages?) and provide an orderly, composition-based growth path. All this supports the use of neighborhood-based integration for systems developed by life cycles in which composition dominates.

The biggest drawback to call graph–based integration testing is the fault isolation problem, especially for large neighborhoods. A more subtle but closely related problem occurs. What happens if (when) a fault is found in a node (unit) that appears in several neighborhoods? The adjacency matrix highlights this immediately—nodes with either a high row sum or a high column sum will be in several neighborhoods. Obviously, we resolve the fault in one neighborhood; but this means changing the unit's code in some way, which in turn means that all the previously tested neighborhoods that contain the changed node need to be retested.

Finally, a fundamental uncertainty exists in any structural form of testing: the presumption that units integrated with respect to structural information will exhibit correct behavior. We know where we are going: we want system-level threads of behavior to be correct. When integration testing based on call graph information is complete, we still have quite a leap to get to system-level threads. We resolve this by changing the basis from call graph information to special forms of paths.

13.3 Path-Based Integration

Much of the progress in the development of mathematics comes from an elegant pattern: have a clear idea of where you want to go, and then define the concepts that take you there. We do this here for path-based integration testing, but first we need to motivate the definitions.

We already know that the combination of structural and functional testing is highly desirable at the unit level; it would be nice to have a similar capability for integration (and system) testing. We also know that we want to express system testing in terms of behavioral threads. Lastly, we revise our goal for integration testing: instead of testing interfaces among separately developed and tested units, we focus on interactions among these units. ("Co-functioning" might be a good term.) Interfaces are structural; interaction is behavioral.

When a unit executes, some path of source statements is traversed. Suppose that a call goes to another unit along such a path. At that point, control is passed from the calling unit to the called unit, where some other path of source statements is traversed. We deliberately ignored this situation in Chapter 8, because this is a better place to address the question. Two possibilities are available: abandon the single-entry, single-exit precept and treat such calls as an exit followed by an entry, or suppress the call statement because control eventually returns to the calling unit anyway. The suppression choice works well for unit testing, but it is antithetical to integration testing.

13.3.1 New and Extended Concepts

To get where we need to go, we need to refine some of the program graph concepts. As before, these refer to programs written in an imperative language. We allow statement fragments to be a complete statement, and statement fragments are nodes in the program graph.

Definition

A *source node* in a program is a statement fragment at which program execution begins or resumes.

The first executable statement in a unit is clearly a source node. Source nodes also occur immediately after nodes that transfer control to other units.

Definition

A *sink node* in a unit is a statement fragment at which program execution terminates.

The final executable statement in a program is clearly a sink node; so are statements that transfer control to other units.

Definition

A *module execution path* is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.

The effect of the definitions thus far is that program graphs now have multiple source and sink nodes. This would greatly increase the complexity of unit testing, but integration testing presumes unit testing is complete.

Definition

A *message* is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.

Depending on the programming language, messages can be interpreted as subroutine invocations, procedure calls, function references, and the usual messages in an object-oriented programming language. We follow the convention that the unit that receives a message (the message destination) always eventually returns control to the message source. Messages can pass data to other units. We can finally make the definitions for path-based integration testing. Our goal is to have an integration testing analog of DD-paths.

Definition

An *MM-path* is an interleaved sequence of module execution paths and messages.

The basic idea of an MM-path (Jorgensen 1985; Jorgensen and Erickson 1994) is that we can now describe sequences of module execution paths that include transfers of control among separate units. In traditional software, "MM" is nicely understood as module–message; in object-oriented software, it is clearer to interpret "MM" as method–message. These transfers are by messages, therefore, MM-paths always represent feasible execution paths, and these paths cross unit boundaries. The hypothetical example in Figure 13.12 shows an MM-path (the solid edges) in which module A calls module B, which in turn calls module C. Notice that, for traditional (procedural) software, MM-paths will always begin (and end) in the main program.

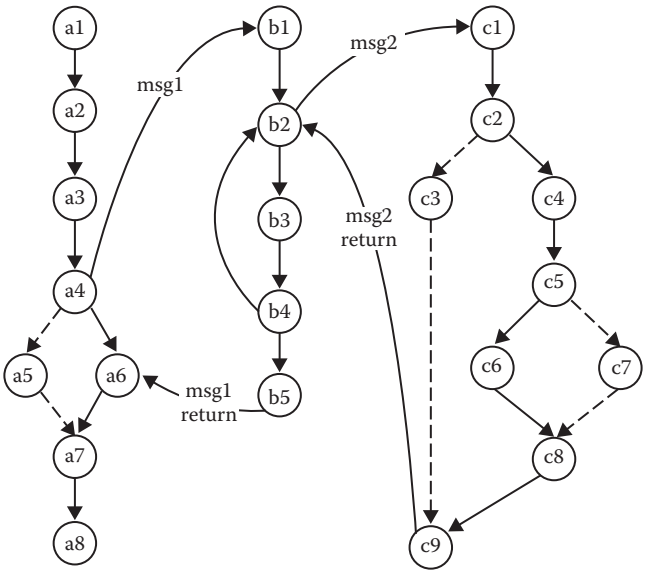

Figure 13.12 Hypothetical MM-path across three units.

In unit A, nodes a1 and a6 are source nodes (a5 and a6 are outcomes of the decision at node a5), and nodes a4 (a decision) and a8 are sink nodes. Similarly, in unit B, nodes b1 and b3 are source nodes, and nodes b2 and b5 are sink nodes. Node b2 is a sink node because control leaves unit B at that point. It could also be a source node, because unit C returns a value used at node b2. Unit C has a single source node, c1, and a single sink node, c9. Unit A contains three module execution paths: <a1, a2, a3, a4>, <a4, a5, a7, a8>, and <a4, a6, a7, a8>. The solid edges are edges actually traversed in this hypothetical example. The dashed edges are in the program graphs of the units as stand-alone units, but they did not "execute" in the hypothetical MM-path. We can now define an integration testing analog of the DD-path graph that serves unit testing so effectively.

Definition

Given a set of units, their *MM-path graph* is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another.

Notice that MM-path graphs are defined with respect to a set of units. This directly supports composition of units and composition-based integration testing. We can even compose down to the level of individual module execution paths, but that is probably more detailed than necessary.

We should consider the relationships among module execution paths, program paths, DD-paths, and MM-paths. A program path is a sequence of DD-paths, and an MM-path is a sequence of module execution paths. Unfortunately, there is no simple relationship between DD-paths and module execution paths. Either might be contained in the other, but more likely, they partially overlap. Because MM-paths implement a function that transcends unit boundaries, we do have one relationship: consider the intersection of an MM-path with a unit. The module execution paths in such an intersection are an analog of a slice with respect to the (MM-path) function. Stated another way, the module execution paths in such an intersection are the restriction of the function to the unit in which they occur.

The MM-path definition needs some practical guidelines. How long ("deep" might be better) is an MM-path? The notion of message quiescence helps here. Message quiescence occurs when a unit that sends no messages is reached (like module C in Figure 13.12). In a sense, this could be taken as a "midpoint" of an MM-path—the remaining execution consists of message returns. This is only mildly helpful. What if there are two points of message quiescence? Maybe a better answer is to take the longer of the two, or, if they are of equal depth, the latter of the two. Points of message quiescence are natural endpoints for an MM-path.

13.3.2 MM-Path Complexity

If you compare the MM-paths in Figures 13.12 and 13.17, it seems intuitively clear that the latter is more complex than the former. Because these are strongly connected directed graphs, we can "blindly" compute their cyclomatic complexities; recall the formula is $V(G) = e - n + 2p$, where p is the number of strongly connected regions. Since messages return to the sending unit, we will always have $p = 1$, so the formula reduces to $V(G) = e - n + 2$. Surprisingly, both graphs have $V(G) = 7$. Clearly, MM-path complexity needs some notion of size in addition to cyclomatic complexity (Figure 13.13).

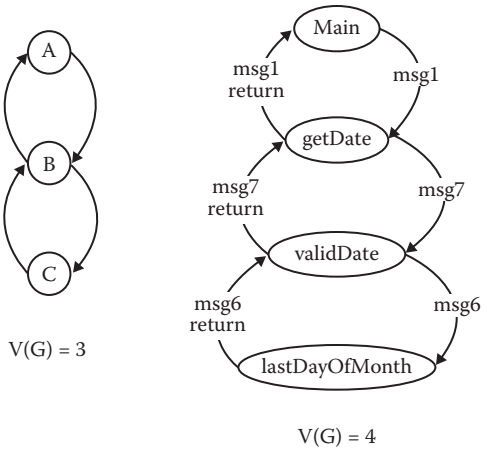

Figure 13.13 Cyclomatic complexities of two MM-paths.

13.3.3 Pros and Cons

MM-paths are a hybrid of functional and structural testing. They are functional in the sense that they represent actions with inputs and outputs. As such, all the functional testing techniques are potentially applicable. The net result is that the cross-check of the functional and structural approaches is consolidated into the constructs for path-based integration testing. We therefore avoid the pitfall of structural testing, and, at the same time, integration testing gains a fairly seamless junction with system testing. Path-based integration testing works equally well for software developed in the traditional waterfall process or with one of the composition-based alternative life cycle models. Finally, the MM-path concept applies directly to object-oriented software.

The most important advantage of path-based integration testing is that it is closely coupled with actual system behavior, instead of the structural motivations of decomposition and call graph–based integration. However, the advantages of path-based integration come at a price—more effort is needed to identify the MM-paths. This effort is probably offset by the elimination of stub and driver development.

13.4 Example: integrationNextDate

Our now familiar NextDate is rewritten here as a main program with a functional decomposition into procedures and functions. This integrationNextDate is a slight extension: there is added validity checking for months, days, and years, so the pseudocode, which follows Figures 13.14 and 13.15, grows from 50 statements to 81. Figures 13.14 and 13.15 show the functional decomposition and the call graph, respectively. Figure 13.16 shows the program graphs of the units in integrationNextDate. Figure 13.17 shows the MM-path for the input date May 27, 2012.

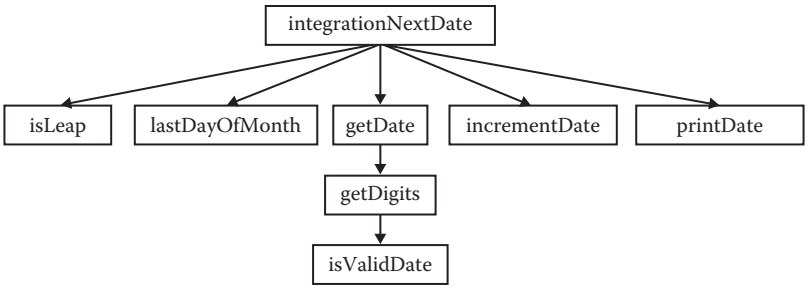

Figure 13.14 Functional decomposition of integrationNextDate.

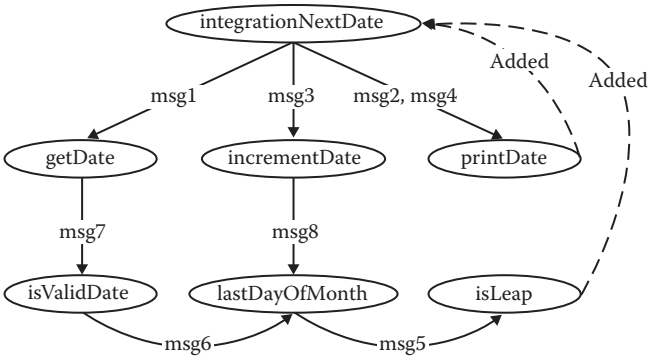

Figure 13.15 Call graph of integrationNextDate.

13.4.1 Decomposition-Based Integration

The isLeap and lastDayOfMonth functions are in the first level of decomposition because they must be available to both GetDate and IncrementDate. (We could move isLeap to be contained within the scope of lastDayOfMonth.) Pairwise integration based on the decomposition in Figure 13.14 is problematic; the isLeap and lastDayOfMonth functions are never directly called by the Main program, so these integration sessions would be empty. Bottom–up pairwise integration starting with isLeap, then lastDayOfMonth, ValidDate, and GetDate would be useful. The pairs involving Main and GetDate, IncrementDate, and PrintDate are all useful (but short) sessions. Building stubs for ValidDate and lastDayOfMonth would be easy.

13.4.2 Call Graph–Based Integration

Pairwise integration based on the call graph in Figure 13.15 is an improvement over that for the decomposition-based pairwise integration. Obviously, there are no empty integration sessions because edges refer to actual unit references. There is still the problem of stubs. Sandwich integration is appropriate because this example is so small. In fact, it lends itself to a build sequence. Build 1 could contain Main and PrintDate. Build 2 could contain Main, IncrementDate, lastDayOfMonth, and IncrementDate in addition to the already present PrintDate. Finally, build 3 would add the remaining units, GetDate and ValidDate.

Neighborhood integration based on the call graph would likely proceed with the neighborhoods of ValidDate and lastDayOfMonth. Next, we could integrate the neighborhoods of GetDate and IncrementDate. Finally, we would integrate the neighborhood of Main. Notice that these neighborhoods form a build sequence.

integrationNextDate pseudocode

```
1  Main integrationNextDate 'start program event occurs here
       Type      Date
                 Month As Integer
                 Day As Integer
                 Year As Integer
       EndType
       Dim today, tomorrow As Date
2      Output("Welcome to NextDate!")
3      GetDate(today)                                                    'msg1
4      PrintDate(today)                                                  'msg2
5      tomorrow = IncrementDate(today)                                   'msg3
6      PrintDate(tomorrow)                                               'msg4
7  End Main
8  Function isLeap(year) Boolean
9      If (year divisible by 4)
10           Then
11               If (year is NOT divisible by 100)
12                     Then isLeap = True
13                     Else
14                         If (year is divisible by 400)
15                            Then isLeap = True
16                            Else isLeap = False
17                     EndIf
18               EndIf
19 Else isLeap = False
20 EndIf
21 End (Function isLeap)

22 Function lastDayOfMonth(month, year)    Integer
23     Case month Of
24           Case 1: 1, 3, 5, 7, 8, 10, 12
25              lastDayOfMonth = 31
26           Case 2: 4, 6, 9, 11
27              lastDayOfMonth = 30
28           Case 3: 2
29              If (isLeap(year))                                         'msg5
30                    Then lastDayOfMonth = 29
31                    Else lastDayOfMonth = 28
32              EndIf
33     EndCase
34 End (Function lastDayOfMonth)
35 Function GetDate(aDate)  Date
       dim aDate As Date

36     Function ValidDate(aDate)    Boolean 'within scope of GetDate
           dim aDate As Date
           dim dayOK, monthOK, yearOK As Boolean
37         If ((aDate.Month > 0) AND (aDate.Month <=12)
```

```
38              Then      monthOK = True
39                        Output("Month OK")
40              Else  monthOK = False
41                        Output("Month out of range")
42          EndIf
43          If (monthOK)
44              Then
45              If ((aDate.Day > 0) AND (aDate.Day <=
                lastDayOfMonth(aDate.Month, aDate.Year))                    'msg6
46                        Then      dayOK = True
47                                  Output("Day OK")
48                        Else      dayOK = False
49                                  Output("Day out of range")
50                    EndIf
51              EndIf
52          If ((aDate.Year > 1811) AND (aDate.Year <=2012)
53              Then      yearOK = True
54                        Output("Year OK")
55              Else      yearOK = False
56                        Output("Year out of range")
57          EndIf
58          If (monthOK AND dayOK AND yearOK)
59              Then ValidDate = True
60                        Output("Date OK")
61              Else ValidDate = False
62                    Output("Please enter a valid date")
63          EndIf
64      End (Function ValidDate)

    ' GetDate body begins here
65      Do
66          Output("enter a month")
67          Input(aDate.Month)
68          Output("enter a day")
69          Input(aDate.Day)
70          Output("enter a year")
71          Input(aDate.Year)
72          GetDate.Month = aDate.Month
73          GetDate.Day = aDate.Day
74          GetDate.Year = aDate.Year
75      Until (ValidDate(aDate))                                            'msg7
76      End (Function GetDate)
77      Function IncrementDate(aDate)Date
78          If (aDate.Day < lastDayOfMonth(aDate.Month))                    'msg8
79              Then aDate.Day = aDate.Day + 1
80              Else aDate.Day = 1
81                    If (aDate.Month = 12)
82                        Then      aDate.Month = 1
83                                  aDate.Year = aDate.Year + 1
84                        Else      aDate.Month = aDate.Month + 1
85                    EndIf
86          EndIf
87  End (IncrementDate)

88  Procedure PrintDate(aDate)
89      Output("Day is ", aDate.Month, "/", aDate.Day, "/", aDate.Year)
90  End (PrintDate)
```

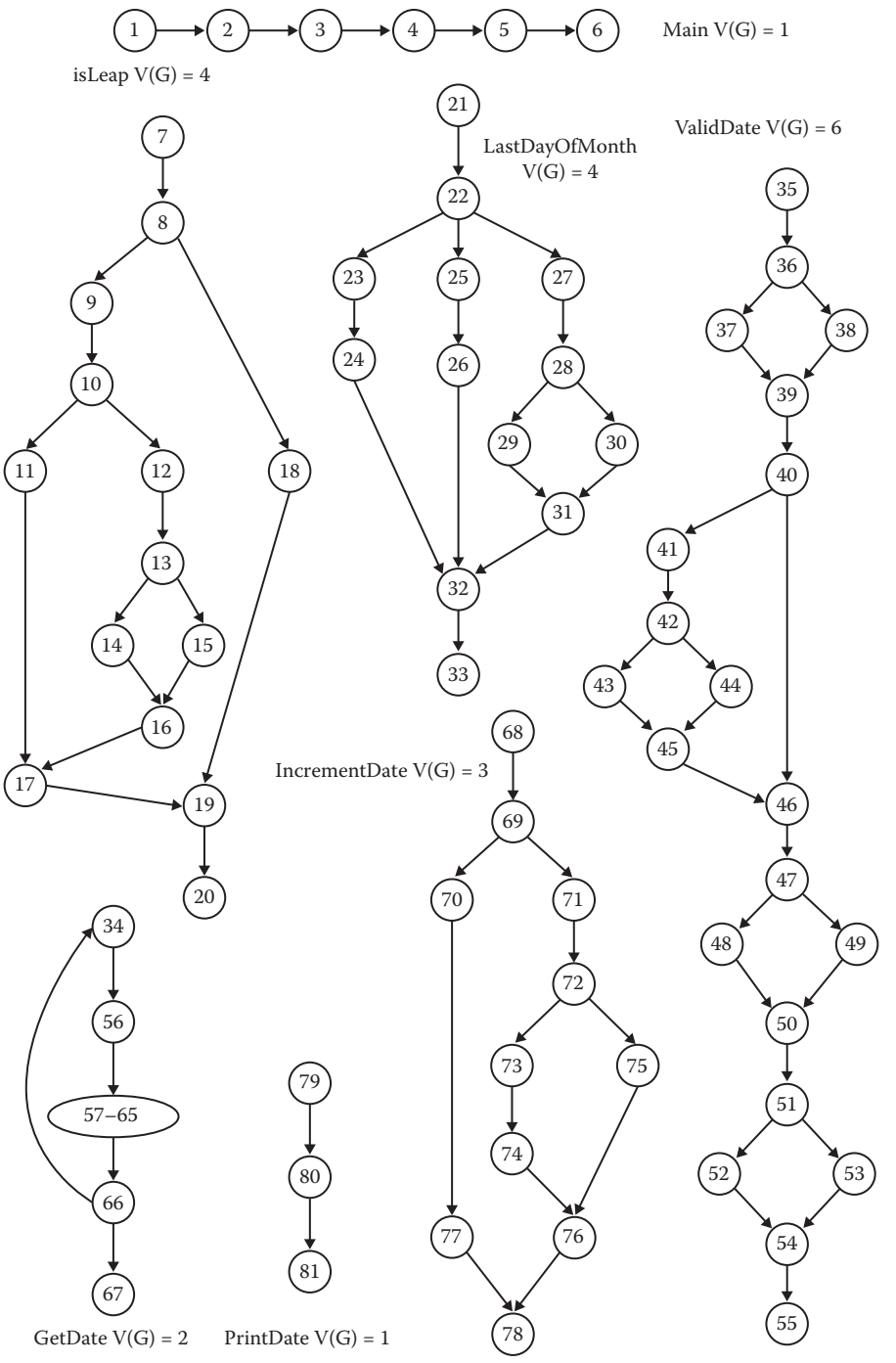

Figure 13.16 Program graphs of units in integrationNextDate.

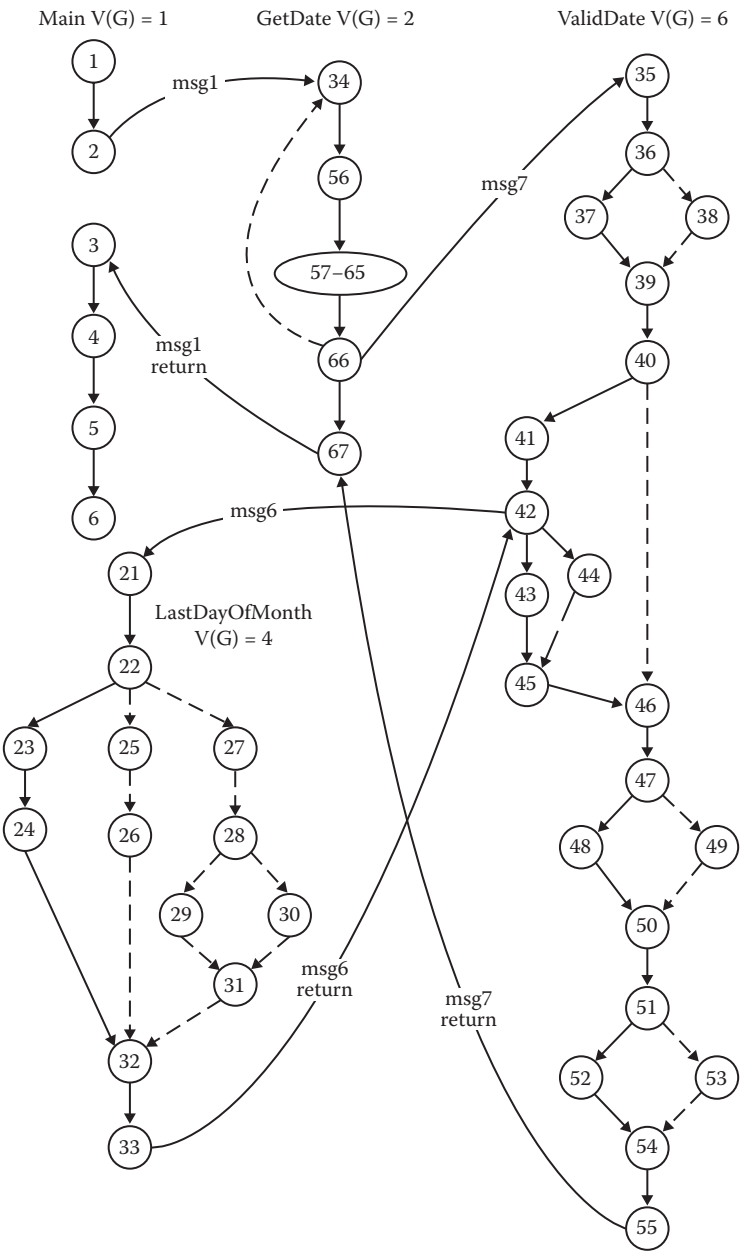

Figure 13.17 MM-path for May 27, 2012.

13.4.3 MM-Path-Based Integration

Because the program is data-driven, all MM-paths begin in and return to the main program. Here is the first MM-path for May 27, 2012 (there are others when the Main program calls PrintDate and IncrementDate). It is shown in Figure 13.17.

```
Main (1, 2)
     msg1
     GetDate (34, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66)
           msg7
           validDate (35, 36, 37, 39, 40, 41, 42))
                 msg6
                 lastDayOfMonth (21, 22, 23, 24, 32, 33)
                     'point of message quiescence
           ValidDate (43, 45, 46, 47, 48, 50, 51, 52, 54, 55)
     GetDate (67)
Main (3)
```

We are now in a strong position to describe how many MM-paths are sufficient: the set of MM-paths should cover all source-to-sink paths in the set of units. This is subtly present in Figure 13.17. The solid edges are in the MM-path, but the dashed edges are not. When loops are present, condensation graphs will result in directed acyclic graphs, thereby resolving the problem of potentially infinite (or excessively large) number of paths.

13.5 Conclusions and Recommendations

Table 13.3 summarizes the observations made in the preceding discussion. The significant improvement of MM-paths as a basis for integration testing is due to their exact representation of dynamic software behavior. MM-paths are also the basis for present research in data flow (define/use) approaches to integration testing. Integration testing with MM-paths requires extra effort. As a fallback position, perform integration testing based on call graphs.

Table 13.3 Comparison of Integration Testing Strategies

Strategy Basis	*Ability to Test Interfaces*	*Ability to Test Co-Functionality*	*Fault Isolation Resolution*
Functional decomposition	Acceptable but can be deceptive	Limited to pairs of units	Good, to faulty unit
Call graph	Acceptable	Limited to pairs of units	Good, to faulty unit
MM-path	Excellent	Complete	Excellent, to faulty unit execution path

EXERCISES

1. Find the source and sink nodes in isValidateDate and in getDate.
2. Write driver modules for isValidateDate and in getDate.
3. Write stubs for isValidateDate and in getDate.
4. Here are some other possible complexity metrics for MM-paths:
   V(G) = e – n
   V(G) = 0.5e – n + 2
   sum of the outdegrees of the nodes
   sum of the nodes plus the sum of the edges
   Make up some examples, try these out, and see if they have any explanatory value.
5. Make up a few test cases, interpret them as MM-paths, and then see what portions of the unit program graphs in Figure 13.16 are traversed by your MM-paths. Try to devise a "coverage metric" for MM-path-based integration testing.
6. One of the goals of integration testing is to be able to isolate faults when a test case causes a failure. Consider integration testing for a program written in a procedural programming language. Rate the relative fault isolation capabilities of the following integration strategies:
   A = Decomposition based top–down integration
   B = Decomposition based bottom–up integration
   C = Decomposition based sandwich integration
   D = Decomposition based "big bang" integration
   E = Call graph–based pairwise integration
   F = Call graph–based neighborhood integration (radius = 2)
   G = Call graph–based neighborhood integration (radius = 1)
   Show your ratings graphically by placing the letters corresponding to a strategy on the continuum below. As an example, suppose Strategies X and Y are about equal and not very effective, and Strategy Z is very effective.

```
      Y
      X                                                    Z
Low                                                        High
```

References

Deutsch, M.S., *Software Verification and Validation-Realistic Project Approaches*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Fordahl, M., *Elementary Mistake Doomed Mars Probe*, The Associated Press, available at http://mars.jpl.nasa.gov/msp98/news/mco990930.html, October 1, 1999.

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

Jorgensen, P.C., *The Use of MM-Paths in Constructive Software Development*, Ph.D. dissertation, Arizona State University, Tempe, AZ, 1985.

Jorgensen, P.C. and Erickson, C., Object-oriented integration testing, *Communications of the ACM*, September 1994.

Kaner, C., Falk, J. and Nguyen, H.Q., *Testing Computer Software*, 2nd ed., Van Nostrand Reinhold, New York, 1993.

Mosley, D.J., *The Handbook of MIS Application Software Testing*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1993.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 6th ed., McGraw-Hill, New York, 2005.

Schach, S.R., *Object-Oriented and Classical Software Engineering*, 5th ed., McGraw-Hill, New York, 2002.

Chapter 14

System Testing

Of the three levels of testing, the system level is closest to everyday experience. We test many things: a used car before we buy it, an online network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations—not with respect to a specification or a standard. Consequently, the goal is not to find faults but to demonstrate correct behavior. Because of this, we tend to approach system testing from a specification-based standpoint instead of from a code-based one. Because it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and this is compounded by the reduced testing interval that usually remains before a delivery deadline.

The craftsperson metaphor continues to serve us. We need a better understanding of the medium; we will view system testing in terms of threads of system-level behavior. We begin with a new construct—an Atomic System Function (ASF)—and develop the thread concept, highlighting some of the practical problems of thread-based system testing. System testing is closely coupled with requirements specification; therefore, we shall use appropriate system-level models to enjoy the benefits of model-based testing. Common to all of these is the idea of "threads," so we shall see how to identify system-level threads in a variety of common models. All this leads to an orderly thread-based system testing strategy that exploits the symbiosis between specification-based and code-based testing. We will apply the strategy to our simple automated teller machine (SATM) system, first described in Chapter 2.

14.1 Threads

Threads are hard to define; in fact, some published definitions are counterproductive, misleading, or wrong. It is possible to simply treat threads as a primitive concept that needs no formal definition. For now, we will use examples to develop a "shared vision." Here are several views of a thread:

A scenario of normal usage
A system-level test case
A stimulus/response pair
Behavior that results from a sequence of system-level inputs

An interleaved sequence of port input and output events
A sequence of transitions in a state machine description of the system
An interleaved sequence of object messages and method executions
A sequence of machine instructions
A sequence of source instructions
A sequence of MM-paths
A sequence of ASFs (to be defined in this chapter)

Threads have distinct levels. A unit-level thread is usefully understood as an execution–time path of source instructions or, alternatively, as a sequence of DD-paths. An integration-level thread is an MM-path—that is, an alternating sequence of module execution paths and messages. If we continue this pattern, a system-level thread is a sequence of ASFs. Because ASFs have port events as their inputs and outputs, a sequence of ASFs implies an interleaved sequence of port input and output events. The end result is that threads provide a unifying view of our three levels of testing. Unit testing tests individual functions; integration testing examines interactions among units; and system testing examines interactions among ASFs. In this chapter, we focus on system-level threads and answer some fundamental questions, such as, "How big is a thread? Where do we find them? How do we test them?"

14.1.1 Thread Possibilities

Defining the endpoints of a system-level thread is a bit awkward. We motivate a tidy, graph theory–based definition by working backward from where we want to go with threads. Here are four candidate threads in our SATM system:

Entry of a digit
Entry of a personal identification number (PIN)
A simple transaction: ATM card entry, PIN entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results
An ATM session containing two or more simple transactions

Digit entry is a good example of a minimal ASF. It begins with a port input event (the digit keystroke) and ends with a port output event (the screen digit echo), so it qualifies as a stimulus/response pair. This level of granularity is too fine for the purposes of system testing.

The second candidate, PIN entry, is a good example of an upper limit to integration testing and, at the same time, a starting point of system testing. PIN entry is a good example of an ASF. It is also a good example of a family of stimulus/response pairs (system-level behavior that is initiated by a port input event, traverses some programmed logic, and terminates in one of several possible responses [port output events]). PIN entry entails a sequence of system-level inputs and outputs.

1. A screen requesting PIN digits.
2. An interleaved sequence of digit keystrokes and screen responses.
3. The possibility of cancellation by the customer before the full PIN is entered.
4. A system disposition: a customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type; otherwise, a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.

Several stimulus/response pairs are evident, putting ASFs clearly in the domain of system-level testing. Other examples of ASFs include card entry, transaction selection, provision of transaction details, transaction reporting, and session termination. Each of these is maximal in an integration testing sense and minimal in a system testing sense. That is, we would not want to integration test something larger than an ASF; at the same time, we would not want to test anything smaller as part of system testing.

The third candidate, the simple transaction, has a sense of "end-to-end" completion. A customer could never execute PIN entry alone (a card entry is needed), but the simple transaction is commonly executed. This is a good example of a system-level thread; note that it involves the interaction of several ASFs.

The last possibility (the session) is actually a sequence of threads. This is also properly a part of system testing; at this level, we are interested in the interactions among threads. Unfortunately, most system testing efforts never reach the level of thread interaction.

14.1.2 Thread Definitions

We simplify our discussion by defining a new term that helps us get to our desired goal.

Definition

An *Atomic System Function (ASF)* is an action that is observable at the system level in terms of port input and output events.

In an event-driven system, ASFs are separated by points of event quiescence; these occur when a system is (nearly) idle, waiting for a port input event to trigger further processing. Event quiescence has an interesting Petri net insight. In a traditional Petri net, deadlock occurs when no transition is enabled. In an Event-Driven Petri Net (defined in Chapter 4), event quiescence is similar to deadlock; but an input event can bring new life to the net. The SATM system exhibits event quiescence in several places: one is the tight loop at the beginning of an ATM session, where the system has displayed the welcome screen and is waiting for a card to be entered into the card slot. Event quiescence is a system-level property; it is a direct analog of message quiescence at the integration level.

The notion of event quiescence does for ASFs what message quiescence does for MM-paths—it provides a natural endpoint. An ASF begins with a port input event and terminates with a port output event. When viewed from the system level, no compelling reason exists to decompose an ASF into lower levels of detail (hence, the atomicity). In the SATM system, digit entry is a good example of an ASF—so are card entry, cash dispensing, and session closing. PIN entry is probably too big; perhaps we should call it a molecular system function.

Atomic system functions represent the seam between integration and system testing. They are the largest item to be tested by integration testing and the smallest item for system testing. We can test an ASF at both levels. We will revisit the integrationNextDate program to find ASFs in Section 14.10.

Definition

Given a system defined in terms of ASFs, the *ASF graph* of the system is the directed graph in which nodes are ASFs and edges represent sequential flow.

Definition

A *source ASF* is an Atomic System Function that appears as a source node in the ASF graph of a system; similarly, a *sink ASF* is an Atomic System Function that appears as a sink node in the ASF graph.

In the SATM system, the card entry ASF is a source ASF, and the session termination ASF is a sink ASF. Notice that intermediary ASFs could never be tested at the system level by themselves—they need the predecessor ASFs to "get there."

Definition

A *system thread* is a path from a source ASF to a sink ASF in the ASF graph of a system.

These definitions provide a coherent set of increasingly broader views of threads, starting with very short threads (within a unit) and ending with interactions among system-level threads. We can use these views much like the ocular on a microscope, switching among them to see different levels of granularity. Having these concepts is only part of the problem; supporting them is another. We next take a tester's view of requirements specification to see how to identify threads.

14.2 Basis Concepts for Requirements Specification

Recall the notion of a basis of a vector space: a set of independent elements from which all the elements in the space can be generated (see problem 9, Chapter 8). Instead of anticipating all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs: data, actions, devices, events, and threads. Every system can be modeled in terms of these five fundamental concepts (and every requirements specification model uses some combination of these). We examine these fundamental concepts here to see how they support the tester's process of thread identification.

14.2.1 Data

When a system is described in terms of its data, the focus is on the information used and created by the system. We describe data in terms of variables, data structures, fields, records, data stores, and files. Entity/relationship (E/R) models are the most common choice at the highest level, and some form of a regular expression (e.g., Jackson diagrams or data structure diagrams) is used at a more detailed level. The data-centered view is also the starting point for several flavors of object-oriented analysis. Data refers to information that is either initialized, stored, updated, or (possibly) destroyed. In the SATM system, initial data describes the various accounts (each with its Personal Account Number, or PAN) and their PINs, and each account has a data structure with information such as the account balance. As ATM transactions occur, the results are kept as created data and used in the daily posting of terminal data to the central bank. For many systems, the data-centered view dominates. These systems are often developed in terms of CRUD actions (Create, Retrieve, Update, Delete). We could describe the transaction portion of the SATM system in this way, but it would not work well for the user interface portion.

Sometimes threads can be identified directly from the data model. Relationships among data entities can be one-to-one, one-to-many, many-to-one, or many-to-many; these distinctions all have implications for threads that process the data. For example, if bank customers can have several accounts, each account needs a unique PIN. If several people can access the same account, they need

ATM cards with identical PANs. We can also find initial data (such as PAN, ExpectedPIN pairs) that are read but never written. Such read-only data must be part of the system initialization process. If not, there must be threads that create such data. Read-only data is therefore an indicator of source ASFs.

14.2.2 Actions

Action-centered modeling is still a common requirements specification form. This is a historical outgrowth of the action-centered nature of imperative programming languages. Actions have inputs and outputs, and these can be either data or port events. Here are some methodology-specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service. Actions can also be decomposed into lower-level actions, most notably in the data flow diagrams of Structured Analysis. The input/output view of actions is exactly the basis of specification-based testing, and the decomposition (and eventual implementation) of actions is the basis of code-based testing.

14.2.3 Devices

Every system has port devices; these are the sources and destinations of system-level inputs and outputs (port events). The slight distinction between ports and port devices is sometimes helpful to testers. Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports. Physical actions (keystrokes and light emissions from a screen) occur on port devices, and these are translated from physical to logical (or logical to physical) forms. In the absence of actual port devices, much of system testing can be accomplished by "moving the port boundary inward" to the logical instances of port events. From now on, we will just use the term "port" to refer to port devices. The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doors, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on. (See Figure 14.1 for our working example.)

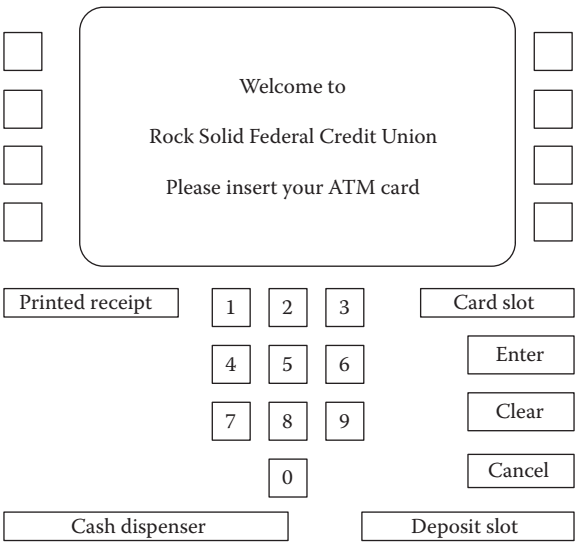

Figure 14.1 The Simple ATM (SATM) terminal.

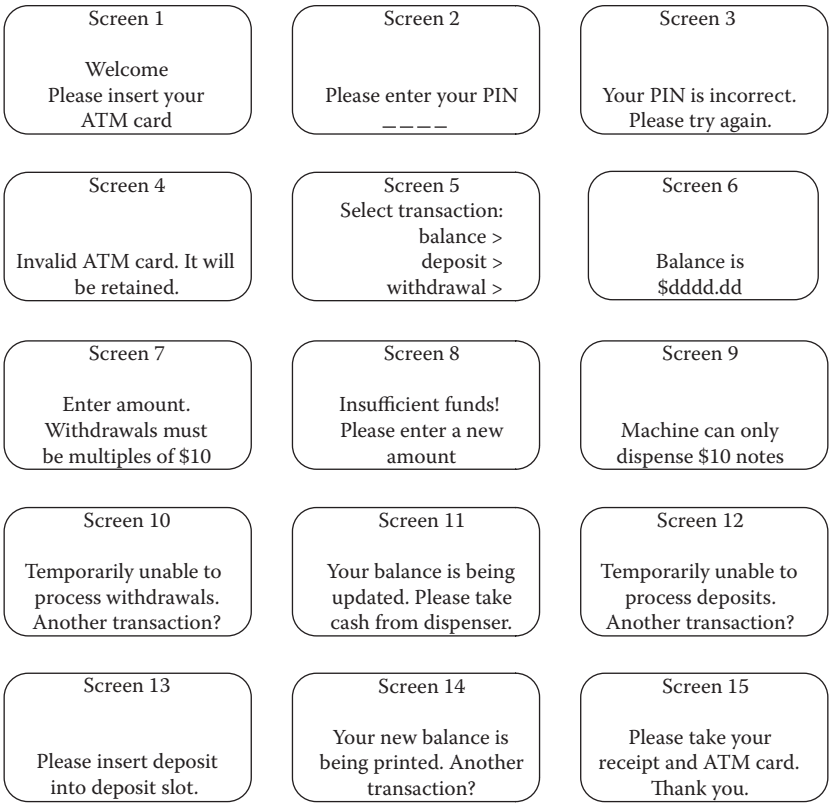

Figure 14.2 SATM screens.

Thinking about the ports helps the tester define both the input space that specification-based system testing needs; similarly, the output devices provide output-based test information. For example, we would like to have enough threads to generate all 15 SATM screens in Figure 14.2.

14.2.4 Events

Events are somewhat schizophrenic: they have some characteristics of data and some of actions. An event is a system-level input (or output) that occurs on a port device. Similar to data, events can be inputs to or outputs of actions. Events can be discrete (such as SATM keystrokes) or they can be continuous (such as temperature, altitude, or pressure). Discrete events necessarily have a time duration, and this can be a critical factor in real-time systems. We might picture input events as destructive read-out data, but it is a stretch to imagine output events as destructive write operations.

Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations, and, symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers). Situations occur where the context of present data values changes the logical meaning of physical events. In the SATM system, for example, the port input event of depressing button B1

means "balance" when screen 5 is displayed, "checking" when screen 6 is displayed, and "yes" when screens 10, 11, and 14 are displayed. We refer to these situations as "context-sensitive port events," and we would expect to test such events in each context.

14.2.5 Threads

Unfortunately for testers, threads are the least frequently used of the five fundamental constructs. Because we test threads, it usually falls to the tester to find them in the interactions among the data, events, and actions. About the only place that threads appear per se in a requirements specification is when rapid prototyping is used in conjunction with a scenario recorder. It is easy to find threads in control models, as we will soon see. The problem with this is that control models are just that—they are models, not the reality of a system.

14.2.6 Relationships among Basis Concepts

Figure 14.3 is an E/R model of our basis concepts. Notice that all relationships are many-to-many: Data and Events are inputs to or outputs of the Action entity. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions. This diagram demonstrates some of the difficulty of system testing. Testers must use events and threads to ensure that all the many-to-many relationships among the five basis concepts are correct.

14.3 Model-Based Threads

In this section, we will use the SATM system (defined in Chapter 2) to illustrate how threads can be identified from models. Figure 14.2 shows the 15 screens needed for SATM. (This is really a bare bones, economy ATM system!)

Finite state machine models of the SATM system are the best place to look for system testing threads. We will start with a hierarchy of state machines; the upper level is shown in Figure 14.4. At this level, states correspond to stages of processing, and transitions are caused by abstract logical (instead of port) events. The card entry "state," for example, would be decomposed into lower levels that deal with details such as jammed cards, cards that are upside down, stuck card rollers,

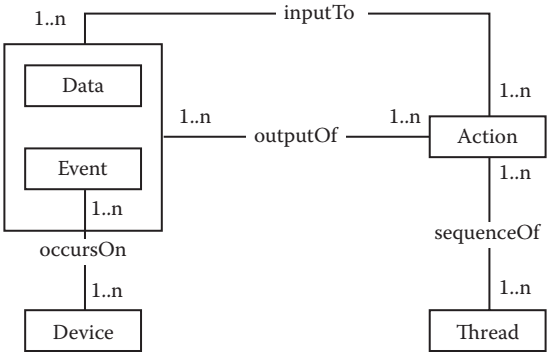

Figure 14.3 E/R model of basis concepts.

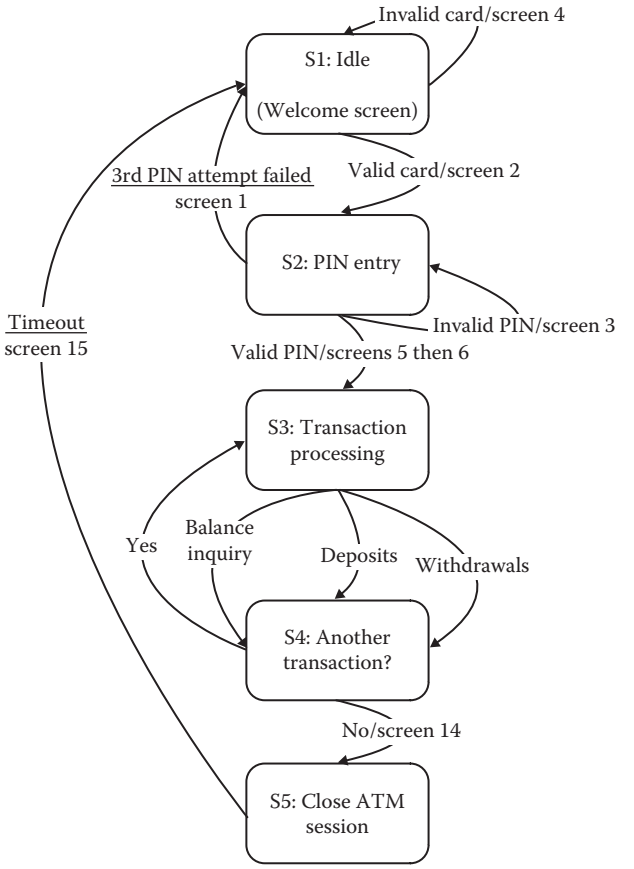

Figure 14.4 Uppermost level SATM finite state machine.

and checking the card against the list of cards for which service is offered. Once the details of a macro-state are tested, we use a simple thread to get to the next macro-state.

The PIN entry state S2 is decomposed into the more detailed view in Figure 14.5. The adjacent states are shown because they are sources and destinations of transitions from the PIN entry state at the upper level. (This approach to decomposition is reminiscent of the old data flow diagramming idea of balanced decomposition.) At the S2 decomposition, we focus on the PIN retry mechanism; all of the output events are true port events, but the input events are still logical events.

The transaction processing state S3 is decomposed into a more detailed view in Figure 14.6. In that finite state machine, we still have abstract input events, but the output events are actual port events. State 3.1 requires added information. Two steps are combined into this state: choice of the account type and selection of the transaction type. The little "<" and ">" symbols are supposed to point to the function buttons adjacent to the screen, as shown in Figure 14.1. As a side note, if this were split into two states, the system would have to "remember" the account type choice in the first state. However, there can be no memory in a finite state machine, hence the combined state. Once again, we have abstract input events and true port output events.

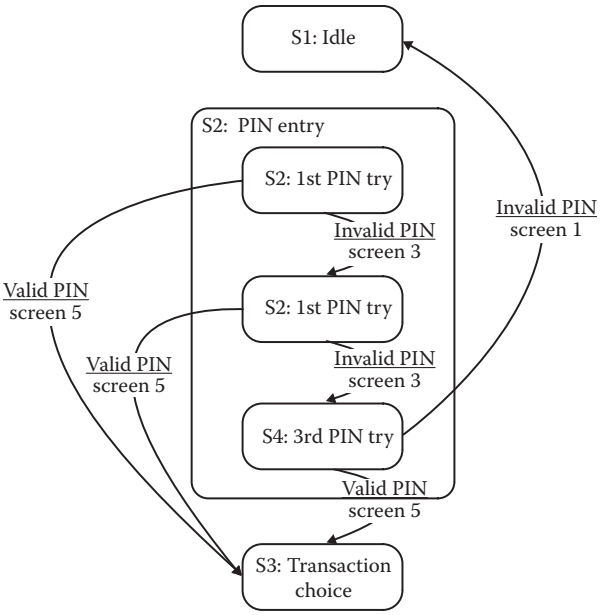

Figure 14.5 Decomposition of PIN entry state.

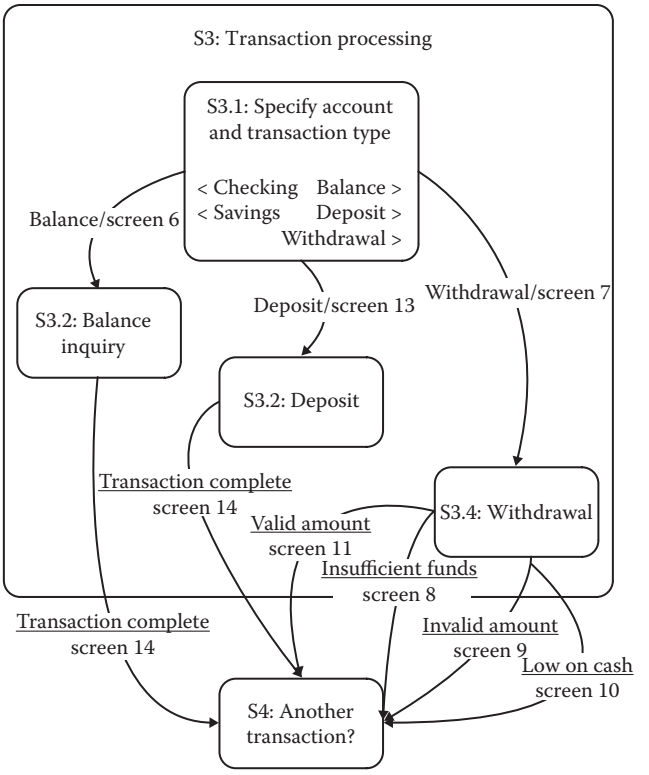

Figure 14.6 Decomposition of transaction processing state.

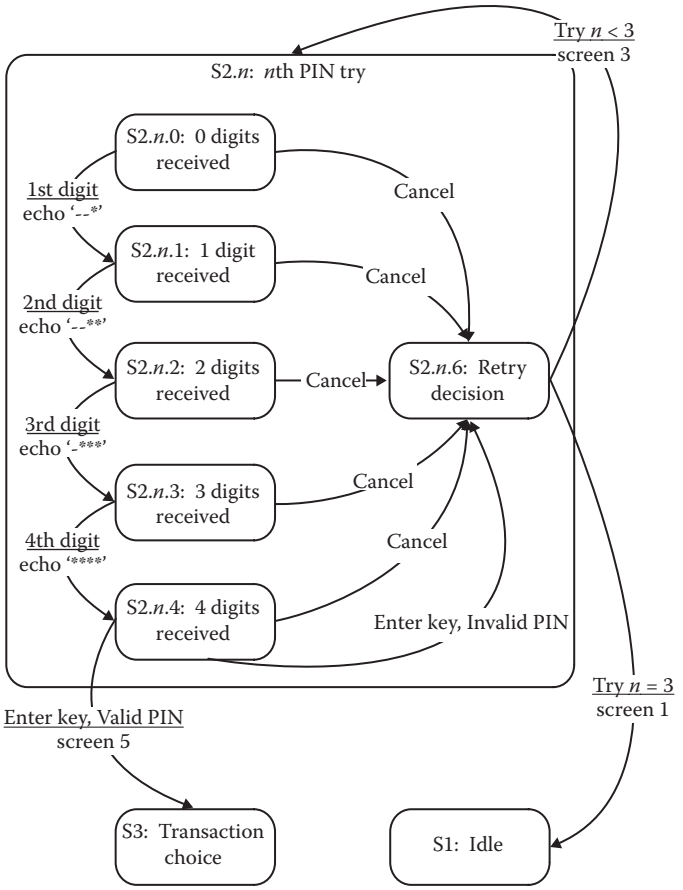

Figure 14.7 Decomposition of PIN try states.

Our final state decomposition is applied to see the details of PIN entry tries S2.1, S2.2, and S2.3 (see Figure 14.7). Each PIN try is identical, so the lower-level states are numbered S2.*n*, where *n* signifies the PIN try. We almost have true input events. If we knew that the expected PIN was "2468" and if we replaced the digit entries, for example, "first digit," with "2," then we would finally have true port input events. A few abstract inputs remain—those referring to valid and invalid PINs and conditions on the number of tries.

It is good form to reach a state machine in which transitions are caused by actual port input events, and the actions on transitions are port output events. If we have such a finite state machine, generating system test cases for these threads is a mechanical process—simply follow a path of transitions and note the port inputs and outputs as they occur along the path. Table 14.1 traces one such path through the PIN try finite state machine in Figure 14.7. This path corresponds to a thread in which a PIN is correctly entered on the first try. To make the test case explicit, we assume a precondition that the expected PIN is "2468." The event in parentheses in the last row of Table 14.1 is the logical event that "bumps up" to the parent state machine and causes a transition there to the Await Transaction Choice state.

Table 14.1 Port Event Sequence for Correct PIN on First Try

Port Input Event	*Port Output Event*
	Screen 2 displayed with '- - - -'
2 Pressed	
	Screen 2 displayed with '- - - *'
4 Pressed	
	Screen 2 displayed with '- - * *'
6 Pressed	
	Screen 2 displayed with '- * * *'
8 Pressed	
	Screen 2 displayed with '* * * *'
(Valid PIN)	Screen 5 displayed

The most common products for model-based testing (Jorgensen, 2009) start with a finite state machine description of the system to be tested and then generate all paths through the graph. If there are loops, these are (or should be) replaced by two paths, as we did at the program graph level in Chapter 8. Given such a path, the port inputs that cause transitions are events in a system test case; similarly for port outputs that occur as actions on transitions.

Here is a hard lesson from industrial experience. A telephone switching system laboratory tried defining a small telephone system with finite state machines. The system, a Private Automatic Branch Exchange (PABX), was chosen because, as switching systems go, it is quite simple. There was a grizzled veteran system tester, Casimir, assigned to help with the development of the model. He was well named. According to Wikipedia, his name means "someone who destroys opponent's prestige/glory during battle" (http://en.wikipedia.org/wiki/Casimir). Throughout the process, Casimir was very suspicious, even untrusting. The team reassured him that, once the project was finished, a tool would generate literally several thousand system test cases. Even better, this provided a mechanism to trace system testing directly back to the requirements specification model. The actual finite state machine had more than 200 states, and the tool generated more than 3000 test cases. Finally, Casimir was impressed, until one day when he discovered an automatically generated test case that was logically impossible. On further, very detailed analysis, the invalid test case was derived from a pair of states that had a subtle dependency (and finite state machines must have independent states). Out of 200-plus states, recognizing such dependencies is extremely difficult. The team explained to Casimir that the tool could analyze any thread that traversed the pair of dependent states, thereby identifying any other impossible threads. This technical triumph was short-lived, however, when Casimir asked if the tool could identify any other pairs of dependent states. No tool can do this because this would be equivalent to the famous Halting Problem. The lesson: generating threads from finite state machines is attractive, and can be quite effective; however, care must be taken to avoid both memory and dependence issues.

14.4 Use Case–Based Threads

Use Cases are a central part of the Unified Modeling Language (UML). Their main advantage is that they are easily understood by both customers/users and developers. They capture the *does view* that emphasizes behavior, rather than the *is view* that emphasizes structure. Customers and testers both tend to naturally think of a system in terms of the *does view*, so use cases are a natural choice.

14.4.1 Levels of Use Cases

One author (Larman, 2001) defines a hierarchy of use cases in which each level adds information to the predecessor level. Larman names these levels as follows:

- High level (very similar to an agile user story)
- Essential
- Expanded essential
- Real

The information content of these variations is shown in Venn diagram form in Figure 14.8.

Tables 14.2 through 14.4 show the gradual increase in Larman's use case hierarchy for the example in Table 14.1. High-level use cases are at the level of the user stories used in agile development. A set of high-level use cases gives a quick overview of the *does view* of a system. Essential use cases add the sequence of port input and output events. At this stage, the port boundary begins to become clear to both the customer/user and the developer.

Expanded essential use cases add pre- and postconditions. We shall see that these are key to linking use cases when they are expressed as system test cases.

Real use cases are at the actual system test case level. Abstract names for port events, such as "invalid PIN," are replaced by an actual invalid PIN character string. This presumes that some

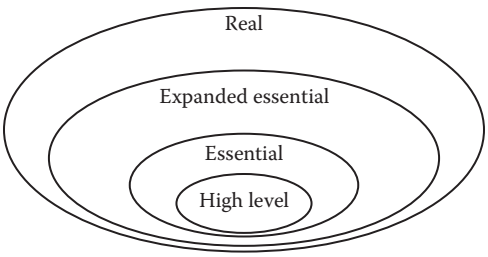

Figure 14.8 Larman's levels of use cases.

Table 14.2 High-Level Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	HLUC-1
Description	A customer enters the PIN number correctly on the first attempt.

Table 14.3 Essential Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	EUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches 1st digit	
	3. Screen 2 shows '- - - * '
4. Customer touches 2nd digit	
	5. Screen 2 shows '- - * * '
6. Customer touches 3rd digit	
	7. Screen 2 shows '- * * * '
8. Customer touches 4th digit	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed

form of testing database has been assembled. In our SATM system, this would likely include several accounts with associated PINs and account balances (Table 14.5).

14.4.2 An Industrial Test Execution System

This section describes a system for automatic test execution that I was responsible for in the early 1980s. Since it was intended for executing regression test cases (a very boring manual assignment), it was named the Automatic Regression Testing System (ARTS). This is as close as I ever came to the art world. The ARTS system had a human readable system test case language that was interpretively executed on a personal computer. In the ARTS language, there were two verbs: CAUSE would cause a port input event to occur, and VERIFY would observe a port output event. In addition, a tester could refer to a limited number of devices and to a limited number of input events associated with those devices. Here is a small paraphrased excerpt of a typical ARTS test case.

```
CAUSE Go-Offhook On Line 4
VERIFY Dialtone On Line 4
CAUSE TouchDigit '3' On Line 4
VERIFY NoDialtone On Line 4
```

Table 14.4 Expanded Essential Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	EEUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions	1. The expected PIN is known
	2. Screen 2 is displayed
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches 1st digit	
	3. Screen 2 shows '- - - * '
4. Customer touches 2nd digit	
	5. Screen 2 shows '- - * * '
6. Customer touches 3rd digit	
	7. Screen 2 shows '- * * * '
8. Customer touches 4th digit	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	
Postconditions	Select Transaction screen is active

The physical connection to a telephone prototype required a harness that connected the personal computer with actual prototype ports. The test case language consisted of the CAUSE and VERIFY verbs, names for port input and output events, and names for available devices that were connected to the harness. On the input side, the harness accomplished a logical-to-physical transformation, with the symmetric physical-to-logical transformation on the output side. The basic architecture is shown in Figure 14.9.

We learned a lesson in human factors engineering. The test case language was actually free form, and the interpreter eliminated noise words. The freedom to add noise words was intended to give test case designers a place to put additional notes that would not be executed, but would be kept in the test execution report. The result was test cases like this (so much for test designer freedom):

> *As long as it is not raining, see if you can CAUSE a Go-Offhook event right away On Line 4, and then, see if you can VERIFY that some variation of Dialtone happened to occur*

Table 14.5 Real Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	RUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions	1. The expected PIN is "2468"
	2. Screen 2 is displayed
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches digit 2	
	3. Screen 2 shows '- - - * '
4. Customer touches digit 4	
	5. Screen 2 shows '- - * * '
6. Customer touches digit 6	
	7. Screen 2 shows '- * * * '
8. Customer touches digit 8	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	(normally done at this point)
Postconditions	Select Transaction is active

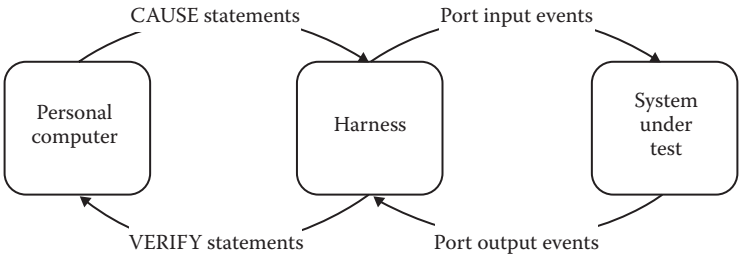

Figure 14.9 Automated test execution system architecture.

On Line 4. Then, if you are in a good mood, why not CAUSE a TouchDigit '3' action On Line 4. Finally, (at last!), see if you can VERIFY that NoDialtone is present On Line 4.

In retrospect, the ARTS system predated the advent of use cases. Notice how the event sequence portion of a real use case is dangerously close to an ARTS test case. I learned later that the system evolved into a commercial product that had a 15-year lifetime.

14.4.3 System-Level Test Cases

A system-level test case, whether executed manually or automatically, has essentially the same information as a real use case (Table 14.6).

Table 14.6 System Test Case for Correct PIN on First Try

Test case name	Correct PIN entry on first try
Test case ID	TC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions needed to run this test case	1. The expected PIN is "2468"
	2. Screen 2 is displayed
Event Sequence	
Input events (performed by tester)	*Output events (observed by system tester)*
	1. Screen 2 shows '- - - - '
2. Touch digit 2	
	3. Screen 2 shows '- - - * '
4. Customer touches digit 4	
	5. Screen 2 shows '- - * * '
6. Customer touches digit 6	
	7. Screen 2 shows '- * * * '
8. Customer touches digit 8	
	9. Screen 2 shows '- - - * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	
Postconditions	Select Transaction is active
Test execution result?	Pass/Fail
Test run by	<tester's name> date

14.4.4 Converting Use Cases to Event-Driven Petri Nets

Event-Driven Petri Nets (EDPNs) were defined in Chapter 4. They were originally developed for use in telephone switching systems. As the name implies, they are appropriate for any event-driven system, particularly those characterized by context-sensitive port input events. In an EDPN drawing, port events are shown as triangles, data places are circles, transitions are narrow rectangles, and the input and output connections are arrows. In an attempt at human factors design,

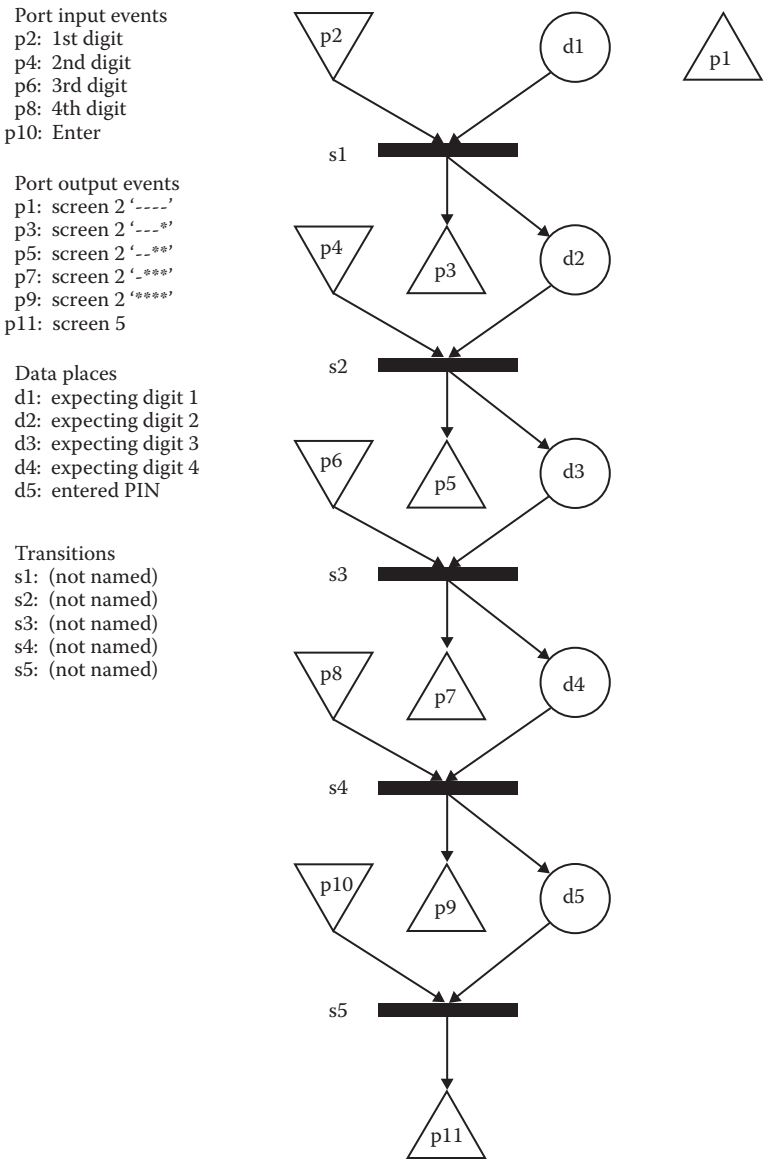

Figure 14.10 Event-Driven Petri Net for correct PIN on first try.

EDPN diagrams show input port events as a downward pointing triangle as if they were a funnel. Similarly, output port events are upward pointing, as if they were megaphones. Figure 14.10 is the EDPN for our continuing example, Correct PIN on First Try.

Automatic derivation of an EDPN from a use case is only partly successful—port input events can be derived from the input portion of the event sequence, similarly for port output events. Also, the interleaved order of input and output events can be preserved. Finally, the pre- and postconditions are mapped to data places. There are a few problems, however.

1. The most obvious is the port output event p1 that refers to screen 2 being displayed with four blank positions for the PIN to be entered. It is an orphan, in the sense that it is not created by a transition.
2. Transitions are not named. If a person were to develop the EDPN, there would likely be descriptive names for the transitions, for example, s1: accept first digit.
3. An immediate cause-and-effect connection is presumed. This would fail if two out-of-sequence input events were required to produce an output event.
4. There is no provision for intermediate data that may be produced.
5. The data places d1–d5 do not appear in the use case. (They could be derived from the finite state machine, however.)

One answer to this is to follow the lead of formal systems and define the information content of a "well-formed use case." At a minimum, a well-formed use case should conform to these requirements.

1. The event sequence cannot begin with an output event. This could just be considered as a precondition.
2. The event sequence cannot end with an input event. This could just be considered as a postcondition.
3. Preconditions must be both necessary and sufficient to the use case. There are no superfluous preconditions, and every precondition must be used or needed by the use case. Similarly for postconditions.
4. There must be at least one precondition and at least one postcondition.

The value in deriving EDPNs from use cases is that, because they are special cases of Petri Nets, they inherit a wealth of analytical possibilities. Here are some analyses that are easy with Petri Nets, and all but impossible with use cases:

1. Interactions among use cases, such as one use case being a prerequisite for another
2. Use cases that are in conflict with others
3. Context-sensitive input events
4. Inverse use cases, where one "undoes" the other

14.4.5 Converting Finite State Machines to Event-Driven Petri Nets

Mathematically speaking, finite state machines are a special case of ordinary Petri Nets in which every Petri Net transition has one input place and one output place. Since EDPNs are an extension of ordinary Petri Nets, the conversion of finite state machines to EDPNs is guaranteed. Figure 14.11 shows a portion of the finite state machine in Figure 14.7 converted to an EDPN.

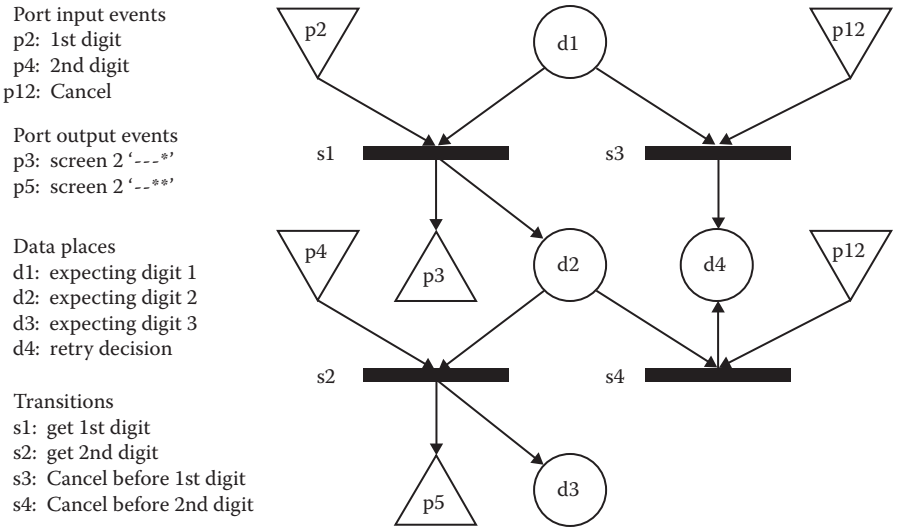

Figure 14.11 Event-Driven Petri Net from part of finite state machine in Figure 14.12.

In Figure 14.11, input events p2 and p12 can both occur when the ATM is awaiting the first PIN digit. Similarly, input events p4 and p12 can both occur when the ATM is awaiting the second PIN digit. Close examination shows three distinct paths. There are two main ways to describe these paths—as a sequence of port input events, or as a sequence of EDPN transitions. Using the latter, the three paths in Figure 14.11 are <s1, s2>, <s1, s4>, and <s3>. There is an interesting connection between EDPNs and obscure database terminology. The *intention* of a database is the underlying data model. Different populations of the intention are known as *extensions* of the database. The intention of a given database is unique, but there can be myriad possible extensions. The same is true for unmarked versus marked EDPNs—an unmarked EDPN can have many possible marking sequence executions.

14.4.6 Which View Best Serves System Testing?

Of the three views in this section, use cases are the best for communication between customers/users and developers; however, they do not support much in the way of analysis. Finite state machines are in common use, but composing finite state machines inevitably leads to the well-known "finite state machine explosion." Finite state machine–based support tools are available, but the explosion part is problematic. Both of these notations can be converted to EDPNs, although some information must be added to EDPNs derived from use cases. The big advantage of EDPNs is that they are easily composed with other EDPNs, and the intension/extension relationship between an unmarked EDPN and various markings (execution sequences) makes them the preferred choice for system testing. We did not discuss deriving system test cases from an EDPN, but the process is obvious.

14.5 Long versus Short Use Cases

There is an element of foreshadowing in the preceding material. Early on, we spoke of various thread candidates. In that discussion, we saw a range of very short to very long threads. Each of these choices translates directly to our three models, use cases, finite state machines, and EDPNs.

In the use case domain, the usual view is that a use case is a full, end-to-end, transaction. For the SATM system, full use cases would start and end with screen 1, the Welcome screen. In between, some path would occur, either as a specific use case, a path in the finite state machine, or as a marking in the full EDPN. The problem with this is that there is a large number of paths in either model, and a large number of individual use cases. The end-to-end use case is a "long use case." The use cases in Chapter 22 (Software Technical Reviews) are short use cases. As a quick example of a long use case, consider a story with the following sequence:

> *A customer enters a valid card, followed by a valid PIN entry on the first try. The customer selects the Withdraw option, and enters $20 for the withdrawal amount. The SATM system dispenses two $10 notes and offers the customer a chance to request another transaction. The customer declines, the SATM system updates the customer account, returns the customer's ATM card, prints a transaction receipt, and returns to the Welcome screen.*

Here we suggest "short use cases," which begin with a port input event and end with a port output event. Short use cases must be at the Expanded Essential Use Case level, so the pre- and postconditions are known. We can then develop sequences of short use cases with the connections based on the pre- and postconditions. Short use case B can follow short use case A if the postconditions of A are consistent with the preconditions of B. The long use case above might be expressed in terms of four short use cases:

1. Valid card
2. Correct PIN on first try
3. Withdrawal of $20
4. Select no more transitions

Table 14.7 Short Use Cases for Successful SATM Transactions

Short Use Case	*Description*
SUC1	Valid ATM card swipe
SUC2	Invalid ATM card swipe
SUC3	Correct PIN attempt
SUC4	Failed PIN attempt
SUC5	Choose Balance
SUC6	Choose Deposit
SUC7	Choose Withdrawal: valid withdrawal amount
SUC8	Choose Withdrawal: amount not a multiple of $20
SUC9	Choose Withdrawal: amount greater than account balance
SUC10	Choose Withdrawal: amount greater than daily limit
SUC11	Choose no other transaction
SUC12	Choose another transaction

The motivation for short use cases is that there are 1909 possible paths through the SATM finite state machine in Figure 14.5 considering all four state decompositions. The great majority of these are due to failed PIN entry attempts (six ways to fail, one way to be successful). Table 14.7 lists a useful set of short use cases for successful SATM transactions. If we added short use cases for all the ways that PIN entry can fail, we would have good coverage of the SATM system.

Figure 14.12 shows the short use cases from Table 14.7 linked with respect to a slightly different finite state machine model of the SATM system.

What about the ways PIN entry can fail? One could argue that this is really a unit-level question; hence, we do not need short use cases for these possibilities. On the other hand, there are only 13 transitions in the detailed view of PIN entry in Figure 14.7, and we have port inputs and outputs for the digit entry transitions. (The transitions for the five possible points of cancellation and the invalid four-digit PIN all have an intermediate state to simplify the figure.) Table 14.8 lists the short use cases for complete PIN entry coverage.

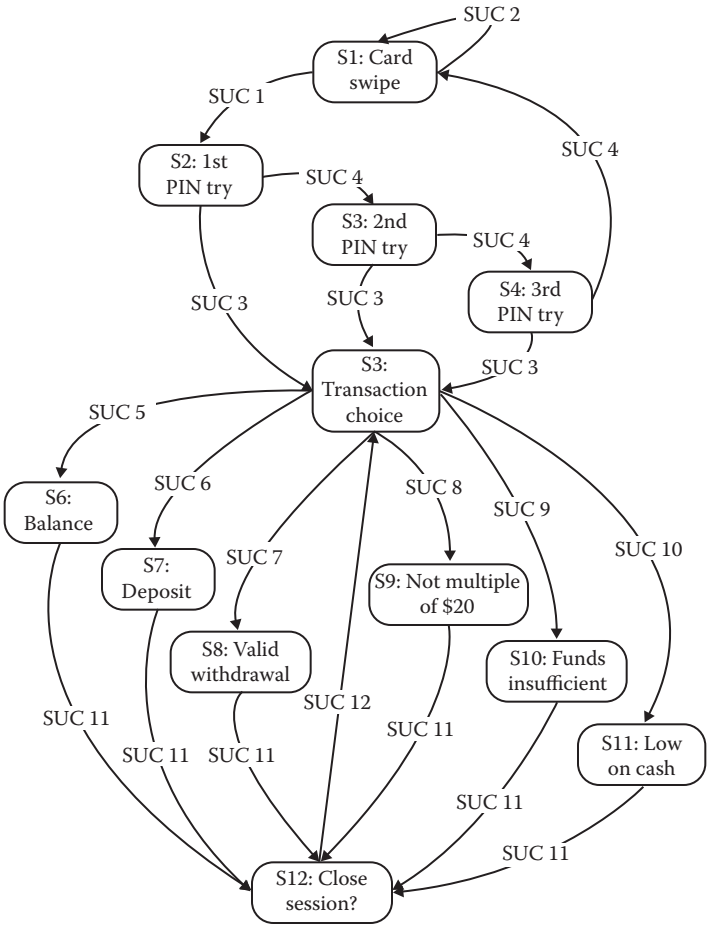

Figure 14.12 SATM finite state machine with short use cases causing transitions.

Table 14.8 Short Use Cases for Failed PIN Entry Attempts

Short Use Case	*Description*
SUC13	Digit 1 entered
SUC14	Digit 2 entered
SUC15	Digit 3 entered
SUC16	Digit 4 entered
SUC17	Enter with valid PIN
SUC18	Cancel before digit 1
SUC19	Cancel after digit 1
SUC20	Cancel after digit 2
SUC21	Cancel after digit 3
SUC22	Cancel after digit 4
SUC23	Enter with invalid PIN
SUC24	Next PIN try
SUC25	Last PIN try

Now we see the advantage of short use cases—1909 long use cases are covered by the 25 short use cases. The advantage of model-based testing becomes yet clearer with this compression. Note the similarity to node and edge test coverage at the unit level that we saw in Chapter 8.

14.6 How Many Use Cases?

When a project is driven by use cases, there is the inevitable question as to how many use cases are needed. Use case–driven development is inherently a bottom–up process. In the agile world, the answer is easy—the customer/user decides how many use cases are needed. But what happens in a non-agile project? Use case–driven development is still (or can be) an attractive option. In this section, we examine four strategies to help decide how many bottom–up use cases are needed. Each strategy employs an incidence matrix (see Chapter 4). We could have a fifth strategy if bottom–up use case development is done in conjunction with a gradually developed model, as we just saw in Section 14.5.

14.6.1 Incidence with Input Events

As use cases are identified jointly between the customer/user and developers, both parties gradually identify port-level input events. This very likely is an iterative process, in which use cases provoke the recognition of port input events, and they, it turn, suggest additional use cases. These are kept in an incidence showing which use cases require which port input events. As the process continues, both parties reach a point where the existing set of input events is sufficient for any new use case. Once this point is reached, it is clear that a minimal set of use cases covers all the

Table 14.9 SATM Port Input Events

Port Input Event	*Description*
e1	Valid ATM card swipe
e2	Invalid ATM card swipe
e3	Correct PIN attempt
e4	Touch Enter
e5	Failed PIN
e6	Touch Cancel
e7	Touch button corresponding to Checking
e8	Touch button corresponding to Savings
e9	Choose Balance
e10	Choose Deposit
e11	Enter deposit amount
e12	Choose Withdrawal
e13	Enter withdrawal amount
e14	Valid withdrawal amount
e15	Withdrawal amount not a multiple of $20
e16	Withdrawal amount greater than account balance
e17	Withdrawal amount greater than cash in SATM
e18	Touch button corresponding to Yes
e19	Touch button corresponding to No

port level input events. Table 14.9 lists (most of) the port input events for the SATM system, and Table 14.10 shows their incidence with the first 12 short use cases. Exercise 5 asks you to develop a similar table for the PIN entry use cases.

Both Tables 14.9 and 14.10 should be understood as the result of an iterative process, which is inevitable in a bottom–up approach. If port inputs are identified that are not used anywhere, we know we need at least one more short use case. Similarly, if we have a short use case that does not involve any of the existing port input events, we know we need at least one more event.

14.6.2 Incidence with Output Events

The matrix showing the incidence of short use cases with port output events is developed in the same iterative way as that for input events. Table 14.11 lists the port output events for the SATM system. As you develop this incidence matrix (see Exercise 5), you should note if any screen is never used in the finite state machines in Figures 14.4 through 14.7. This is also revisited in Chapter 22 on Software Technical Reviews.

Table 14.10 Short Use Case Incidence Matrix with Port Input Events

	Port Input Events																		
SUC	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
1	X																		
2		X																	
3			X	X															
4				X	X	X													
5							X	X	X										
6							X	X		X	X								
7							X	X				X	X	X					
8							X	X				X	X		X				
9							X	X				X	X			X			
10							X	X				X	X				X		
11																			X
12																		X	

Table 14.11 SATM Port Output Events

Port Input Event	*Description*
Screen 1	Welcome. Please insert your ATM card.
Screen 2	Please enter your PIN.
Screen 3	Your PIN is incorrect. Please try again.
Screen 4	Invalid ATM card. It will be retained.
Screen 5	Select transaction: balance, deposit, or withdrawal.
Screen 6	Your account balance is $_ _ _._.
Screen 7	Enter withdrawal amount. Must be a multiple of $10.
Screen 8	Insufficient funds. Please enter a new withdrawal amount.
Screen 9	Sorry. Machine can only dispense $10 notes.
Screen 10	Temporarily unable to process withdrawals. Another transaction?
Screen 11	Your balance is updated. Please remove cash from dispenser.
Screen 12	Temporarily unable to process deposits. Another transaction?
Screen 13	Please insert deposit envelope into slot.
Screen 14	Your new balance is being printed. Another transaction?
Screen 15	Please take your receipt and ATM card.

14.6.3 Incidence with All Port Events

In practice, the incidence approach needs to be done for all port-level events. This is just a combination of the discussions in Sections 16.2.1 and 16.2.2. One advantage of the full port event incidence matrix is that it can be reordered in useful ways. For example, the short use cases could be placed together, or possibly listed in a sensible transaction-based order. The port events can also be permuted into cohesive subgroups, for example, input events related to PIN entry, or output events related to deposits.

14.6.4 Incidence with Classes

There is a perennial debate among object-oriented developers as to how to begin—use cases first, or classes first. One of my colleagues (a very classy person) insists on the class-first approach, while others are more comfortable with the use case first view. A good compromise is to develop an incidence matrix showing which classes are needed to support which use cases. Often, it is easier to identify classes for a use case, rather than for a full system. As with the other incidence matrices, this approach provides a good answer to when a sufficient set of classes has been identified.

14.7 Coverage Metrics for System Testing

In Chapter 10, we saw the advantage of combining specification-based and code-based testing techniques, because they are complementary. We are in the same position now with system testing. The model–based approaches of Section 14.3 can be combined with the use case–based approaches of Section 14.4. Further, the incidence matrices of Section 14.6 can be used as a basis for specification-based system test coverage metrics.

14.7.1 Model-Based System Test Coverage

We can use model-based metrics as a cross-check on use case–based threads in much the same way that we used DD-paths at the unit level to identify gaps and redundancies in specification-based test cases. We really have pseudostructural testing (Jorgensen, 1994) because the node and edge coverage metrics are defined in terms of a model of a system, not derived directly from the system implementation. In general, behavioral models are only approximations of a system's reality, which is why we could decompose our models down to several levels of detail. If we made a true code-based model, its size and complexity would make it too cumbersome to use. The big weakness of model-based metrics is that the underlying model may be a poor choice. The three most common behavioral models (decision tables, finite state machines, and Petri nets) are appropriate to transformational, interactive, and concurrent systems, respectively.

Decision tables and finite state machines are good choices for ASF testing. If an ASF is described using a decision table, conditions typically include port input events, and actions are port output events. We can then devise test cases that cover every condition, every action, or, most completely, every rule. For finite state machine models, test cases can cover every state, every transition, or every path.

Thread testing based on decision tables is cumbersome. We might describe threads as sequences of rules from different decision tables, but this becomes very messy to track in terms of coverage. We need finite state machines as a minimum, and if any form of interaction occurs, Petri nets are a better choice. There, we can devise thread tests that cover every place, every transition, and every sequence of transitions.

14.7.2 Specification-Based System Test Coverage

The model-based approaches to thread identification are clearly useful, but what if no behavioral model exists for a system to be tested? The testing craftsperson has two choices: develop a behavioral model or resort to the system-level analogs of specification-based testing. Recall that when specification-based test cases are identified, we use information from the input and output spaces as well as the function itself. We describe system testing threads here in terms of coverage metrics that are derived from three of the basis concepts (events, ports, and data).

14.7.2.1 Event-Based Thread Testing

Consider the space of port input events. Five port input thread coverage metrics are easily defined. Attaining these levels of system test coverage requires a set of threads such that

Port Input 1: each port input event occurs
Port Input 2: common sequences of port input events occur
Port Input 3: each port input event occurs in every "relevant" data context
Port Input 4: for a given context, all "inappropriate" input events occur
Port Input 5: for a given context, all possible input events occur

The Port Input 1 metric is a bare minimum and is inadequate for most systems. Port Input 2 coverage is the most common, and it corresponds to the intuitive view of system testing because it deals with "normal use." It is difficult to quantify, however. What is a common sequence of input events? What is an uncommon one?

The last three metrics are defined in terms of a "context." The best view of a context is that it is a point of event quiescence. In the SATM system, screen displays occur at the points of event quiescence. The Port Input 3 metric deals with context-sensitive port input events. These are physical input events that have logical meanings determined by the context within which they occur. In the SATM system, for example, a keystroke on the B1 function button occurs in five separate contexts (screens displayed) and has three different meanings. The key to this metric is that it is driven by an event in all of its contexts. The Port Input 4 and Port Input 5 metrics are converses: they start with a context and seek a variety of events. The Port Input 4 metric is often used on an informal basis by testers who try to break a system. At a given context, they want to supply unanticipated input events just to see what happens. In the SATM system, for example, what happens if a function button is depressed during the PIN entry stage? The appropriate events are the digit and cancel keystrokes. The inappropriate input events are the keystrokes on the B1, B2, and B3 buttons.

This is partially a specification problem: we are discussing the difference between prescribed behavior (things that should happen) and proscribed behavior (things that should not happen). Most requirements specifications have a hard time only describing prescribed behavior; it is usually testers who find proscribed behavior. The designer who maintains my local ATM system told me that once someone inserted a fish sandwich in the deposit envelope slot. (Apparently they thought it was a waste receptacle.) At any rate, no one at the bank ever anticipated insertion of a fish sandwich as a port input event. The Port Input 4 and Port Input 5 metrics are usually very effective, but they raise one curious difficulty. How does the tester know what the expected response should be to a proscribed input? Are they simply ignored? Should there be an output warning message? Usually, this is left to the tester's intuition. If time permits, this is a powerful point of feedback to requirements specification. It is also a highly desirable focus for either rapid prototyping or executable specifications.

We can also define two coverage metrics based on port output events:

Port Output 1: each port output event occurs
Port Output 2: each port output event occurs for each cause

Port Output 1 coverage is an acceptable minimum. It is particularly effective when a system has a rich variety of output messages for error conditions. (The SATM system does not.) Port Output 2 coverage is a good goal, but it is hard to quantify. For now, note that Port Output 2 coverage refers to threads that interact with respect to a port output event. Usually, a given output event only has a small number of causes. In the SATM system, screen 10 might be displayed for three reasons: the terminal might be out of cash, it may be impossible to make a connection with the central bank to get the account balance, or the withdrawal door might be jammed. In practice, some of the most difficult faults found in field trouble reports are those in which an output occurs for an unsuspected cause. Here is one example: My local ATM system (not the SATM) has a screen that informs me that "Your daily withdrawal limit has been reached." This screen should occur when I attempt to withdraw more than $300 in one day. When I see this screen, I used to assume that my wife has made a major withdrawal (thread interaction), so I request a lesser amount. I found out that the ATM also produces this screen when the amount of cash in the dispenser is low. Instead of providing a lot of cash to the first users, the central bank prefers to provide less cash to more users.

14.7.2.2 Port-Based Thread Testing

Port-based testing is a useful complement to event-based testing. With port-based testing, we ask, for each port, what events can occur at that port. We then seek threads that exercise input ports and output ports with respect to the event lists for each port. (This presumes such event lists have been specified; some requirements specification techniques mandate such lists.) Port-based testing is particularly useful for systems in which the port devices come from external suppliers. The main reason for port-based testing can be seen in the E/R model of the basis constructs (Figure 14.3). The many-to-many relationship between devices and events should be exercised in both directions. Event-based testing covers the one-to-many relationship from events to ports, and, conversely, port-based testing covers the one-to-many relationship from ports to events. The SATM system fails us at this point—no SATM event occurs at more than one port.

14.8 Supplemental Approaches to System Testing

All model-based testing approaches have been open to the criticism that the testing is only as good as the underlying model. There is no escaping this. In response, some authorities recommend various "random" supplements. Two such techniques, mutation testing and fuzzing, are discussed in Chapter 21. In this section, we consider two fallback strategies, each of which has thread execution probability as a starting point. Both operational profiling and risk-based testing are responses to the "squeeze" on available system testing time.

14.8.1 Operational Profiles

In its most general form, Zipf's law holds that 80% of the activities occur in 20% of the space. Activities and space can be interpreted in numerous ways: people with messy desks hardly ever

use most of their desktop clutter; programmers seldom use more than 20% of the features of their favorite programming language, and Shakespeare (whose writings contain an enormous vocabulary) uses a small fraction of his vocabulary most of the time. Zipf's law applies to software (and testing) in several ways. The most useful interpretation for testers is that the space consists of all possible threads, and activities are thread executions (or traversals). Thus, for a system with many threads, 80% of the execution traverses only 20% of the threads.

Recall that a failure occurs when a fault is executed. The whole idea of testing is to execute test cases such that, when a failure occurs, the presence of a fault is revealed. We can make an important distinction: the distribution of faults in a system is only indirectly related to the reliability of the system. The simplest view of system reliability is the probability that no failure occurs during a specific time interval. (Notice that no mention is even made of faults, the number of faults, or fault density.) If the only faults are "in the corners" on threads that are seldom traversed, the overall reliability is higher than if the same number of faults were on "high-traffic" threads. The idea of operational profiles is to determine the execution frequencies of various threads and to use this information to select threads for system testing. Particularly when test time is limited (usually), operational profiles maximize the probability of finding faults by inducing failures in the most frequently traversed threads. Here we use our SATM system. In Figure 14.13, the short use case labels on the transitions in Figure 14.12 are replaced by estimated transition probabilities.

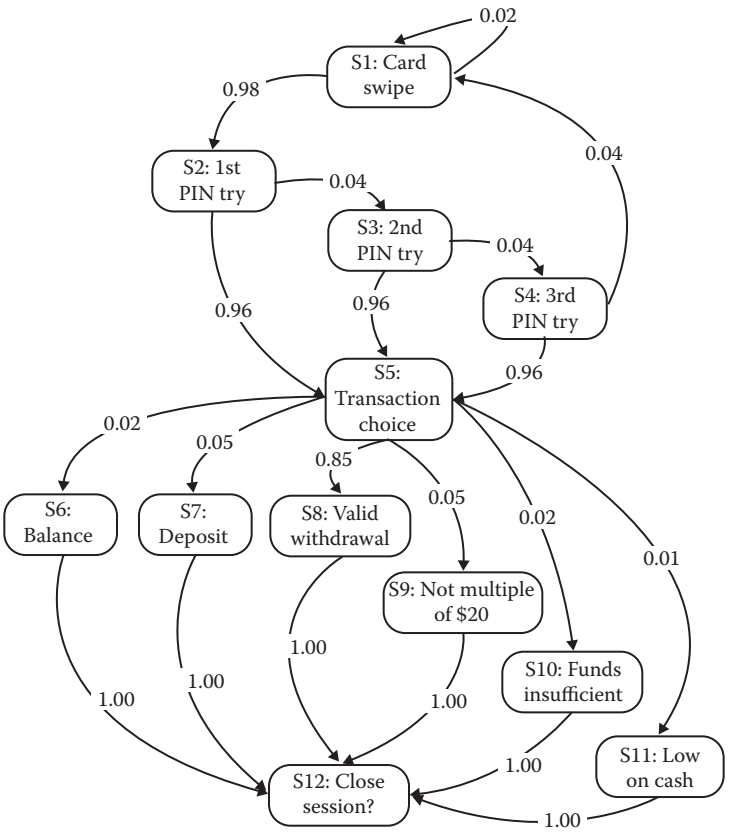

Figure 14.13 Transition probabilities in SATM finite state machine of Figure 14.12.

Finite state machines are the preferred model for identifying thread execution probabilities. The mathematics behind this is that the transition probabilities can be expressed is a "transition matrix" where the element in row *i*, column *j* is the probability of the transition from state *i* to state *j*. Powers of the transition matrix are analogous to the powers of the adjacency matrix when we discussed reachability in Chapter 4. For small systems, it is usually easier to show the transition probabilities in a spreadsheet, as in Table 14.12. Once the thread probabilities are known, they sorted according to execution probability, most to least probable. This is done in Table 14.13.

Just as the quality of model-based testing is limited by the correctness of the underlying model, the analysis of operational profiles is limited by the validity of the transition probability estimates. There are strategies to develop these estimates. One is to use historical data from similar systems. Another is to use customer-supplied estimates. Still another is to use a Delphi approach in which a group of experts give their guesses, and some average is determined. This might be based on convergence of a series of estimates, or possibly by having seven experts, and eliminating the high and low estimates.

Table 14.12 Spreadsheet of SATM Transition Probabilities

	Path	*Transition Probabilities*					*Path Probability*
First try	S1, S2, S5, S6, S12	0.999	0.96	0.02	1	1	0.019181
	S1, S2, S5, S7, S12	0.999	0.96	0.05	1	1	0.047952
	S1, S2, S5, S8, S12	0.999	0.96	0.85	1	1	0.815184
	S1, S2, S5, S9, S12	0.999	0.96	0.05	1	1	0.047952
	S1, S2, S5, S10, S12	0.999	0.96	0.02	1	1	0.019181
	S1, S2, S5, S11, S12	0.999	0.96	0.01	1	1	0.009590
Second try	S1, S2, S3, S5, S6, S12	0.999	0.04	0.96	0.02	1	0.000767
	S1, S2, S3, S5, S7, S12	0.999	0.04	0.96	0.05	1	0.001918
	S1, S2, S3, S5, S8, S12	0.999	0.04	0.96	0.85	1	0.032607
	S1, S2, S3, S5, S9, S12	0.999	0.04	0.96	0.05	1	0.001918
	S1, S2, S3, S5, S10, S12	0.999	0.04	0.96	0.02	1	0.000767
	S1, S2, S3, S5, S11, S12	0.999	0.04	0.96	0.01	1	0.000384
Third try	S1, S2, S3, S4, S5, S6, S12	0.999	0.04	0.04	0.96	0.02	0.000031
	S1, S2, S3, S4, S5, S7, S12	0.999	0.04	0.04	0.96	0.05	0.000077
	S1, S2, S3, S4, S5, S8, S12	0.999	0.04	0.04	0.96	0.85	0.001304
	S1, S2, S3, S4, S5, S9, S12	0.999	0.04	0.04	0.96	0.05	0.000077
	S1, S2, S3, S4, S5, S10, S12	0.999	0.04	0.04	0.96	0.02	0.000031
	S1, S2, S3, S4, S5, S11, S12	0.999	0.04	0.04	0.96	0.01	0.000015
Bad card	S1, S1	0.001	1	1	1	1	0.001000
PIN failed	S1, S2, S3, S1	0.999	0.04	0.04	0.04	1	0.000064

Table 14.13 SATM Operational Profile

	Path	*Transition Probabilities*					*Path Probability*
First try	S1, S2, S5, S8, S12	0.999	0.96	0.85	1	1	81.5184%
First try	S1, S2, S5, S7, S12	0.999	0.96	0.05	1	1	4.7952%
First try	S1, S2, S5, S9, S12	0.999	0.96	0.05	1	1	4.7952%
Second try	S1, S2, S3, S5, S8, S12	0.999	0.04	0.96	0.85	1	3.2607%
First try	S1, S2, S5, S6, S12	0.999	0.96	0.02	1	1	1.9181%
First try	S1, S2, S5, S10, S12	0.999	0.96	0.02	1	1	1.9181%
First try	S1, S2, S5, S11, S12	0.999	0.96	0.01	1	1	0.9590%
Second try	S1, S2, S3, S5, S7, S12	0.999	0.04	0.96	0.05	1	0.1918%
Second try	S1, S2, S3, S5, S9, S12	0.999	0.04	0.96	0.05	1	0.1918%
Third try	S1, S2, S3, S4, S5, S8, S12	0.999	0.04	0.04	0.96	0.85	0.1304%
Bad card	S1, S1	0.001	1	1	1	1	0.1000%
Second try	S1, S2, S3, S5, S6, S12	0.999	0.04	0.96	0.02	1	0.0767%
Second try	S1, S2, S3, S5, S10, S12	0.999	0.04	0.96	0.02	1	0.0767%
Second try	S1, S2, S3, S5, S11, S12	0.999	0.04	0.96	0.01	1	0.0384%
Third try	S1, S2, S3, S4, S5, S7, S12	0.999	0.04	0.04	0.96	0.05	0.0077%
Third try	S1, S2, S3, S4, S5, S9, S12	0.999	0.04	0.04	0.96	0.05	0.0077%
PIN failed	S1, S2, S3, S1	0.999	0.04	0.04	0.04	1	0.0064%
Third try	S1, S2, S3, S4, S5, S6, S12	0.999	0.04	0.04	0.96	0.02	0.0031%
Third try	S1, S2, S3, S4, S5, S10, S12	0.999	0.04	0.04	0.96	0.02	0.0031%
Third try	S1, S2, S3, S4, S5, S11, S12	0.999	0.04	0.04	0.96	0.01	0.0015%

Whatever approach is used, the final transition probabilities are still estimates. On the positive side, we could do a sensitivity analysis. In this situation, the overall ordering of probabilities is not particularly sensitive to small variations in the individual transition probabilities.

Operational profiles provide a feeling for the traffic mix of a delivered system. This is helpful for reasons other than only optimizing system testing. These profiles can also be used in conjunction with simulators to get an early indication of execution time performance and system transaction capacity.

14.8.2 Risk-Based Testing

Risk-based testing is a refinement of operational profiles. Just knowing which threads are most likely to execute might not be enough. What if a malfunction of a somewhat obscure thread were to be extremely costly? The cost might be in terms of legal penalties, loss of revenue, or difficulty of repair. The basic definition of risk is

Risk = cost * (probability of occurrence)

Since operational profiles give the (estimate of) probability of occurrence, we only need to make an estimate of the cost factor.

Hans Schaefer, a consultant who specializes in risk-based testing, advises that the first step is to group the system into risk categories. He advises four risk categories: Catastrophic, Damaging, Hindering, and Annoying (Schaefer, 2005). Next, the cost weighting is assessed. He suggests a logarithmic weighting: 1 for low cost of failure, 3 for medium, and 10 for high. Why logarithmic? Psychologists are moving in this direction because subjects who are asked to rank factors on linear scales, for example, 1 for low and 5 for high, do not make enough of a distinction in what is usually a subjective assessment. Table 14.14 is the result of this process for our SATM use cases in Table 14.13. In this assessment, failure of a deposit was the most severe.

Table 14.14 SATM Risk Assessment

Use Case Description	*Use Case Probability*	*Cost of Failure*	*Risk*
First try, normal withdrawal	81.5184%	3	2.4456
First try, deposit	4.7952%	10	0.4795
First try, withdrawal but insufficient funds	1.9181%	10	0.1918
First try, withdrawal not multiple of $20	4.7952%	3	0.1439
Second try, normal withdrawal	3.2607%	3	0.0978
First try, withdrawal, ATM low on cash	0.9590%	10	0.0959
First try, balance inquiry	1.9181%	1	0.0192
Second try, deposit	0.1918%	10	0.0192
Insertion of invalid ATM card	0.1000%	10	0.0100
Second try, withdrawal insufficient funds	0.0767%	10	0.0077
Second try, withdrawal not multiple of $20	0.1918%	3	0.0058
Third try, normal withdrawal	0.1304%	3	0.0039
Second try, withdrawal, ATM low on cash	0.0384%	10	0.0038
Second try, balance inquiry	0.0767%	1	0.0008
Third try, deposit	0.0077%	10	0.0008
Third try, withdrawal but insufficient funds	0.0031%	10	0.0003
Third try, withdrawal not multiple of $20	0.0077%	3	0.0002
PIN entry failed after third attempt	0.0064%	3	0.0002
Third try, withdrawal, ATM low on cash	0.0015%	10	0.0002
Third try, balance inquiry	0.0031%	1	0.0000

Schaefer's risk categories applied to the SATM use cases are given below. Deposit failures are seen as most severe because a customer may depend on a deposit being made to cover other checks. Balance inquiries are least severe because a malfunction is only inconvenient.

Catastrophic:	deposits, invalid withdrawals
Damaging:	normal withdrawals
Hindering:	invalid ATM card, PIN entry failure
Annoying:	balance inquiries

The risk-ordered SATM use cases in Table 4.14 differ slightly from their operational profile in Table 14.13. The normal withdrawal after a successful PIN entry on the first try still heads the list, mostly due to its high probability.

As a refinement, Schaefer suggests assigning several attributes to a use case and giving these attributes weighting values. For our SATM system, we might consider factors such as customer convenience, bank security, and identity theft.

14.9 Nonfunctional System Testing

The system testing ideas thus far discussed have been based on specification-based, or behavioral, requirements. Functional requirements are absolutely in the *does view*, as they describe what a system does (or should do). To generalize, nonfunctional testing refers to how well a system performs its functional requirements. Many nonfunctional requirements are categorized onto "-abilities": reliability, maintainability, scalability, usability, compatibility, and so on. While many practitioners have clear ideas on the meaning of the -abilities in their product domains, there is not much standardization of either the terms or the techniques. Here we consider the most common form of nonfunctional testing—stress testing.

14.9.1 Stress Testing Strategies

Synonymously called performance testing, capacity testing, or load testing, this is the most common, and maybe the most important form of nonfunctional testing. Because stress testing is so closely related to the nature of the system being tested, stress testing techniques are also application dependent. Here we describe three common strategies, and illustrate them with examples.

14.9.1.1 Compression

Consider the performance of a system in the presence of extreme loads. A web-based application may be very popular, and its server might not have the capacity. Telephone switching systems use the term Busy Hour Call Attempts (BHCAs) to refer to such offered traffic loads. The strategy in those systems is best understood as compression.

A local switching system must recognize when a subscriber originates a call. Other than sensing a change in subscriber line status from idle to active, the main indicator of a call attempt is the entry of digits. Although some dial telephones still exist, most subscribers use digit keys. The technical term is Dual Tone Multifrequency (DTMF) tones, as the usual 3×4 array of digit keys has three frequencies for the columns and four frequencies for the rows of digit keypads. Each digit is therefore represented by two frequency tones, hence the name. The local switching system must convert the tones to a digital form, and this is done with a DTMF receiver.

Here is a hypothetical example, with simplified numbers, to help understand the compression strategy. Suppose a local switching system must support 50,000 BHCAs. To do so, the system might have 5000 DTMF receivers. To test this traffic load, somehow 50,000 call originations must be generated in 60 minutes. The whole idea of compression strategies it to reduce these numbers to more manageable sizes. If a prototype only has 50 DTMF receivers, the load testing would need to generate 500 call attempts.

This pattern of compressing some form of traffic and associated devices to handle the offered traffic occurs in many application domains, hence the general term, traffic engineering.

14.9.1.2 Replication

Some nonfunctional requirements may be unusually difficult to actually perform. Many times, actual performance would destroy the system being tested (destructive vs. nondestructive testing). There was a Calvin and Hobbes comic strip that succinctly explained this form of testing. In the first frame, Calvin sees a sign on a bridge that says "Maximum weight 5 tons." He asks his father how this is determined. The father answers that successively heavier trucks are driven over the bridge until the bridge collapses. In the last frame, Calvin has his standard shock/horror expression. Rather than destroy a system, some form of replication can be tried. Two examples follow.

One of the nonfunctional requirements for an army field telephone switching center was that it had to be operational after a parachute drop. Actually doing this was both very expensive and logistically complex. None of the system testers knew how to replicate this; however, in consultation with a former paratrooper, the testers learned that the impact of a parachute drop is similar to jumping off a ten-foot (three meter) wall. The testers put a prototype on a fork lift skid, lifted it to a height of ten feet, and tilted it forward until it fell off the skid. After hitting the ground, the prototype was still operational, and the test passed.

One of the most dangerous incidents for aircraft is a mid-air collision with a bird. Here is an excerpt of a nonfunctional requirement for the F 35 jet aircraft built by Lockheed Martin (Owens et al., 2009).

> *The Canopy System Must Withstand Impact of a 4 lb Bird at 480 Knots on the Reinforced Windscreen and 350 Knots on the Canopy Crown Without:*
>
> - *Breaking or Deflecting so as to Strike the Pilot When Seated in the Design Eye "High" Position,*
> - *Damage To The Canopy That Would Cause Incapacitating Injury To The Pilot, or*
> - *Damage That Would Preclude Safe Operation of, or Emergency Egress from the Aircraft*

Clearly it would be impossible to arrange a mid-air bird collision, so the Lockheed Martin testers replicated the problem with an elaborate cannon that would shoot a dead chicken at the windscreen and at the canopy. The tests passed.

There is an urban legend, debunked on Snopes.com, about follow-up on a British (or French, or fill-in-your-favorite-country) firm that used the same idea for canopy testing, but their tests all failed. When they asked the US testers why the failures were so consistent, they received a wry answer: "you need to thaw the chicken first." Why mention this? If nonfunctional testing is done with a replication strategy, it is important to replicate, as closely as possible, the actual test scenario. (But it is funny.)

14.9.2 Mathematical Approaches

In some cases, nonfunctional testing cannot be done either directly, indirectly, or with commercial tools. There are three forms of analysis that might help—queuing theory, reliability models, and simulation.

14.9.2.1 Queuing Theory

Queuing theory deals with servers and queues of tasks that use the service. The mathematics behind queuing theory deals with task arrival rates, and service times, as well as the number of queues and the number of servers. In everyday life, we see examples of queuing situations: checkout lines in a grocery store, lines to buy tickets at a movie theater, or lift lines at a ski area. Some settings, for example, a local post office, uses a single queue of patrons waiting for service at one of several clerk positions. This happens to be the most efficient queuing discipline—single queue, multiple server. Service times represent some form of system capacity, and queues represent traffic (transactions) offered to the system.

14.9.2.2 Reliability Models

Reliability models are somewhat related to queuing theory. Reliability deals with failure rates of components and computes characteristics such as likelihood of system failure, mean time to failure (MTTF), mean time between failures (MTBF), and mean time to repair (MTTR). Given actual or assumed failure rates of system components, these quantities can be computed.

A telephone switching system has a reliability requirement of not more than two hours of downtime in 40 years of continuous operation. This is an availability of 0.99999429, or stated negatively, failure rate of 5.7×10^{-6}, (0.0000057). How can this be guaranteed? Reliability models are the first choice. They can be expressed as tree diagrams or as directed graphs, very similar to the approach used to compute an operational profile. These models are based on failure rates of individual system components that are linked together physically, and abstractly in the reliability model.

A digital end office intended for the rural U.S. market had to be certified by an agency of the U.S. government, the Rural Electric Administration (REA). That body followed a compression strategy, and required an on-site test for six months. If the system functioned with less than 30 minutes of downtime, it was certified. A few months into the test interval, the system had less than two minutes of downtime. Then a tornado hit the town and destroyed the building that contained the system. The REA declared the test to be a failure. Only extreme pleading resulted in a retest. The second time, there was less than 30 seconds of downtime in the six-month interval.

Reliability models have a solid history of applicability to physical systems, but can they be applied to software? Physical components can age, and therefore deteriorate. This is usually shown in the Weibull distribution, in which failures drop to nearly zero rapidly. Some forms show an increase after an interval that represents the useful life of a component. The problem is that software, once well tested, does not deteriorate. The main difference between reliability models applied to software versus hardware comes down to the arrival rate of failures. Testing based on operational profiles, and the extension to risk-based testing is a good start; however, no amount of testing can guarantee the absence of software faults.

14.9.2.3 Monte Carlo Testing

Monte Carlo testing might be considered a last resort in the system tester's arsenal. The basic idea of Monte Carlo testing is to randomly generate a large number of threads (transactions) and then

see if anything unexpected happens. The Monte Carlo part comes from the use of pseudorandom numbers, not from the fact that the whole approach is a gamble. Monte Carlo testing has been successful in applications where computations involving physical (as opposed to logical, see Chapter 6) variables are used. The major drawback to Monte Carlo testing is that the large number of random transactions requires a similarly large number of expected outputs in order to determine whether a random test case passes or fails.

14.10 Atomic System Function Testing Example

We can illustrate ASF testing on our integrationNextDate pseudocode. This version differs slightly from that in Chapter 13. A few output statements were added to make ASF identification more visible.

```
1   Main integrationNextDate      'start program event occurs here
       Type   Date
              Month As Integer
              Day As Integer
              Year As Integer
       EndType
       Dim today As Date
       Dim tomorrow As Date
2      Output("Welcome to NextDate!")
3      GetDate(today)                                  'msg1
4      PrintDate(today)                                'msg2
5      tomorrow = IncrementDate(today)                 'msg3
6      PrintDate(tomorrow)                             'msg4
7   End Main

8   Function isLeap(year) Boolean
9      If (year divisible by 4)
10          Then
11             If (year is NOT divisible by 100)
12                   Then isLeap = True
13                   Else
14                       If (year is divisible by 400)
15                             Then isLeap = True
16                             Else isLeap = False
17                       EndIf
18             EndIf
19     Else isLeap = False
20     EndIf
21  End (Function isLeap)

22  Function lastDayOfMonth(month, year) Integer
23     Case month Of
24         Case 1: 1, 3, 5, 7, 8, 10, 12
25             lastDayOfMonth = 31
26         Case 2: 4, 6, 9, 11
27             lastDayOfMonth = 30
28         Case 3: 2
```

```
29              If (isLeap(year))                        'msg5
30                    Then lastDayOfMonth = 29
31                    Else lastDayOfMonth = 28
32              EndIf
33     EndCase
34  End (Function lastDayOfMonth)

35  Function GetDate(aDate) Date
       dim aDate As Date

36      Function ValidDate(aDate)   Boolean 'within scope of GetDate
           dim aDate As Date
           dim dayOK, monthOK, yearOK As Boolean
37         If ((aDate.Month > 0) AND (aDate.Month < = 12)
38               Then   monthOK = True
39                      Output("Month OK")
40               Else   monthOK = False
41                      Output("Month out of range")
42         EndIf
43         If (monthOK)
44               Then
45               If ((aDate.Day > 0) AND (aDate.Day < =
                       lastDayOfMonth(aDate.Month, aDate.Year))  'msg6
46                     Then   dayOK = True
47                            Output("Day OK")
48                     Else   dayOK = False
49                            Output("Day out of range")
50               EndIf
51         EndIf
52         If ((aDate.Year > 1811) AND (aDate.Year < = 2012)
53               Then   yearOK = True
54                      Output("Year OK")
55               Else   yearOK = False
56                      Output("Year out of range")
57         EndIf
58         If (monthOK AND dayOK AND yearOK)
59               Then ValidDate = True
60                      Output("Date OK")
61               Else ValidDate = False
62                      Output("Please enter a valid date")
63         EndIf
64      End (Function ValidDate)

    'GetDate body begins here
65         Do
66               Output("enter a month")
67               Input(aDate.Month)
68               Output("enter a day")
69               Input(aDate.Day)
70               Output("enter a year")
71               Input(aDate.Year)
72               GetDate.Month = aDate.Month
73               GetDate.Day = aDate.Day
74               GetDate.Year = aDate.Year
75         Until (ValidDate(aDate))                              'msg7
76  End (Function GetDate)
```

```
77  Function IncrementDate(aDate) Date
78              If (aDate.Day < lastDayOfMonth(aDate.Month))    'msg8
79                   Then aDate.Day = aDate.Day + 1
80                   Else aDate.Day = 1
81                        If (aDate.Month = 12)
82                             Then aDate.Month = 1
83                                  aDate.Year = aDate.Year + 1
84                             Else aDate.Month = aDate.Month + 1
85                        EndIf
86              EndIf
87  End (IncrementDate)

88  Procedure PrintDate(aDate)
89        Output("Day is ", aDate.Month, "/", aDate.Day, "/", aDate.Year)
90  End (PrintDate)
```

14.10.1 Identifying Input and Output Events

Recall that an ASF begins with a port input event, conducts some processing, and, depending on the project-chosen granularity, ends with one or more port output events. We can identify ASFs from source code by locating the nodes at which port inputs and outputs occur. Table 14.15 lists the port input and output events in integrationNextDate and the corresponding node numbers.

Table 14.15 Location of Port Events in integrationNextDate

Input Events	*Node*	*Output Event Description*	*Node*
e0: Start program event	1	e7: Welcome message	2
e1: Center a valid month	67	e8: Print today's date	4
e2: Enter an invalid month	67	e9: Print tomorrow's date	6
e3: Enter a valid day	69	e10: "Month OK"	39
e4: Enter an invalid day	69	e11: "Month out of range"	41
e5: Enter a valid year	71	e12: "Day OK"	47
e6: Enter an invalid year	71	e13: "Day out of range"	49
		e14: "Year OK"	54
		e15: "Year out of range"	56
		e16: "Date OK"	60
		e17: "Please enter a valid date"	62
		e18: "Enter a month"	66
		e19: "Enter a day"	68
		e20: "Enter a year"	70
		e21: "Day is month, day, year"	89

14.10.2 Identifying Atomic System Functions

The next step is to identify the ASFs in terms of the port input and output events. Table 14.16 contains the first attempt at ASF identification. There is a subtle problem with this first set of ASFs. They presume that the states in the ASF graph in Figure 14.14 are independent, but they are not. As we have seen, dependent nodes often lead to impossible paths.

The ASF graph shows all the ways both valid months, days, and years can be entered, but the transition to ASF-8 (or to ASF-9) depends on the history of previous ASFs. Since FSMs have no memory, these transitions are necessarily undefined.

Table 14.16 First Attempt at ASFs

Atomic System Function	*Inputs*	*Outputs*
ASF-1: start program	e0	e7
ASF-2: enter a valid month	e1	e10
ASF-3: enter an invalid month	e2	e11
ASF-4: enter a valid day	e3	e12
ASF-5: enter an invalid day	e4	e13
ASF-6: enter a valid year	e5	e14
ASF-7: enter an invalid year	e6	e15
ASF-8: print for valid input		
ASF-9: print for invalid input		

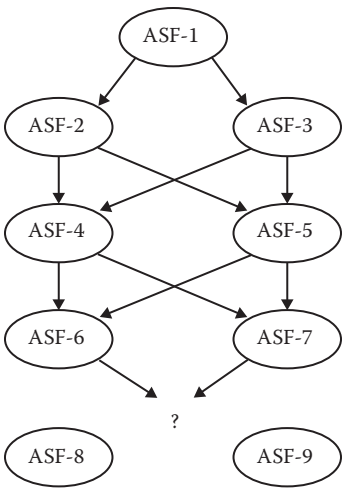

Figure 14.14 Directed graph of Atomic System Functions for integrationNextDate.

14.10.3 Revised Atomic System Functions

The Do Until loop from nodes 65 to 75 allows for many mistakes. The checking for valid month, day, and year values is linear, but all three must be correct to terminate the Do Until loop. Since we can make any number of input variable mistakes, in any order, there is no way to represent in an ASF graph when the Do Until loop will terminate. Incidentally, this is case of the Anna Karenina principle (see Diamond, 1997). The principle refers to situations where a conjunction of criteria must all be true; any one becoming false negates the whole situation. It comes from the first sentence in Leo Tolstoy's famous Russian novel, *Anna Karenina*: "Happy families are all alike; every unhappy family is unhappy in its own way."

The second attempt (see Table 14.17) postulates larger ASFs (maybe they are "molecular"?) that have pluralities of port inputs and port outputs.

Now each ASF is the entry of a triple (month, day, year). ASFs 2, 3, and 4 make one mistake at a time, and ASF-5 gets all the values right. Would we really need the last four (two or three mistakes at a time)? Probably not. That kind of testing should have been done at the unit level. Figure 14.15 is the new ASF graph for the first five of the "revised" ASFs.

Table 14.17 Second Attempt at ASFs

Atomic System Function	*Inputs*	*Outputs*
ASF-1: start program	e0	e7
ASF-2: enter a date with an invalid month, valid day, and valid year	e2, e3, e5	e11, e12, e14, e17
ASF-3: enter a date with an invalid day, valid month, and valid year	e1, e4, e5	e10, e13, e14, e17
ASF-4: enter a date with an invalid year, valid day, and valid month	e1, e3, e6	e10, e12, e15, e17
ASF-5: enter a date with valid month, day, and year	e1, e3, e5	e10, e12, e14, e16, e21
ASF-6: enter a date with valid month, invalid day, and invalid year	e1, e4, e6	e10, e13, e15, e17
ASF-7: enter a date with valid day, invalid month, and invalid year	e2, e3, e6	e11, e12, e15, e17
ASF-8: enter a date with valid year, invalid day, and invalid month	e5, e4, 46	e14, e13, e15, e17
ASF-9: enter a date with invalid month, invalid day, and invalid year	e2, e4, e6	e11, e13, e15, e17

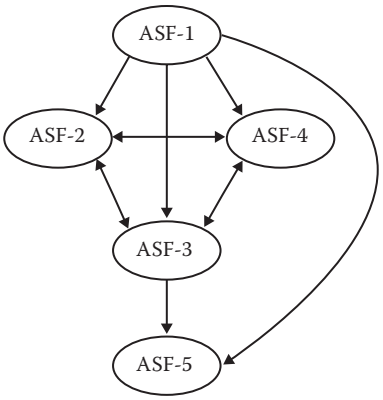

Figure 14.15 Directed graph of five Atomic System Functions for integrationNextDate.

EXERCISES

1. One of the problems of system testing, particularly with interactive systems, is to anticipate all the strange things the user might do. What happens in the SATM system if a customer enters three digits of a PIN and then leaves?
2. To remain "in control" of abnormal user behavior (the behavior is abnormal, not the user), the SATM system might introduce a timer with a 30-second time-out. When no port input event occurs for 30 seconds, the SATM system could ask if the user needs more time. The user can answer yes or no. Devise a new screen and identify port events that would implement such a time-out event.
3. Suppose you add the time-out feature described in Exercise 2 to the SATM system. What regression testing would you perform?
4. Make an additional refinement to the PIN try finite state machine (Figure 14.6) to implement your time-out mechanism from Exercise 2, then revise the thread test case in Table 14.3.
5. Develop an incidence matrix similar to Table 14.10 for the PIN entry short use cases in Table 14.8.
6. Does it make sense to use test coverage metrics in conjunction with operational profiles? Same question for risk-based testing. Discuss this.

For questions 7 through 9, revisit the description of the Garage Door Controller in Chapter 2 and the corresponding finite state machine in Chapter 4.

7. Develop extended essential use cases for the Garage Door Controller.
8. The input and output events for the Garage Door Controller are shown in Figure 4.6. Use these as starting points for incidence matrices of your use cases with respect to input events, output events, and all events.
9. Develop Event-Driven Petri Nets for the Garage Door Controller.

References

Diamond, J., *Guns, Germs, and Steel,* W. W. Norton, New York, 1997.

Jorgensen, P.C., System testing with pseudo-structures, *American Programmer*, Vol. 7, No. 4, April 1994, pp. 29–34.

Jorgensen, P.C., *Modeling Software Behavior: A Craftsman's Approach*, CRC Press, New York, 2009.

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 2001.

Schaefer, H., Risk based testing, strategies for prioritizing tests against deadlines, *Software Test Consulting*, http://home.c2i.net/schaefer/testing.html, 2005.

Owens, S.D., Caldwell, E.O. and Woodward, M.R., Birdstrike certification tests of F-35 canopy and airframe structure, *2009 Aircraft Structural Integrity Program (ASIP) Conference*, Jacksonville, FL, December 2009, also can be found at Trimble, S., July 28, 2010, http://www.flightglobal.com/blogs/the-dewline/2010/07/video-f-35-birdstrike-test-via.html and http://www.flightglobal.com/blogs/the-dewline/Birdstrike%20Impact%20Studies.pdf.