# Chapter 16

# Issues in Object-Oriented Testing

Both theoretical and practical work on the testing of object-oriented software flourished since the second half of the 1990s. The conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and vendor-sponsored testing conferences have consistently featured both papers and tutorials on object-oriented testing. One Web site boasts over 18,000 links to object-oriented testing literature (www.cetus-links.org/oo_testing.html). The beginning of the 21st century is marked, among other things, by increased consensus on questions such as: What is an object-oriented unit? How are object-oriented applications best modeled? One of the original hopes for object-oriented software was that objects could be reused without modification or additional testing. This was based on the assumption that well-conceived objects encapsulate functions and data "that belong together," and once such objects are developed and tested, they become reusable components. The interest in aspect-oriented programming (Kiczales, 1997) is one response to some of the limitations of the object-oriented paradigm. The new consensus is that there is little reason for this optimism — object-oriented software has potentially more severe testing problems than does traditional software. On the positive side, the Unified Modeling Language (UML) has emerged as a strong unifying (and driving) force for several aspects of object-oriented technology.

Our goal in this chapter is to identify the testing issues raised by object-oriented software. First, we have the question of levels of testing; this, in turn, requires clarification of object-oriented units. Next, we consider some of the implications of the strategy of composition (as opposed to functional decomposition). Object-oriented software is characterized by inheritance, encapsulation, and polymorphism; therefore, we look at ways that traditional testing can be extended to address the implications of these issues. In the remaining chapters of Part V, we examine class testing, graphical user interface (GUI) testing, integration and system testing, UML-based testing, and the application of dataflow testing to object-oriented software. We will do this using the object-oriented calendar and currency converter examples.

## 16.1 Units for Object-Oriented Testing

Traditional software has a variety of definitions for *unit*. Two that pertain to object-oriented testing are:

A unit is the smallest software component that can be compiled and executed.
A unit is a software component that would never be assigned to more than one designer to develop.

These definitions can be contradictory. Certain industrial applications have huge classes; these clearly violate the one-designer, one-class definition. In such applications, it seems better to define an object-oriented unit as the work of one person, which likely ends up as a subset of the class operations. In an extreme case, an object-oriented unit might be a subclass of a class that contains only the attributes needed by a single operation or method. (In Part V, we will use *operation* to refer to the definition of a class function and *method* to refer to its implementation.) For such units, object-oriented unit testing reduces to traditional testing. This is a nice simplification but somewhat problematic, because it shifts much of the object-oriented testing burden onto integration testing. Also, it throws out the gains made by encapsulation.

The class-as-unit choice has several advantages. In a UML context, a class has an associated StateChart that describes its behavior. Later, we shall see that this is extremely helpful in test case identification. A second advantage is that object-oriented integration testing has clearer goals, namely, to check the cooperation of separately tested classes, which echoes traditional software testing.

## 16.2 Implications of Composition and Encapsulation

Composition (as opposed to decomposition) is the central design strategy in object-oriented software development. Together with the goal of reuse, composition creates the need for very strong unit testing. Because a unit (class) may be composed with previously unknown other units, the traditional notions of coupling and cohesion are applicable. Encapsulation has the potential to resolve this concern, but only if the units (classes) are highly cohesive and very loosely coupled. The main implication of composition is that, even presuming very good unit-level testing, the real burden is at the integration testing level.

Some of this is clarified by example. Suppose we revisit the Saturn windshield wiper system from an object-oriented viewpoint. We would most likely identify three classes — lever, wiper, and dial; their behavior is shown in Figure 16.1.

One choice of pseudocode for the interfaces of these classes is as follows:

```
Class lever(leverPosition;
            private senseLeverUp(),
            private senseLeverDown() )
Class dial(dialPosition;
            private senseDialUp(),
            private senseDialDown() )
Class wiper(wiperSpeed;
            setWiperSpeed(newSpeed) )
```

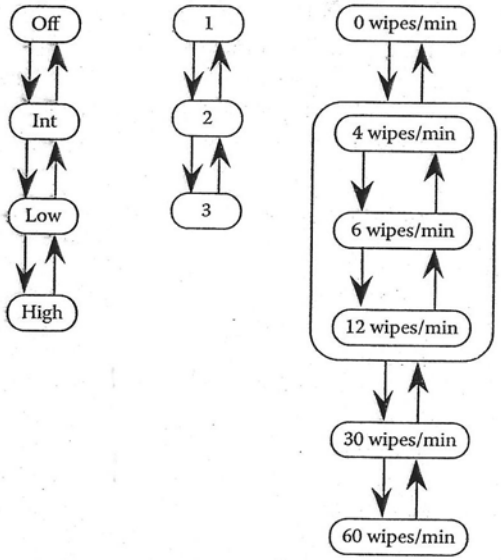



**Figure 16.1 Behavior of windshield wiper classes.**

The lever and dial classes have operations that sense physical events on their respective devices. When these methods (corresponding to the operations) execute, they report their respective device positions to the wiper class. The interesting part of the windshield wiper example is that the lever and the dial are independent devices, and they interact when the lever is in the INT (intermittent) position. The question raised by encapsulation is: Where should this interaction be controlled?

The precept of encapsulation requires that classes only know about themselves and operate on their own. Thus, the lever does not know the dial position, and the dial does not know the lever position. The problem is that the wiper needs to know both positions. One possibility, as shown in the previous interface, is that the lever and dial always report their positions, and the wiper figures out what it must do. With this choice, the wiper class becomes the "main program" and contains the basic logic of the whole system.

Another choice might be to make the lever class the "smart" object because it knows when it is in the intermittent (INT) state. With this choice, when the response to a lever event puts the lever in the INT state, a method gets the dial status (with a getDialPosition message) and simply tells the wiper what speed is needed. With this choice, the three classes are more tightly coupled and, as a result, less reusable. Another problem occurs with this choice. What happens if the lever is in the INT position and a subsequent dial event occurs? There would be no reason for the lever to get the new dial position, and no message would be sent to the wiper class.

A third choice might be to make the wiper the main program (as in the first choice), but use a Has relation to the lever and dial classes. With this choice, the wiper class uses the sense operations of the lever and dial classes to detect physical events. This forces the wiper class to be continuously active, in a polling sense, so that asynchronous events at the lever and dial can be observed.

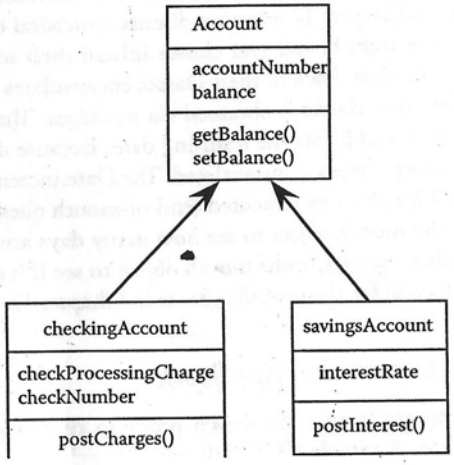
Consider these three choices from the standpoint of composition and encapsulation. The first

a cheaper windshield wiper might omit the dial altogether, and an expensive windshield wiper might replace the three-position dial with a "continuous" dial. Similar changes might be made to the lever. In the other two choices, the increased coupling among the classes reduces their ability to be composed. This is our conclusion: doing a good job at encapsulation results in classes that can more easily be composed (and thus reused) and tested.

## 16.3 Implications of Inheritance

Although the choice of classes as units seems natural, the role of inheritance complicates this choice. If a given class inherits attributes or operations from super classes, the stand-alone compilation criterion of a unit is sacrificed. Binder (1996) suggests "flattened classes" as an answer. A flattened class is an original class expanded to include all the attributes and operations it inherits. Flattened classes are mildly analogous to the fully flattened dataflow diagrams of structured analysis. (Notice that flattened classes are complicated by multiple inheritance, and really complicated by selective and multiple selective inheritance.) Unit testing on a flattened class solves the inheritance problem, but it raises another. A flattened class will not be part of a final system, so some uncertainty remains. Also, the methods in a flattened class might not be sufficient to test the class. The next workaround is to add special-purpose test methods. This facilitates class-as-unit testing but raises a final problem: a class with test methods is not (or should not be) part of the delivered system. This is perfectly analogous to the question of testing original or instrumented code in traditional software. Some ambiguity is also introduced: the test methods can also be faulty. What if a test method falsely reports a fault, or worse, incorrectly reports success? Test methods are subject to the same false-positive and false-negative outcomes as medical experiments. This leads to an unending chain of methods testing other methods, very much like the attempt to provide external proofs of consistency of a formal system.

Figure 16.2 shows a UML inheritance diagram of a part of our earlier simple automated teller machine (SATM) system; some functionality has been added to make this a better example. Both checking and savings accounts have account numbers and balances, and these can be accessed



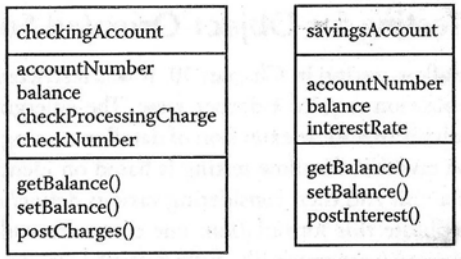




**Figure 16.3 Flattened checkingAccount and savingsAccount classes.**

and changed. Checking accounts have a per-check processing charge that must be deducted from the account balance. Savings accounts draw interest that must be calculated and posted on some periodic basis.

If we did not "flatten" the checkingAccount and savingsAccount classes, we would not have access to the balance attributes, and we would not be able to access or change the balances. This is clearly unacceptable for unit testing. Figure 16.3 shows the flattened checkingAccount and savingsAccount classes. These are clearly stand-alone units that are sensible to test. Solving one problem raises another: with this formulation, we would test the getBalance and setBalance operations twice, thereby losing some of the hoped-for economies of object orientation.

## 16.4 Implications of Polymorphism

The essence of polymorphism is that the same method applies to different objects. Considering classes as units implies that any issues of polymorphism will be covered by the class/unit testing. Again, the redundancy of testing polymorphic operations sacrifices hoped-for economies.

## 16.5 Levels of Object-Oriented Testing

Three or four levels of object-oriented testing are used, depending on the choice of what constitutes a unit. If individual operations or methods are considered to be units, we have four levels: operation/method, class, integration, and system testing. With this choice, operation/method testing is identical to unit testing of procedural software. Class and integration testing can be well renamed intraclass and interclass testing. The second level, then, consists of testing interactions among previously tested operations/methods. Integration testing, which we will see is the major issue of object-oriented testing, must be concerned with testing interactions among previously tested classes. Finally, system testing is conducted at the port event level and is (or should be) identical to system testing of traditional software. The only difference is where system-level test cases originate.

## 16.6 GUI Testing

Because graphical user interfaces have become so closely associated with object-oriented software, we will devote a separate chapter to those testing issues. GUIs are a special case of event-driven systems, and event-driven systems are vulnerable to the problem of an infinite number of event

## 16.7 Dataflow Testing for Object-Oriented Software

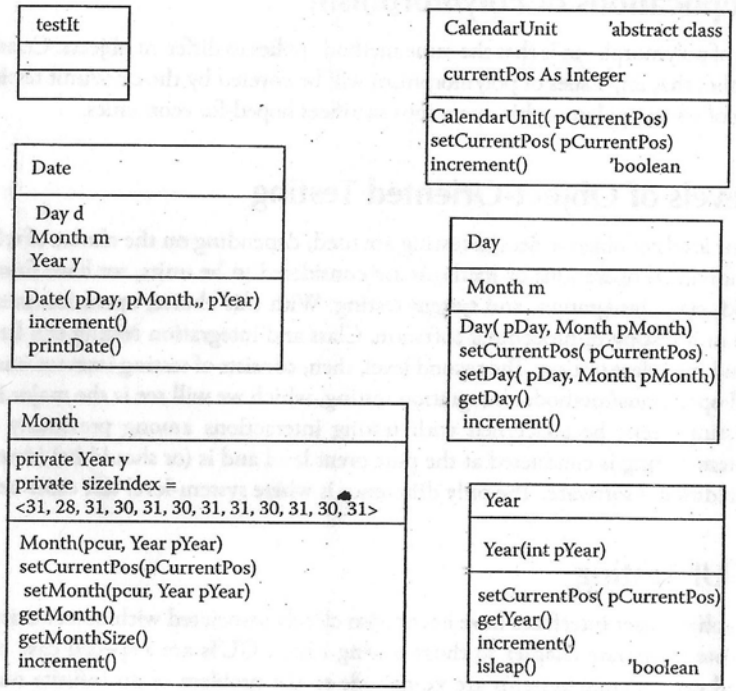
When we considered dataflow testing in Chapter 10, it was restricted to a single unit. The issues of inheritance and composition require a deeper view. The emerging consensus in the object-oriented testing community is that some extension of dataflow testing should address these special needs. In Chapter 10, we saw that dataflow testing is based on identifying define and use nodes in the program graph of a unit and then considering various define/use paths. Procedure calls in traditional software complicate this formulation; one common workaround is to embed called procedures into the unit tested (very much like fully flattened classes). Chapter 18 presents a revision of event-driven Petri nets that exactly describes dataflow among object-oriented operations. Within this formulation, we can express the object-oriented extension of dataflow testing.

## 16.8 Examples for Part V

We will use two primary examples for our discussion of object-oriented testing. The first is an object-oriented implementation of our now-familiar NextDate problem; the second is typical of GUI applications — a currency conversion application.

### 16.8.1 The Object-Oriented Calendar

The o-oCalendar program is an object-oriented implementation of the NextDate example given in Chapter 2. Figure 16.4 and Figure 16.5 show the UML class diagrams and the inheritance and



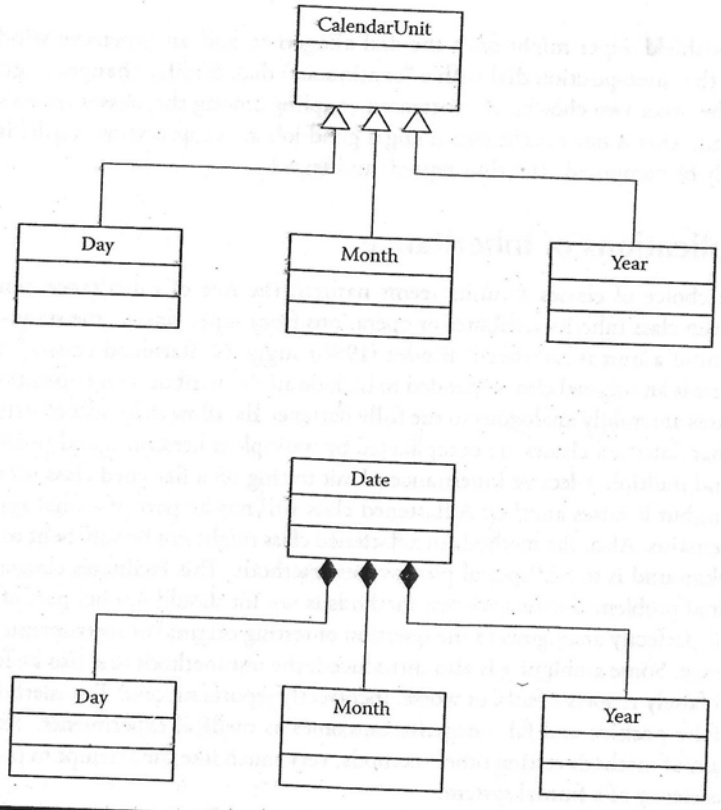




**Figure 16.5 Class inheritance and aggregation.**

aggregation of classes in the o-oCalendar problem. The letters and numbers on statement lines in the pseudocode will be used in Chapter 17 when we discuss structural testing for o-o software.

Object instances of the day, month, and year classes inherit their set and increment methods from the abstract CalendarUnit class. Each of these classes encapsulates exactly the information it needs; any information from other classes is obtained via messages. The testIt class instantiates a test date and then increments it and prints the resulting date. Because date objects are composed of month, day, and year instances, these are instantiated. The Date.increment method sends a message to the day object to see if it can be incremented (end-of-month question). The Day.increment method sends a message to the month object to see how many days are in the month. Next, the Date.increment method sends a message to the month object to see if it can be incremented (end-of-year question). The pseudocode for the o-oCalendar is in Chapter 17.

### 16.8.2 The Currency Conversion Application

The currency conversion program is an event-driven program that emphasizes code associated with a graphical user interface. A sample GUI built with

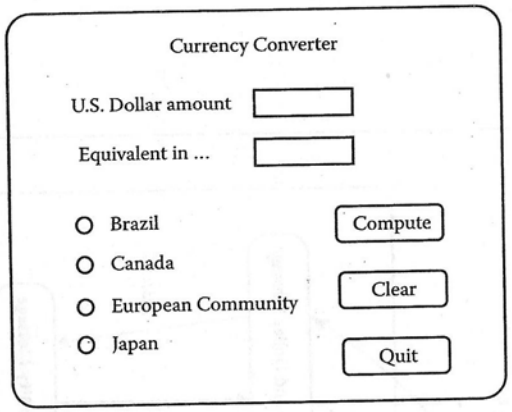



**Figure 16.6 Currency converter GUI.**

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

To test a GUI application, begin by identifying all the user input events and all the system output events (these must be externally visible and observable). These are listed in Table 16.1a and b for the currency conversion program.

**Table 16.1a Input Events for the Currency Conversion Program**

| | *Input Events* | | *Input Events* |
|---|---|---|---|
| ip1 | Enter U.S. dollar amount | ip2.4 | Click on Japan |
| ip2 | Click on a country button | ip3 | Click on Compute button |
| ip2.1 | Click on Brazil | ip4 | Click on Clear button |
| ip2.2 | Click on Canada | ip5 | Click on Quit button |
| ip2.3 | Click on European Community | ip6 | Click on OK in error message |

**Table 16.1b Output Events for the Currency Conversion Program**

| | *Output Events* | | *Output Events* |
|---|---|---|---|
| op1 | Display U.S. dollar amount | op4 | Reset selected country |
| op2 | Display currency name | op4.1 | Reset Brazil |
| op2.1 | Display Brazilian reals | op4.2 | Reset Canada |
| op2.2 | Display Canadian dollars | op4.3 | Reset European Community |
| op2.3 | Display European Community euros | op4.4 | Reset Japan |
| op2.4 | Display Japanese yen | op5 | Display foreign currency value |
| op2.5 | Display ellipsis | op6 | Error message: Must select a country |
| op3 | Indicate selected country | op7 | Error message: Must enter U.S. dollar amount |
| op3.1 | Indicate Brazil | op8 | Error message: Must select a country and enter U.S. dollar amount |
| op3.2 | Indicate Canada | op9 | Reset U.S. dollar amount |
| op3.3 | Indicate European Community | op10 | Reset equivalent currency amount |
| op3.4 | Indicate Japan | | |

Figure 16.7 contains a high-level, mostly complete view of the application. States are externally visible appearances of the GUI. The idle state, for example, occurs when the application is started and no user input has occurred (see Figure 16.6). It also occurs after a click event on the Clear button. The U.S. dollar amount entered state occurs when the user has entered a dollar amount and has done nothing else. The other states are similarly named. Notice that the country selected state is really a macro-state that refers to one of the four countries selected. This state is shown in more detail in Figure 16.8. The annotations on the transitions refer to the abbreviations of the input and output events listed in Tables 16.1a and b. Even this little GUI is surprisingly complex. We can suppress some of the complexity by using artificial input events such as ip2: click on a country button. Notice that the artificial input event ip2 and the artificial output events op2, op3, and op4 greatly simplify the high-level finite state machine (FSM) in Figure 16.7. Several events are missing, particularly click events on the Clear and Quit buttons. These can occur in any state; showing them results in a very cluttered diagram.

Figure 16.8 contains a detailed view of the internal behavior of the select country state, in which an exclusive-or relationship among the country buttons is maintained. The more detailed view in Figure 16.8 replaces the artificial input and output events with the corresponding multiple actual events; however, Figure 16.8 is incomplete.

The annotations for the transitions between Brazil and Japan, and the transitions between Canada and European Community, are not shown. Also, neither Figure 16.7 nor Figure 16.8 shows all the events that might occur in each state, especially clicking on either the Clear or Quit buttons.

Readers familiar with the StateChart notation will recognize the elegant way such repeti-

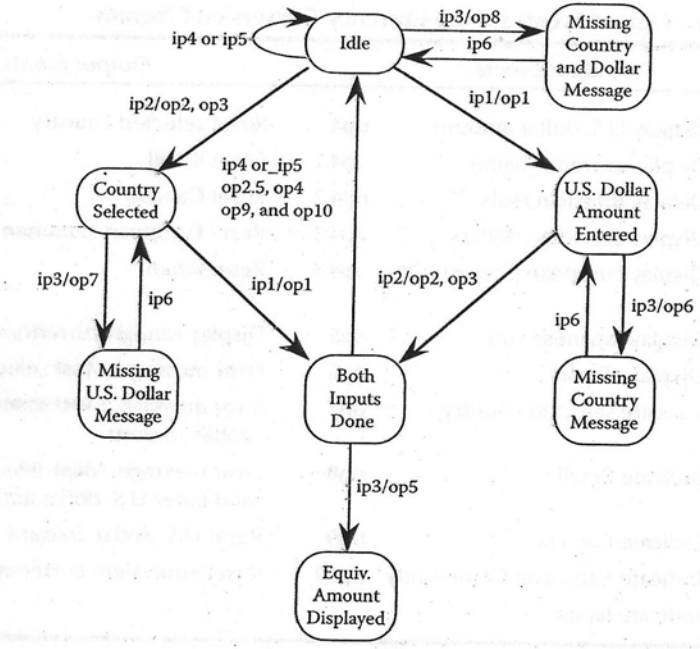
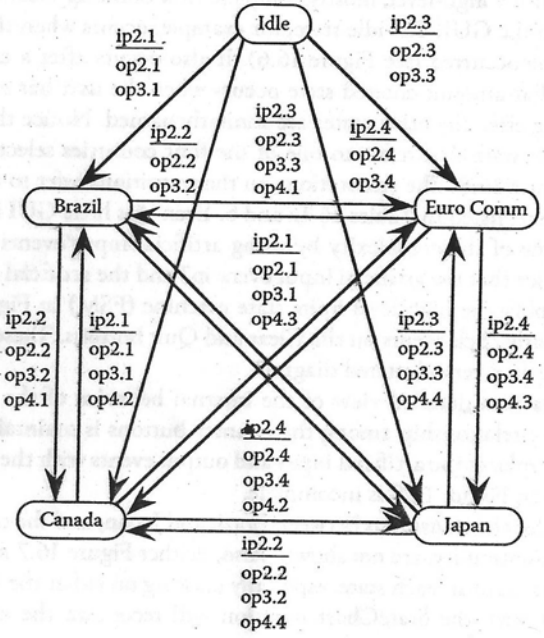



**Figure 16.7 High-level FSM.**



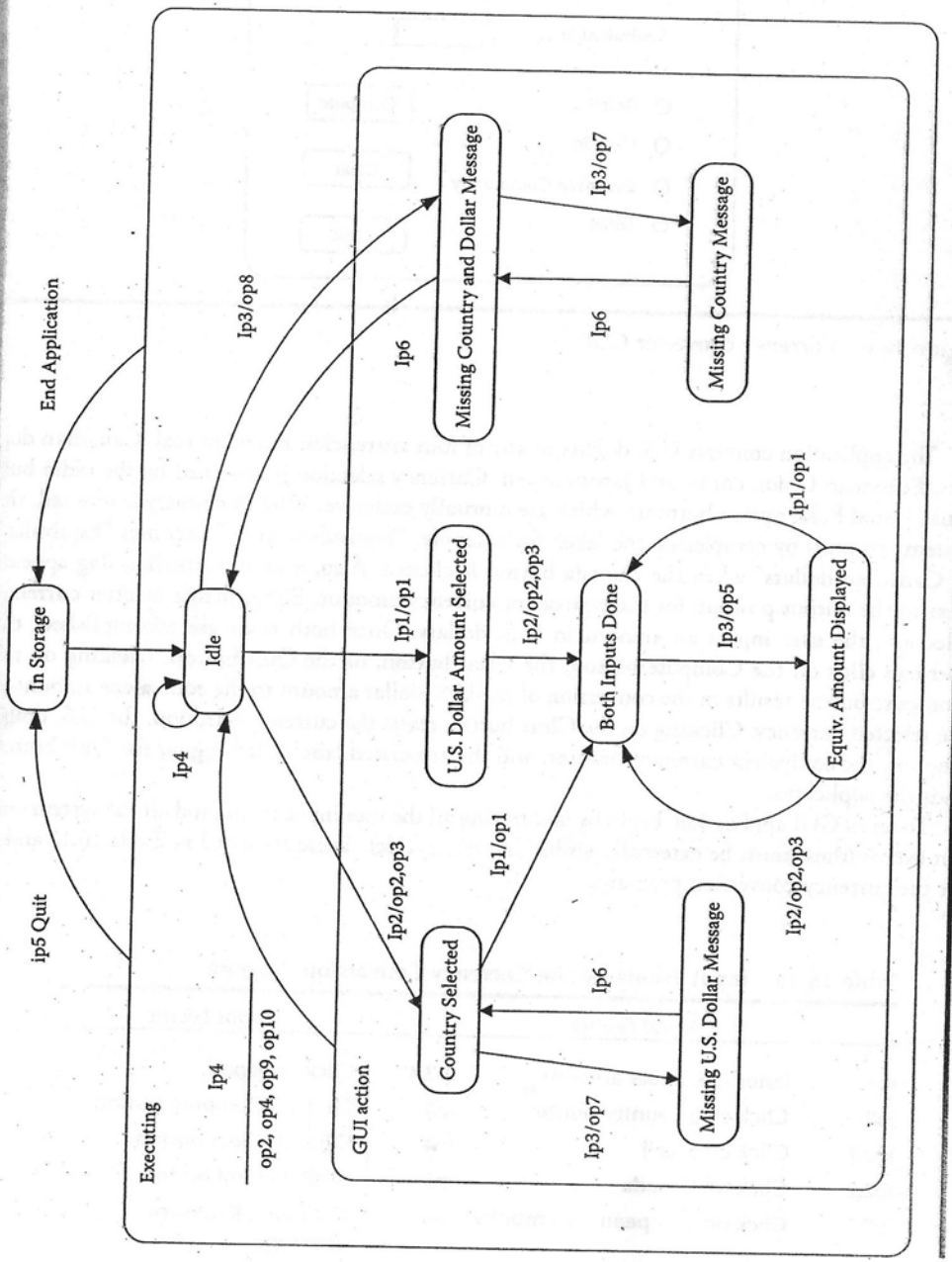




**Figure 16.9 StateChart of the currency controller application.**

**Table 16.2 Event Table for the Both Inputs Done State**

| *Input Events* | | *Output Events* | |
|---|---|---|---|
| ip1 | Enter U.S. dollar amount | op1 | Display U.S. dollar amount |
| ip2.1 | Click on Brazil | op2.1 op3.1 | Display Brazilian reals Indicate Brazil |
| ip2.2 | Click on Canada | op2.2 op3.2 | Display Canadian dollars Indicate Canada |
| ip2.3 | Click on European Community | op2.3 op3.3 | Display European Community euros Indicate European Community |
| ip2.4 | Click on Japan | op2.4 op3.4 | Display Japanese yen Indicate Japan |
| ip3 | Click on Compute button | op5 | Display foreign currency value |
| ip4 | Click on Clear button | op2.5 op4 op9 op10 | Display ellipsis, Reset selected country, Reset U.S. dollar amount, Reset equivalent currency amount |
| ip5 | Click on Quit button | | Application ends |
| ip6 | Click on OK in error message | | Ignored (error message not visible in this state) |

completely. Notice that the transitions from the executing state to the in-storage state show that the user input ip6 to quit the application, and the operating system "End Application" event can occur in any state internal to the executing state. Similarly, the clear input event (ip4) can occur from any state inside the unnamed state.

One way to deal with the suppressed events is to make individual, state-level diagrams, showing the response to each possible input event. This is also nicely done with an event table; Table 16.2 shows this for the "both inputs done" state.

## References


Binder, R.V., The free approach for testing use cases, threads, and relations, *Object*, Vol. 6, No. 2, February 1996.

Kiczales, G. et al., Aspect-oriented programming, in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, LNCS1241, Springer-Verlag, June 1997.


## Exercises

1. The o-oCalendar problem can be extended is several ways. One extension is to add an astrological content: each zodiac sign has a name and a beginning date (usually the 21st of a month). Add attributes and methods to the month class so that testIt can find the zodiac sign for a given date.

# *Chapter 17*

# Class Testing

The main question for class testing is whether a class or a method is a unit. Most of the o-o literature leans toward the class-as-unit side, but this definition has problems. In traditional software (where it is also hard to find a definition of a unit), the common guidelines are that a unit is:

The smallest chunk that can be compiled by itself
A single procedure/function (stand-alone)
Something so small it would be developed by one person

These guidelines also make sense for o-o software, but they do not resolve whether classes or methods should be considered as units. A method implements a single function, and it would not be assigned to more than one person, so methods might legitimately be considered units. The smallest compilation requirement is problematic. Technically, we could compile a single-method class by ignoring the other methods in the class (probably by commenting them out), but this creates an organizational mess. We will present both views of o-o unit testing; you can let particular circumstances decide which is more appropriate.

## 17.1 Methods as Units

Superficially, this choice reduces o-o unit testing to traditional (procedural) unit testing. A method is nearly equivalent to a procedure, so all the traditional functional and structural testing techniques should apply. Unit testing of procedural code requires stubs and a driver test program to supply test cases and record results. Similarly, if we consider methods as o-o units, we must provide stub classes that can be instantiated, and a "main program" class that acts as a driver to provide and analyze test cases.

When we look more closely at individual methods, we see the happy consequence of encapsulation: they are generally simple. In the next subsection, please find the pseudocode and corresponding program graphs for the classes that make up the o-oCalendar application. Notice that

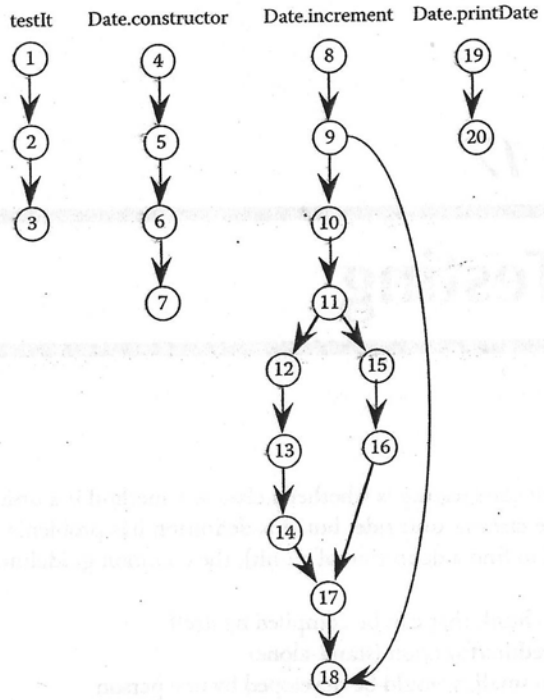



**Figure 17.1 Program graphs for the testIt and Date classes.**

the cyclomatic complexity is uniformly low; the increment method of the Date class has the highest cyclomatic complexity, and it is only V(G) = 3. To be fair, this implementation is intentionally simple — no checking is necessary for valid inputs. If there were, the cyclomatic complexity would increase.

Even though the cyclomatic complexity is low, the interface complexity is quite high. Looking at Date.increment again, notice the intense messaging: messages are sent to two methods in the Day class, to one method in the Year class, and to two methods in the Month class. This means that nearly as much effort will be made to create the proper stubs as in identifying test cases. Another more important consequence is that much of the burden is shifted to integration testing. In fact, we can identify two levels of integration testing: intraclass and interclass.

### 17.1.1 Pseudocode for o-oCalendar

Very little documentation is required by Unified Modeling Language (UML) at the unit/class level. Here, we add the class responsibility collaboration (CRC) cards for each class, followed by the class pseudocode, and then the program graphs for the class operations (see Figure 17.1 to Figure 17.4). CRC cards are not formally part of UML.

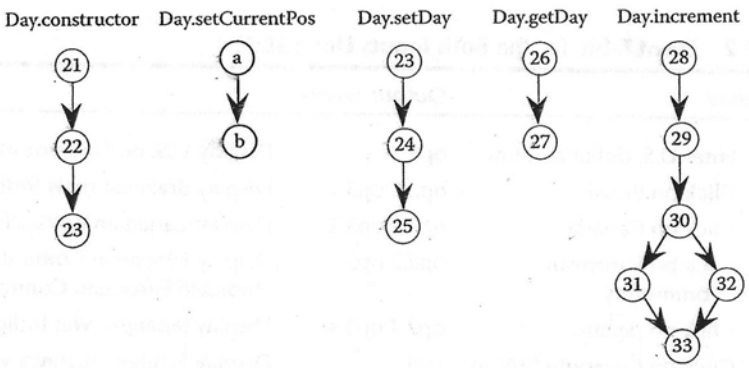



**Figure 17.2 Program graphs for the Day class.**

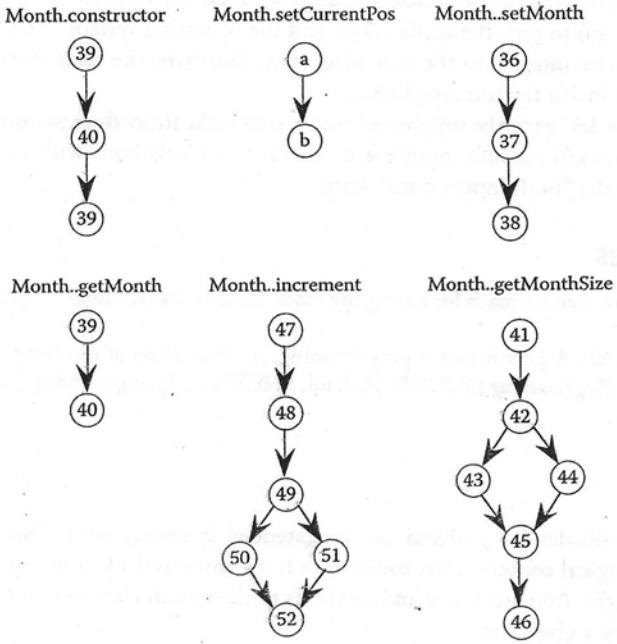



**Figure 17.3 Program graphs for the Month class.**

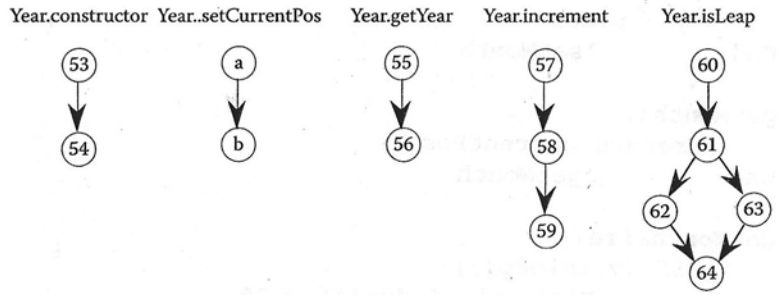



**Figure 17.4 Program graphs for the Year class.**

### 17.1.1.1 Class: CalendarUnit

Responsibility: Provides an operation to set its value in inherited classes and provides a Boolean operation that tells whether an attribute in an inherited class can be incremented.

```
Collaborates with: Day, Month, and Year

      class CalendarUnit 'abstract class
      currentPos As Integer

      CalendarUnit (pCurrentPos)
      currentPos = pCurrentPos
      End   'CalendarUnit

a.    setCurrentPos(pCurrentPos)
b.    currentPos = pCurrentPos
      End   'setCurrentPos
      abstract protected boolean increment()
```

### 17.1.1.2 Class: testIt

Responsibility: Serves as a test driver by creating a test date object, then requesting the object to increment itself and, finally, to print its new value.

```
Collaborates with:    Date

   class testIt
   main()
1       testdate = instantiate Date(testMonth, testDay, testYear)
2       testdate.increment()
3       testdate.printDate()

   End      'testIt
```

### 17.1.1.3 Class: Date

Responsibility: A Date object is composed of day, month, and year objects. A Date object increments itself using the inherited Boolean increment methods in day and month objects. If the day and month objects cannot be incremented (e.g., last day of the month or year), Date's increment method resets day and month as needed. In the case of December 31, it also increments the year. The printDate operation uses the get() methods in day, month, and year objects and prints a date value in mm/dd/yyyy format.

```
Collaborates with:    testIt, Day, Month, and Year

      class Date
      private Day d
      private Month m
      private Year y

4     Date(pMonth, pDay, pYear)
5           y = instantiate Year(pYear)
6           m = instantiate Month(pMonth, y)
7           d = instantiate Day(pDay, m)
      End   'Date constructor

8      increment ()
9     if (NOT(d.increment()))
10          Then
11                if (NOT(m.increment()))
12                      Then
13                            y.increment()
14                            m.setMonth(1,y)
15                      Else
16                            d.setDay(1, m)
17                EndIf
18          EndIf
      End   'increment

19    printDate ()
20          Output (m.getMonth() + "/" + d.getDay() + "/" +
            y.getYear())
      End 'printDate
```

### 17.1.1.4 Class: Day

Responsibility: A day object has a private month attribute that the increment method uses to see if a day value can be incremented or reset to 1. Day objects also provide get() and set() methods

```
Collaborates with:    Month

      class Day isA CalendarUnit
      private Month m

21    Day(pDay, Month pMonth)
22          setDay(pDay, pMonth)
      End         'Day constructor

23    setDay(pDay, Month pMonth)
24          setCurrentPos(pDay)
25          m = pMonth
      End   '     setDay

26    getDay()
27          return currentPos
      End         'getDay

28    boolean increment()
29          currentPos = currentPos + 1
30          if (currentPos <= m.getMonthSize())
31                Then  return true
32                Else        return false
33          EndIf
      End         'increment
```

### 17.1.1.5 Class: Month

Responsibility: Month objects have a value attribute that is used as a subscript to an array of values of last month days (e.g., the last day of January is 31, the last day of February is 28, and so on). Month objects provide get() and set() services, and the inherited Boolean increment method. The possibility of February 29 is determined with the isLeap message to a year object.

```
Collaborates with:    Year

      class Month isA CalendarUnit
            private Year y
            private sizeIndex ≙ <31, 28, 31, 30, 31, 30, 31,
            31, 30, 31, 30, 31>

34    Month( pcur, Year pYear)
35          setMonth(pCurrentPos, Year pyear)
      End         'Month constructor

38          y = pYear
      End         'setMonth

39    getMonth()
40          return currentPos
      End         'getMonth

41    getMonthSize()
42          if (y.isleap())
43                Then  sizeIndex[2] = 29
44                Else        sizeIndex[2] = 28
45          EndIf
46          return sizeIndex[currentPos -1]
      End         'getMonthSize

47    boolean increment()
48          currentPos = currentPos + 1
49          if (currentPos > 12)
50                Then  return false
51                Else        return true
52          EndIf

      End         'increment
```

### 17.1.1.6 Class: Year

Responsibility: In addition to the usual get() and set() methods, a year object increments itself when the test date is December 31 of any year. Year objects provide a Boolean service that tells whether the current value corresponds to a leap year.

```
Collaborates with:    (no external messages sent)

      class Year isA CalendarUnit
53    Year( pYear)
54          setCurrentPos(pYear)
      End         'Year constructor

55    getYear()
56          return currentPos
      End         'getYear

57    boolean increment()
58          currentPos = currentPos + 1
59          return true
```

```
60    boolean isleap()
61          if (((currentPos MOD 4 = 0) AND NOT(currentPos MOD
            400 = 0)) OR (currentPos MOD 400 = 0))
62                Then  return true
63                Else  return false
64          EndIf
      End           'isleap
```

### 17.1.2 Unit Testing for Date.increment

As we saw in Chapter 6, equivalence class testing is a good choice for logic-intensive units. The Date.increment operation treats the three equivalence classes of days:

D1 = {day: 1 ≤ day < last day of the month}
D2 = {day: day is the last day of a non-December month}
D3 = {day: day is December 31}

At first, these equivalence classes appear to be loosely defined, especially D1, with its reference to the unspecified last day of the month and no reference to which month. Thanks to encapsulation, we can ignore these questions. (Actually, the questions are transferred to the testing of the Month.increment operation.)

## 17.2 Classes as Units

Treating a class as a unit solves the intraclass integration problem, but it creates other problems. One has to do with various views of a class. In the static view, a class exists as source code. This is fine if all we do is code reading. The problem with the static view is that inheritance is ignored, but we can fix this by using fully flattened classes. We might call the second view the compile time view, because this is when the inheritance actually occurs. The third view is the execution time view, when objects of classes are instantiated. Testing really occurs with the third view, but we still have some problems. For example, we cannot test abstract classes because they cannot be instantiated. Also, if we are using fully flattened classes, we will need to "unflatten" them to their original form when our unit testing is complete. If we do not use fully flattened classes, to compile a class, we will need all the other classes above it in the inheritance tree. One can imagine the software configuration management implications of this requirement.

The class-as-unit choice makes the most sense when little inheritance occurs, and classes have what we might call internal control complexity. The class itself should have an interesting (as opposed to simple or boring) StateChart, and there should be a fair amount of internal messaging. To explore class-as-unit testing, we will revisit the windshield wiper example with a more complex version.

### 17.2.1 Pseudocode for the windshieldWiper Class

The three classes discussed in Chapter 16 are merged into one class here. With this formulation, methods sense lever and dial events and maintain the state of the lever and dial in the lever-

corresponding state variable wiperSpeed. Our revised windshieldWiper class has three attributes, get and set operations for each variable, and methods that sense the four physical events on the lever and dial devices.

```
class windshieldWiper
      private wiperSpeed
      private leverPosition
      private dialPosition

      windshieldWiper(wiperSpeed, leverPosition, dialPosition)

      getWiperSpeed()
      setWiperSpeed()

      getLeverPosition()
      setLeverPosition()

      getDialPosition()
      setDialPosition()

      senseLeverUp()
      senseLeverDown()

      senseDialUp()
      senseDialDown()

End class windshieldWiper
```

The class behavior is shown in the StateChart in Figure 17.5, where the three devices appear in the orthogonal components. In the Dial and Lever components, transitions are caused by events, whereas the transitions in the wiper component are all caused by propositions that refer to what state is active in the dial or lever orthogonal component. (Such propositions are part of the rich syntax of transition annotations permitted in the StateMate product.)

### 17.2.2 Unit Testing for the windshieldWiper Class

Part of the difficulty with the class-as-unit choice is that there are levels of unit testing. In our example, it makes sense to proceed in a bottom-up order beginning with the get/set methods for the state variables (these are only present in case another class needs them). The dial and lever sense methods are all quite similar; pseudocode for the senseLeverUp method is given next.

```
senseLeverUp()
     Case leverPosition Of
          Case 1: Off
               leverPosition = Int
               Case dialPosition Of
                    Case 1:1
                         wiperSpeed = 4
```

```
                Case 3:3
                    wiperSpeed = 12
        EndCase     'dialPosition
Case 2:Int
        leverPosition = Low
        wiperSpeed = 30
Case 3: Low
        leverPosition = High
        wiperSpeed = 60
Case 4: High
        (impossible; error condition)
EndCase     'leverPosition
```

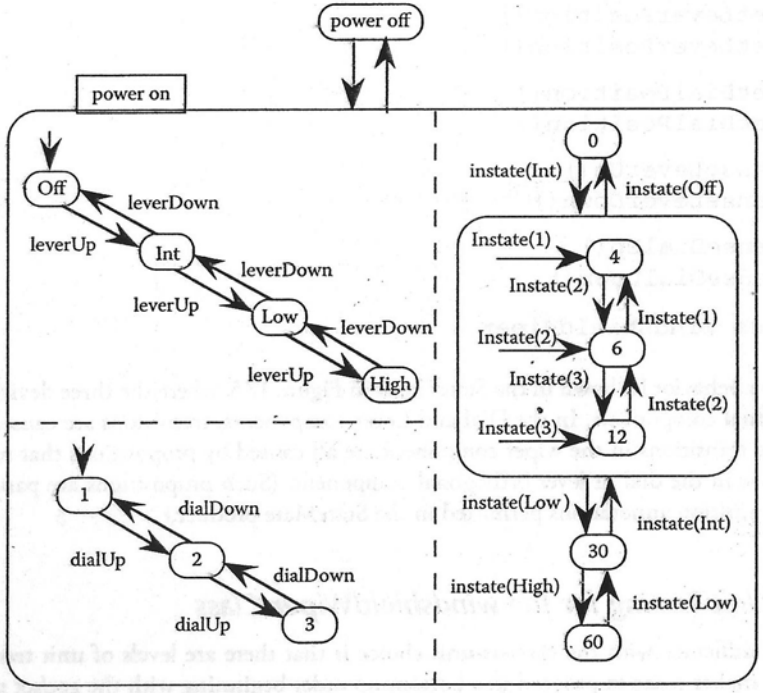



**Figure 17.5 StateChart for the windshieldWiper class.**

Testing the senseLeverUp method will require checking each of the alternatives in the case and nested case statements. The tests for the outer case statement cover the corresponding leverUp transitions in the statechart. In a similar way, we must test the leverDown, dialUp, and dialDown methods. Once we know that the dial and lever components are correct, we can test the wiper component.

Pseudocode for the test driver class will look something like this:

```
class testSenseLeverUp
        dialPos
        testResult  'boolean
main()
        testCase = instantiate windshieldWiper(0, Off, 1)
        windshieldWiper.senseLeverUp()
        leverPos = windshieldWiper.getLeverPosition()
        If leverPos = Int
              Then testResult = Pass
              Else testResult = Fail
        EndIf
End     'main
```

There would be two other test cases, testing the transitions from INT to LOW, and LOW to HIGH. Next, we test the rest of the windshieldWiper class with the following pseudocode.

```
class test WindshieldWiper
        wiperSpeed
        leverPos
        dialPos
        testResult  'boolean
main()
        testCase = instantiate windshieldWiper(0, Off, 1)
        windshieldWiper.senseLeverUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 4
              Then testResult = Pass
              Else testResult = Fail
        EndIf
End     'main
```

Two subtleties occur here. The easy one is that the instantiate windshieldWiper statement establishes the preconditions of the test case. The test case in the pseudocode happens to correspond to the default entry states of the Dial and Lever components of the StateChart in Figure 17.5. Notice it is easy to force other preconditions. The second subtlety is more obscure. The wiper component of the StateChart has what we might call the tester's (or external) view of the class. In this view, the wiper default entries and transitions are caused by various "inState" propositions. The implementation, however, causes these transitions by using the set methods to change the values of the state variables.

We have the StateChart definition of the class behavior; therefore, we can use it to define our test cases in much the same way that we used finite state machines to identify system-level test cases. StateChart-based class testing supports reasonable test coverage metrics. Some obvious ones are:

Every event
Every state in a component
Every transition in a component

Table 17.1 contains test cases with the instantiate statements (to establish preconditions) and expected outputs for the "every transition in a component" coverage level for the lever component.

**Table 17.1 Test Cases for the Lever Component**

| *Test Case* | *Preconditions (Instantiate Statement)* | *windshieldWiper Event (Method)* | *Expected Value of leverPos* |
|---|---|---|---|
| 1 | windshieldWiper(0, Off, 1) | senseLeverUp() | INT |
| 2 | windshieldWiper(0, Int, 1) | senseLeverUp() | LOW |
| 3 | windshieldWiper(0, Low, 1) | senseLeverUp() | HIGH |
| 4 | windshieldWiper(0, High, 1) | senseLeverDown() | LOW |
| 5 | windshieldWiper(0, Low, 1) | senseLeverDown() | INT |
| 6 | windshieldWiper(0, Int, 1) | senseLeverDown() | OFF |

Notice that the higher levels of coverage actually imply an intraclass integration of methods, which seems to contradict the idea of class as unit. The scenario coverage criterion is nearly identical to that of system-level testing. Here is a use case and the corresponding message sequences needed in a test class:

| *UC 1* | *Normal Usage* |
|---|---|
| Description | The windshield wiper is in the OFF position, and the Dial is at the 1 position; the user moves the lever to INT, and then moves the dial first to 2 and then to 3; the user then moves the lever to LOW; the user moves the lever to INT, and then to OFF |
| Preconditions | The windshield wiper is in the OFF position, and the Dial is at the 1 position; wiper speed is 0 |

| *Event Sequence* | *User Action* | *System Response* |
|---|---|---|
| 1 | Move lever to INT | Wiper speed is 4 |
| 2 | Move dial to 2 | Wiper speed is 6 |
| 3 | Move dial to 3 | Wiper speed is 12 |
| 4 | Move lever to LOW | Wiper speed is 20 |
| 5 | Move lever to INT | Wiper speed is 12 |
| 6 | Move lever to OFF | Wiper speed is 0 |

```
class testScenario
    wiperSpeed
    leverPos
    step1OK     'boolean
    step2OK     'boolean
    step3OK     'boolean
    step4OK     'boolean
    step5OK     'boolean
    step6OK     'boolean

main()
    testCase = instantiate windshieldWiper(0, Off, 1)
    windshieldWiper.senseLeverUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 4
        Then step1OK = Pass
        Else step1OK = Fail
    EndIf

    windshieldWiper.senseDialUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 6
        Then step2OK = Pass
        Else step2OK = Fail
    EndIf

    windshieldWiper.senseDialUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 12
        Then step3OK = Pass
        Else step3OK = Fail
    EndIf

    windshieldWiper.senseLeverUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 20
        Then step4OK = Pass
        Else step4OK = Fail
    EndIf

    windshieldWiper.senseLeverDown()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 12
        Then step5OK = Pass
        Else step5OK = Fail
    EndIf

    windshieldWiper.senseLeverDown()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 0
        Then step6OK = Pass
        Else step6OK = Fail
```

# Chapter 18

# Object-Oriented Integration Testing

Of the three main levels of software testing, integration testing is the least understood; this is true for both traditional and object-oriented software. As with traditional procedural software, object-oriented integration testing presumes complete unit-level testing. Both unit choices have implications for object-oriented integration testing. If the operation/method choice is taken, two levels of integration are required: one to integrate operations into a full class and one to integrate the class with other classes. This should not be dismissed. The whole reason for the operation-as-unit choice is that the classes are very large, and several designers were involved.
Turning to the more common class-as-unit choice, once the unit testing is complete, two steps must occur: (1) if flattened classes were used, the original class hierarchy must be restored, and (2) if test methods were added, they must be removed.

Once we have our "integration test bed," we need to identify what needs to be tested. As we saw with traditional software integration, static and dynamic choices can be made. We can address the complexities introduced by polymorphism in a purely static way: test messages with respect to each polymorphic context. The dynamic view of object-oriented integration testing is more interesting.

## 18.1 UML Support for Integration Testing

In this chapter and the next, we will use the Unified Modeling Language (UML) to describe our examples. In UML-defined, object-oriented software, collaboration and sequence diagrams are the basis for integration testing. At the system level, UML description is comprised of various levels of use cases, a use case diagram, class definitions, and class diagrams (see Figure 16.4 and Figure 16.5). Once this level is defined, integration-level details are added. A collaboration diagram shows some of the message traffic among classes. Figure 18.1 is a collaboration diagram for the o-oCalendar application. A collaboration diagram is very analogous to the unit call graph we used in Chapter 13 (for example, see Figure 13.2). As such, a collaboration diagram supports both

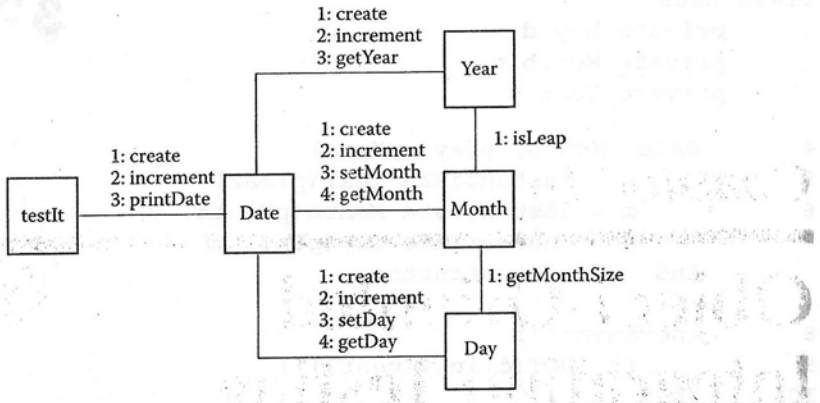



**Figure 18.1 Collaboration diagram for o-oCalendar.**

With pairwise integration, a unit (class) is tested in terms of separate adjacent classes that either send messages to or receive messages from the class being integrated. To the extent that the class sends/receives messages from other classes, the other classes must be expressed as stubs. All this extra effort makes pairwise integration of classes as undesirable as pairwise integration of procedural units. Based on the collaboration diagram in Figure 18.1, we would have the following pairs of classes to integrate:

testIt and Date, with stubs for Year, Month, and Day
Date and Year, with stubs for testIt, Month, and Day
Date and Month, with stubs for testIt, Year, and Day
Date and Day, with stubs for testIt, Month, and Year
Year and Month, with stubs for Date and Day
Month and Day, with stubs for Date and Year

One drawback to basing object-oriented integration testing on collaboration diagrams is that at the class level, the behavior model of choice in UML is the StateChart. For class-level behavior, StateCharts are an excellent basis of test cases, and this is particularly appropriate for the class-as-unit choice. The problem, however, is that in general it is difficult to combine StateCharts to see behavior at a higher level (Regmi, 1999).

Neighborhood integration raises some very interesting questions from graph theory. Using the (undirected) graph in Figure 18.1, the neighborhood of Date is the entire graph, while the neighborhood of testIt is just Date. It turns out that the mathematicians have identified various "centers" of a linear graph. One of them, for example, is the ultra-center, which minimizes the maximum distances to the other nodes in the graph. In terms of an integration order, we might picture the circular ripples caused by tossing a stone in calm water. We start with the ultra-center and the neighborhood of nodes one edge away, then add the nodes two edges away, and so on. Neighborhood integration of classes will certainly reduce the stub effort, but this will be at the expense of diagnostic precision. If a test case fails, we will have to look at more classes to find the fault.

A sequence diagram traces an execution time path through a collaboration diagram. (In UML,

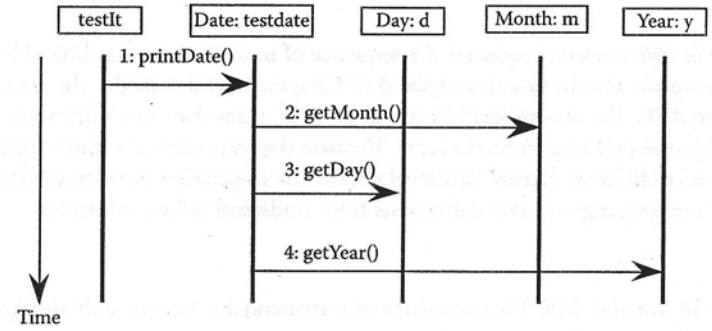



**Figure 18.2 Sequence diagram for printDate.**

messages sent by (instances of) the classes in their time order. The portion of the o-oCalendar application that prints out the new date is shown as a sequence diagram in Figure 18.2.

To the extent that sequence diagrams are created, they are an excellent basis for object-oriented integration testing. They are nearly equivalent to the object-oriented version of MM-Paths (which we define in the next subsection). An actual test for this sequence diagram would have pseudocode similar to this:

```
1. testDriver
2.     m.setMonth(1)
3.     d.setDay(15)
4.     y.setYear(2002)
5.     Output ("expected value is 1/15/2002"
6.     Output ("actual output is ")
7.     testIt.printDate()
8.     End testDriver
```

Statements 2 to 4 use the previously unit-tested methods to set the expected output in the classes to which messages are sent. As it stands, this test driver depends on a person to make a pass/fail judgment based on the printed output. We could put comparison logic into the testDriver class to make an internal comparison. This might be problematic in the sense that if we made a mistake in the code tested, we might make the same mistake in the comparison logic.

## 18.2 MM-Paths for Object-Oriented Software

When we spoke of MM-Paths in traditional software, we used *message* to refer to the invocation among separate units (modules), and we spoke of module execution paths (module-level threads) instead of full modules. Here, we use the same acronym to refer to an alternating sequence of method executions separated by messages — method-to-message path. Just as in traditional software, methods may have several internal execution paths. We choose not to operate at that level of detail for

***Definition***

An *MM-Path in object-oriented software* is a sequence of method executions linked by messages.

Recall the atomic system functions (ASFs) of Chapter 14, and consider the set of classes that support a given ASF. The system-level nature of an ASF means that, as a minimum, it represents a stimulus–response pair of port-level events. Because the port events are (most likely) associated with operations of different classes, this level entails the cooperation of distinct classes. We only need to make slight changes to two definitions from traditional software testing.

***Definition***

An *atomic system function* (*ASF*) is a sequence of statements that begins with an input port event and ends with an output port event.

The ASF construct addresses the event-driven nature of object-oriented software. Because ASFs commonly begin with a port input event and end with a port output event, they constitute what traditional models call a stimulus–response path. This system-level input triggers the method–message sequence of an MM-Path, which may trigger other MM-Paths until, finally, the sequence of MM-Paths ends with a port output event. When such a sequence ends, the system is event quiescent; that is, the system is waiting for another port input event that initiates another ASF. Just as we had with traditional software, this formulation places ASF testing at the point where integration- and system-level testing overlap.

### 18.2.1 Pseudocode for o-oCalendar

To see the complexity of object-oriented integration testing, we examine the o-oCalendar application in detail. The pseudocode is given next, with comments numbering the messages. The message traffic is shown in Figure 18.3 (just the message numbers are shown to reduce congestion).

```
class CalendarUnit    'abstract class
      currentPos As Integer

      CalendarUnit(pCurrentPos)
            currentPos = pCurrentPos
      End   'CalendarUnit

a     setCurrentPos(pCurrentPos)
b           currentPos = pCurrentPos
      End   'setCurrentPos

      abstract protected boolean increment()

class testIt
      main()
1           testdate = instantiate Date(testMonth, testDay,
            testYear)                                          msg1
2           testdate.increment()                               msg2
```

```
class Date
      private Day d
      private Month m
      private Year y

4       Date(pMonth, pDay, pYear)
5           y = instantiate Year(pYear)                        msg4
6           m = instantiate Month(pMonth, y)                   msg5
7           d = instantiate Day(pDay, m)                       msg6
        End 'Date constructor

8       increment ()
9           if (NOT(d.increment()))                            msg7
10      Then
11          if (NOT(m.increment()))                            msg8
12                Then
13                      y.increment()                          msg9
14                      m.setMonth(1,y)                        msg10
15                Else
16                      d.setDay(1, m)                         msg11
17          EndIf
18      EndIf
        End 'increment

19    printDate ()
20          Output (m.getMonth() + "/" + d.getDay() + "/" +
            y.getYear())
                                                  msg12, msg13, msg14
      End         'printDate

class Day isA CalendarUnit
      private Month m

21    Day(pDay, Month pMonth)
22          setDay(pDay, pMonth)                               msg15
      End         'Day constructor

23    setDay(pDay, Month pMonth)
24          setCurrentPos(pDay)                                msg16
25          m = pMonth
      End         'setDay

26    getDay()
27          return currentPos
      End         'getDay

28    boolean increment()
```

```
30          if (currentPos <= m.getMonthSize())                msg17
31               Then  return true
32               Else  return false
33          EndIf
    End           'increment

    class Month isA CalendarUnit
          private Year y
          private sizeIndex = <31, 28, 31, 30, 31, 30, 31,
          31, 30, 31, 30, 31>

34  Month( pcur, Year pYear)
35          setMonth(pCurrentPos, Year pyear)                  msg18
    End           'Month constructor

36   setMonth( pcur, Year pYear)
37          setCurrentPos(pcur)                                msg19
38          y = pYear
    End           'setMonth

39  getMonth()
40          return currentPos
    End           'getMonth

41  getMonthSize()
42          if (y.isleap())                                    msg20
43               Then  sizeIndex[1] = 29
44               Else  sizeIndex[1] = 28
45          EndIf
46          return sizeIndex[currentPos -1]
    End           'getMonthSize

47  boolean increment()
48          currentPos = currentPos + 1
49          if (currentPos > 12)
50               Then  return false
51               Else  return true
52          EndIf
    End           'increment

    class Year isA CalendarUnit

53  Year( pYear)
54          setCurrentPos(pYear)                               msg21
    End           'Year constructor

56          return currentPos
    End           'getYear

57  boolean increment()
58          currentPos = currentPos + 1
59          return true
    End           'increment

60  boolean isleap()
61          if (((currentPos MOD 4 = 0) AND NOT(currentPos MOD
            100 = 0)) OR(currentPos MOD 400 = 0))
62               Then  return true
63               Else  return false
64          EndIf
    End           'isleap
```

Figure 18.3 is a more detailed view of the collaboration diagram in Figure 18.1. The difference is that the source and destination methods of each method are shown, instead of just the classes. With this formulation, we can view object-oriented integration testing independently of whether the units were chosen as methods or classes.

Here is a partial MM-Path to instantiate January 15, 2007. As we did in Chapter 13, we simply use the statement and message numbers from the pseudocode. This MM-Path is shown in Figure 18.4.

```
testIt<1>
      msg1
Date:testdate<4, 5>
      msg4
Year:y<53, 54>
      msg21
Year:y.setCurrentPos<a, b>
      (return to Year.y)
      (return to Date:testdate)
Date:testdate<6>
      msg5
Month:m<34, 35>
      msg18
Month:m.setMonth<36, 37>
      msg19
Month:m.setCurrentPos<a, b>
      (return to Month:m.setMonth)
      (return to Month:m)
      (return to Date:testdate)
Date:testdate<7>
      msg6
```

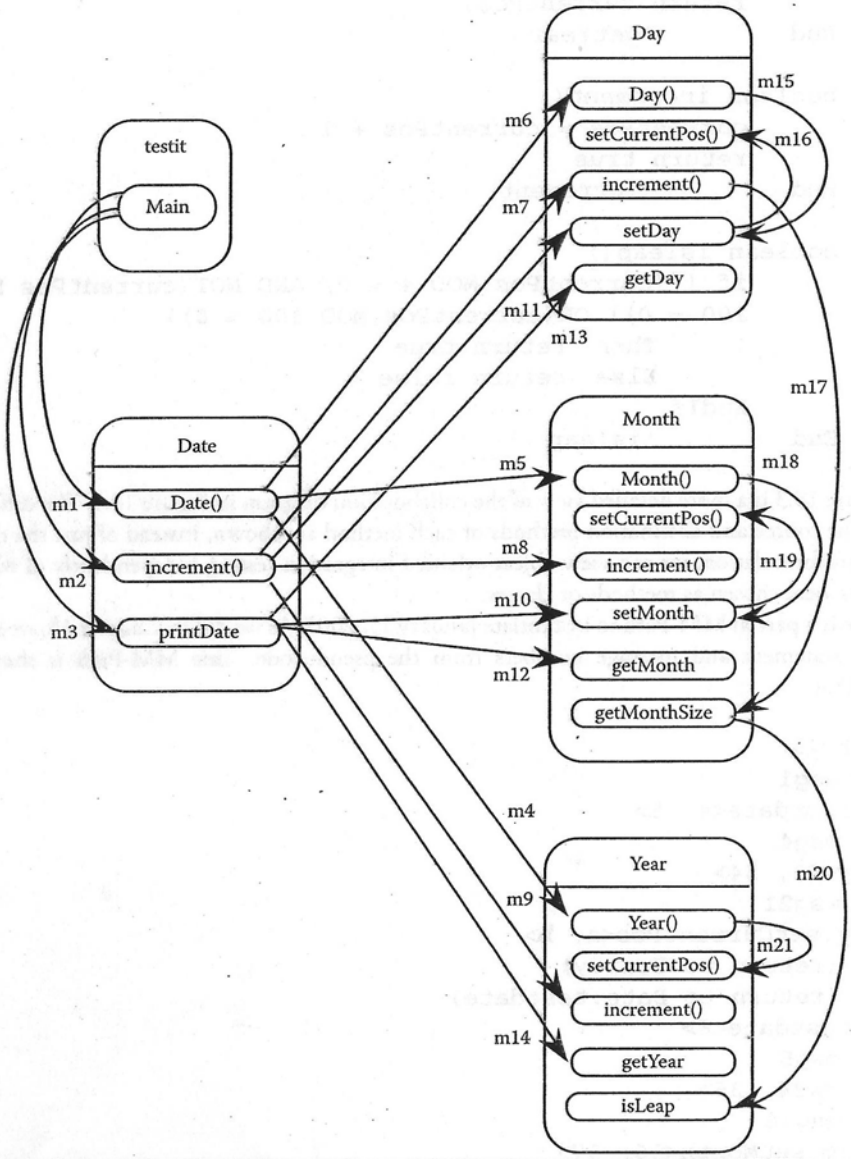



**Figure 18.3 Messages in o-oCalendar.**

```
Day:d.setDay<23, 24>
      msg16
Day:d.setCurrentPos<a, b>
      (return to Day:d.setDay)
Day:d.setDay<25>
```

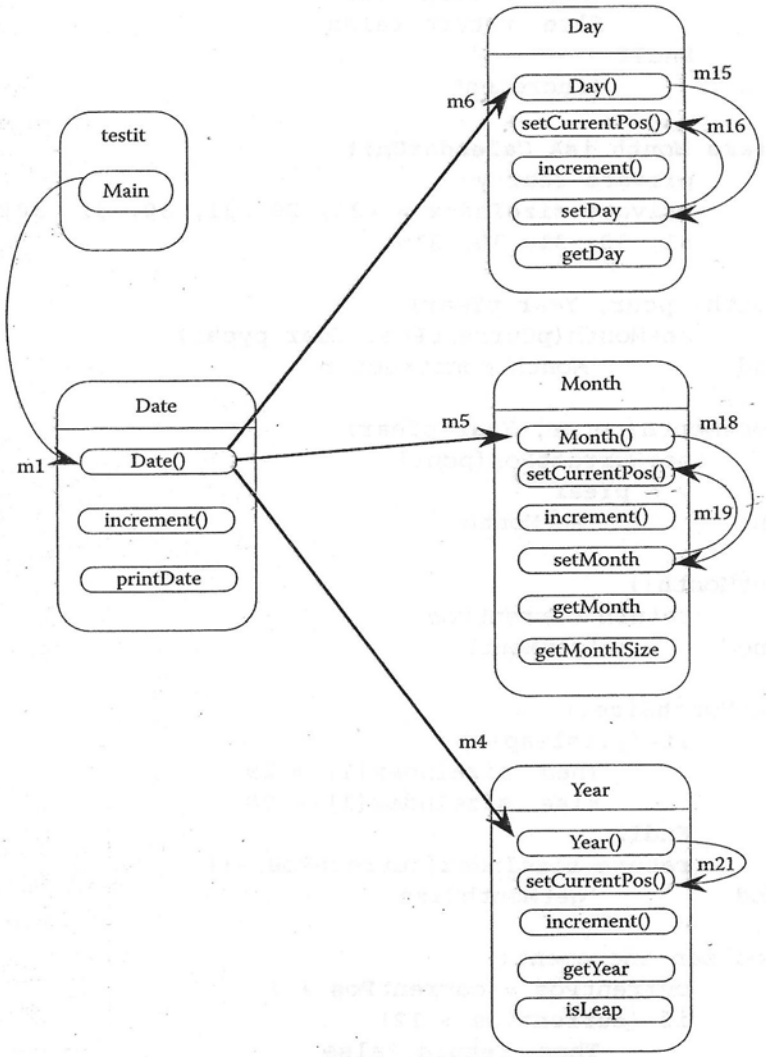



**Figure 18.4 MM-Path for instantiate date (1, 15, 2007).**

Here is a more interesting MM-Path, for the instantiated date April 30, 2007 (see Figure 18.5).

```
testIt<2>
      msg2
Date:testdate.increment<8, 9>
      msg7
```

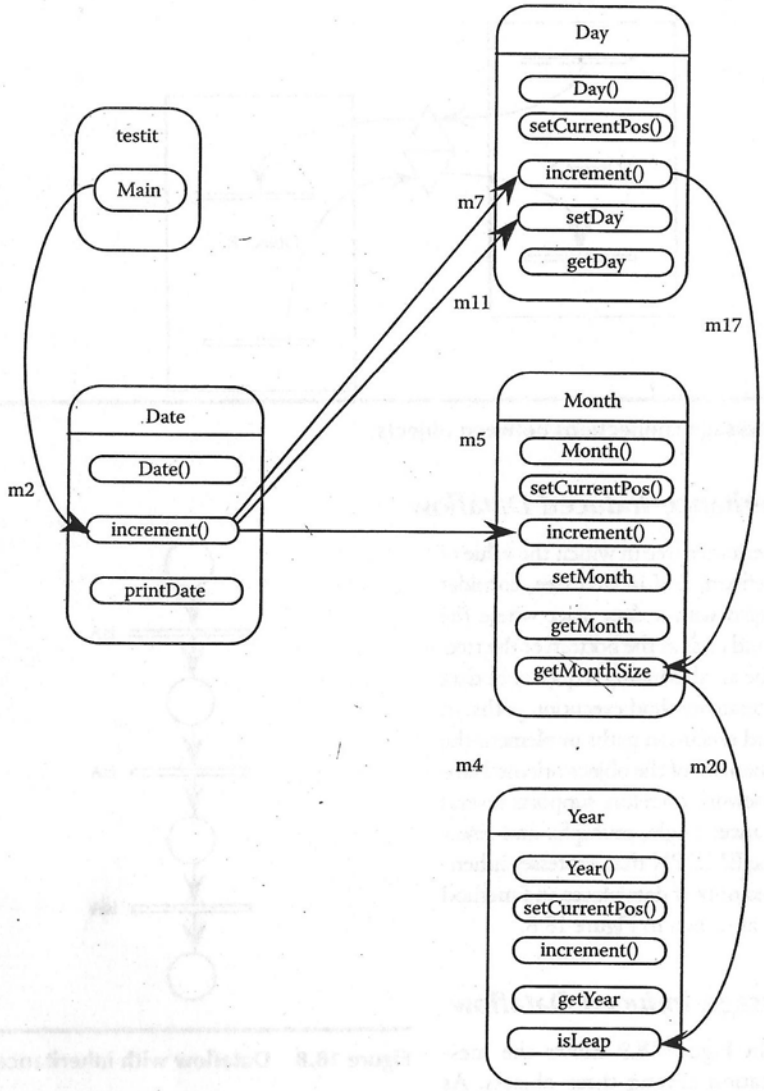



**Figure 18.5 MM-Path for Date.increment (4, 30, 2007).**

```
Month:m.getMonthSize<41, 42>
      msg20
Year:y.isleap<60, 61, 63, 64>      'not a leap year
      (return to Month:m.getMonthSize)
Month:m.getMonthSize<44, 45, 46>   'returns month size = 30
      (return to Day:d.increment)
Date:testdate.increment<10,11>
      msg8
Month:m.increment<47, 48, 49, 51, 52>    'returns true
      (return to Date:testdate.increment)
Date:testdate.increment<15, 16>
      msg11
Day:d.setDay<23, 24, 25>      'now day is 1, month is 5
      (return to Date:testdate.increment)
Date:testdate.increment<17, 18>
      return to testIt>
```

Having a directed graph formulation puts us in a position to be analytical about choosing MM-Path–based integration test cases. First, we might ask how many test cases will be needed. The directed graph in Figure 18.3 has a cyclomatic complexity of 23 (the return edges of each message are not shown, but they must be counted). Although we could certainly find the same number of basis paths, it is not necessary to do this because a single MM-Path will cover many of these paths, and many more are logically infeasible. A lower limit would be three test cases: the MM-Paths beginning with statements 1, 2, and 3 in the testIt pseudocode. This might not be sufficient because, for example, if we choose an "easy" date (such as January 15, 2007), the messages involving isLeap and setMonth will not occur. Just as we saw with unit-level testing of procedural code, as a minimum, we need a set of MM-Paths that cover every message. The 13 decision table-based functional test cases we identified for NextDate in Chapter 7 (see Table 7.16) constitute a thorough set of integration test cases for the o-oCalendar application. This is a point where the structural view of object-oriented integration testing gives insights that we cannot get from the functional view. We can look for MM-Paths to make sure that every message (edge) in the graph of Figure 18.3 is traversed.

## 18.3 A Framework for Object-Oriented Dataflow Integration Testing

MM-Paths were defined to serve as integration testing analogs of DD-Paths. As we saw for procedural software, DD-Path testing is often insufficient, and in such cases, dataflow testing is more appropriate. The same holds for integration testing of object-oriented software; if anything, the need is greater for two reasons: (1) data can get values from the inheritance tree, and (2) data can be defined at various stages of message passing.

Program graphs formed the basis for describing and analyzing dataflow testing for procedural code. The complexities of object-oriented software exceed the expressive capabilities of directed (program) graphs. This subsection presents an expressive framework within which dataflow testing questions for object-oriented software can be described and analyzed.

### *18.3.1 Event- and Message-Driven Petri Nets*

Event-driven Petri nets were defined in Chapter 4; we extend them here to express the message

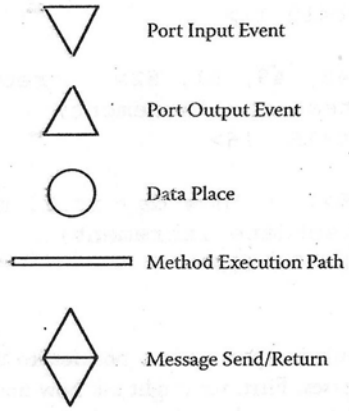



**Figure 18.6 Symbols for EMDPNs.**

***Definition***

An *Event- and message-driven petri net* (EMDPN) is a quadripartite directed graph (P, D, M, S, In, Out) composed of four sets of nodes, P, D, M, and S, and two mappings, In and Out, where:

P is a set of port events
D is a set of data places
M is a set of message places
S is a set of transitions
In is a set of ordered pairs from $(P \cup D \cup M) \times S$
Out is a set of ordered pairs from $S \times (P \cup D \cup M)$

We retain the port input and output events because these will certainly occur in event-driven, object-oriented applications. Obviously, we still need data places, and we will interpret Petri net transitions as method execution paths. The new symbol is intended to capture the essence of interobject messages:

They are an output of a method execution path in the sending object.
They are an input to a method execution path in the receiving object.
The return is a very subtle output of a method execution path in the receiving object.
The return is an input to a method execution path in the sending object.

Figure 18.7 shows the only way that the new message place can appear in an EMDPN.

The EMDPN structure, because it is a directed graph, provides the needed framework for dataflow analysis of object-oriented software. Recall that dataflow analysis for procedural code centers on nodes where values are defined and used. In the EMDPN framework, data is represented by a data place, and values are defined and used in method execution paths. A data place can be either an input to or an output of a method execution path; therefore, we can now represent the define/use paths (du-path) in a way very similar to that for procedural code. Even though four

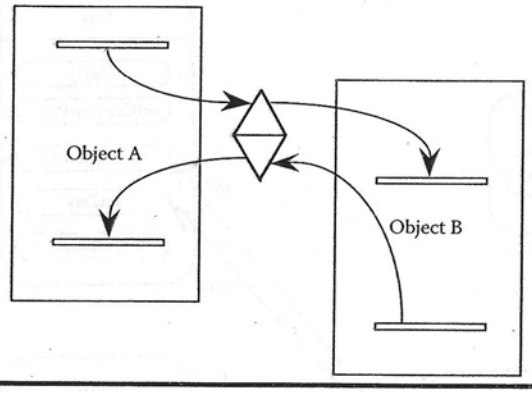



**Figure 18.7 Message connections between objects.**

### 18.3.2 Inheritance-Induced Dataflow

Consider an inheritance tree in which the value of a data item is defined, and in that tree, consider a chain that begins with a data place where the value is defined and ends at the bottom of the tree. That chain will be an alternating sequence of data places and degenerate method execution paths, in which the method execution paths implement the inheritance mechanism of the object-oriented language. This framework therefore supports several forms of inheritance: single, multiple, and selective multiple. The EMDPN that expresses inheritance is comprised only of data places and method execution paths, as shown in Figure 18.8.

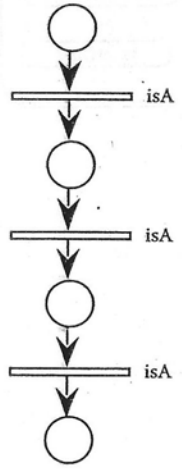



**Figure 18.8 Dataflow with inheritance.**

### 18.3.3 Message-Induced Dataflow

The EMDPN in Figure 18.9 shows the message communication among three objects. As an example of a du-path, suppose mep3 is a define node for a data item that is passed on by mep5, modified in mep6, and finally used in the use node mep2. We can identify these two du-paths:

```
du1 = <mep3, msg2, mep5, d6, mep6, return(msg2), mep4,
      return(msg1), mep2>
du2 = <mep6, return(msg2), mep4, return(msg1), mep2>
```

In this example, du2 is definition-clear, du1 is not.

### 18.3.4 Slices?

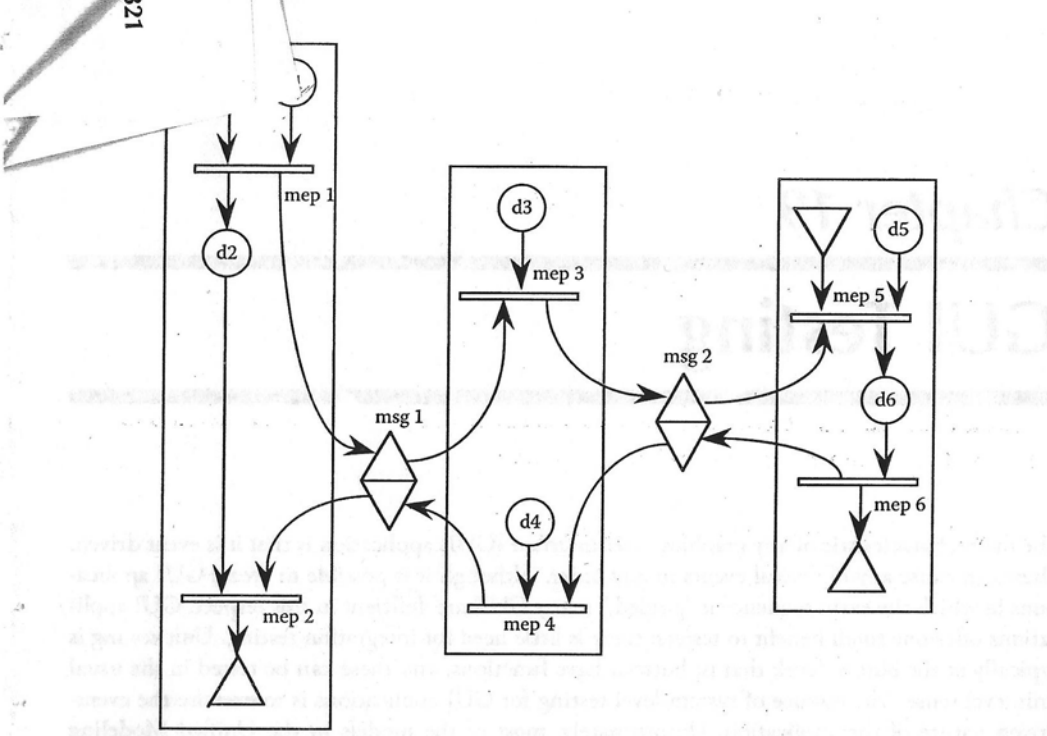



**Figure 18.9 Dataflow via message communication.**

desirable form of a slice is one that is executable. This appears to be a real stretch, and without it, such slices are interesting only as a desk-checking approach to fault location.

## Reference

Regmi, D.R., Object-Oriented Software Construction Based on State Charts, Grand Valley State University master's project, Allendale, MI, 1999.

## Exercises

Here is a real change of pace. It is an excerpt from a play that my wife's sixth-grade classes sometimes perform. It is also a sustained analogy with object-oriented integration testing. Using this analogy, apply the integration testing concepts we studied to play rehearsals. To start, consider a character in a play to be a class. Because characters communicate by cues, we can consider a character's line as a message to another character.

First, the "source code" (excerpted from *The Phantom Tollbooth*, by Susan Janus (based on the book by Norton Juster), Act I, Scene 2):

1. **WATCHDOG.** Dictionopolis, here we come.
4. **MILO.** What kind of place is Dictionopolis, anyway?
5. **TOCK.** It's where all the words in the world come from. It used to be a marvelous place, but ever since Rhyme and Reason left, it hasn't been the same.
6. **MILO.** Rhyme and Reason?
7. **TOCK.** The two princesses. They used to settle all the arguments between their two brothers who rule over the Land of Wisdom. You see, Azaz is the king of Dictionopolis and the Mathemagician is the king of Digitopolis and they almost never see eye to eye on anything. It was the job of the Princesses Sweet Rhyme and Pure Reason to solve the differences between the two kings, and they always did so well that both sides usually went home satisfied. But then, one day, the kings had an argument to end all arguments...

   [The lights dim on Tock and Milo and come up on King Azaz of Dictionopolis on another part of the stage. Azaz has a great stomach, a gray beard reaching to his waist, a small crown, and a long robe with the letters of the alphabet written all over it.]
8. **Azaz.** Of course, I'll abide by the decision of Rhyme and Reason, though I have no doubt as to what it will be. They will choose words, of course. Everyone knows that words are more important than numbers any day of the week.

   [The Mathemagician appears opposite Azaz. The Mathemagician wears a long, flowing robe covered entirely with complex mathematical equations, and a tall, pointed hat. He carries a long staff with a pencil point at one end and a large rubber eraser at the other.]
9. **Mathemagician.** That's what you think, Azaz. People wouldn't even know what day of the week it is without numbers. Haven't you ever looked at a calendar? Face it, Azaz, It's numbers that count.
10. **Azaz.** Don't be ridiculous. [To audience, as if leading a cheer.] Let's hear it for WORDS!
11. **Mathemagician.** [To audience, in the same manner.] Cast your vote for NUMBERS!
12. **Azaz.** A, B, C's!
13. **Mathemagician.** 1, 2, 3's!
14. **Azaz and Mathemagician.** [To each other.] Quiet! Rhyme and Reason are about to announce their decision.
15. **Rhyme.** Ladies and gentlemen, letters and numerals, fractions and punctuation marks — may we have your attention please. After careful consideration of the problem set before us by King Azaz of Dictionopolis [Azaz bows.] and the Mathemagician of Digitopolis [Mathemagician raises his hands in a victory salute.] we have come to the following conclusion.
16. **Reason.** Words and numbers are of equal value, for in the cloak of knowledge, one is the warp and the other is the woof.
17. **Rhyme.** It is no more important to count the sands than it is to name the stars.
18. **Rhyme and Reason.** Therefore, let both kingdoms, Dictionopolis and Digitopolis, live in peace.

    [The sound of cheering is heard.]
19. **Azaz.** Boo! is what I say. Boo and Bah and Hiss!
20. **Mathemagician.** What good are these girls if they can't even settle an argument in anyone's favor? I think I have come to a decision of my own.
21. **Azaz.** So have I.
22. **Azaz and Mathemagician.** [To the princesses.] You are hereby banished from this

[During this time, the set is changed to the Market Square of Dictionopolis. The lights come up on the deserted square.]

23. **TOCK.** And ever since then, there has been neither Rhyme nor Reason in this kingdom. Words are misused and numbers are mismanaged. The argument between the two kings has divided everyone and the real value of both words and numbers has been forgotten. What a waste!
24. **MILO.** Why doesn't somebody rescue the princesses and set everything straight again?
25. **TOCK.** That is easier said than done. The Castle-in-the-Air is very far from here, and the one path that leads to it is guarded by ferocious demons. But hold on, here we are. [A man appears, carrying a gate and a small tollbooth.]
26. **Gatekeeper.** AHHHHREMMMM! This is Dictionopolis, a happy kingdom, advantageously located in the foothills of confusion and caressed by gentle breezes from the Sea of Knowledge. Today, by royal proclamation, is Market Day. Have you come to buy or sell?
27. **MILO.** I beg your pardon?
28. **Gatekeeper.** Buy or sell, buy or sell. Which is it? You must have come here for a reason.
29. **MILO.** Well, I ...
30. **Gatekeeper.** Come now, if you don't have a reason, you must at least have an explanation or certainly an excuse.
31. **MILO.** [Meekly.] Uh ... no.
32. **Gatekeeper.** [Shaking his head.] Very serious. You can't get in without a reason. [Thoughtfully.] Wait a minute. Maybe I have an old one you can use. [Pulls out an old suitcase from the tollbooth and rummages through it.] No ... no ... no ... this won't do ... hmmmm ...
33. **MILO.** [To Tock.] What's he looking for? [Tock shrugs.]
34. **Gatekeeper.** Ah! This is fine. [Pulls out a medallion on a chain. Engraved in the medallion is "WHY NOT?"] Why not. That's a good reason for almost anything ... a bit used perhaps, but still quite serviceable. There you are, sir. Now I can truly say: Welcome to Dictionopolis.

Finally, the exercises:

1. Develop the analog of a call graph (or collaboration diagram) for the characters in this excerpt.
2. What is the analog of pairwise integration? How effective is this as a rehearsal technique? List several likely pairs of characters that would rehearse.
3. What is the analog of neighborhood integration? How effective is this as a rehearsal technique? Describe at least two neighborhoods of characters.
4. Can you find something that corresponds to an ASF?
5. A play director might schedule rehearsals in much the same way that a software development project would develop builds for integration testing. What builds do you find?
6. In what ways does the difference between ordinary and dress rehearsals correspond to the difference between integration and system testing?
7. What happens to this analogy if we take characters to be objects and cues to be messages? Take lines 1 to 7 as scene 1, lines 8 to 22 as scene 2, and lines 23 to 34 as scene 3, and

# *Chapter 19*

# GUI Testing

The main characteristic of any graphical user interface (GUI) application is that it is event driven. Users can cause any of several events in any order. Although it is possible to create GUI applications in which the event sequence is "guided," many GUIs are deficient in this respect. GUI applications offer one small benefit to testers: there is little need for integration testing. Unit testing is typically at the button level; that is, buttons have functions, and these can be tested in the usual unit-level sense. The essence of system-level testing for GUI applications is to exercise the event-driven nature of the application. Unfortunately, most of the models in the Unified Modeling Language (UML) are of little help with event-driven systems. The main exception is behavioral models, specifically StateCharts and their simpler case, finite state machines.

## 19.1 The Currency Conversion Program

The currency conversion program is an event-driven program that emphasizes code associated with a GUI. A sample GUI built with Visual Basic is shown in Figure 19.1 (repeated from Chapters 2 and 16). The full description of the currency conversion program is in Section 16.8.2. Those pages are not repeated here, but they are necessary to the following discussion.

## 19.2 Unit Testing for the Currency Conversion Program

The functional buttons (Compute, Clear, and Quit) have user-supplied code; hence, they are sensible places to perform unit-level testing. Some of the output events indicate functions that should occur in the Compute click event, namely, error messages when missing inputs occur. Unit testing of the Compute button should also consider invalid U.S. dollar amount entries, such as nonnumerical inputs, negative inputs, and very large inputs. Both functional and structural testing should occur at this level, and our earlier discussion (Part II) on traditional software still applies.

The best methods for performing unit-level testing are open for discussion. One possibility is to run test cases from a specially coded driver that would provide values for input data and check

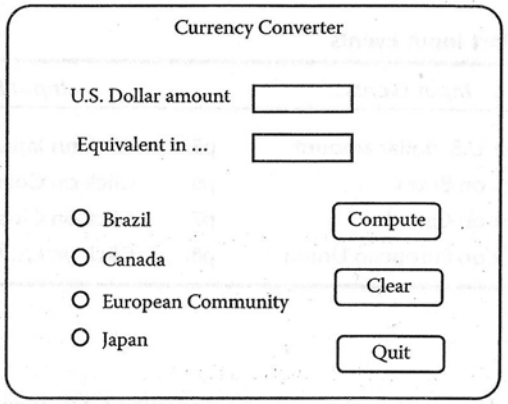



**Figure 19.1 Currency converter GUI.**

like system-level unit testing, which seems oxymoronic, but it is workable. It is system level in the sense that test case inputs are provided via system-level user input events, and test case result comparisons are based on system-level output events. This is okay for small applications, but it does beg some serious questions. For example, suppose the computation is correct, but a fault occurs in the output software. Another problem is that it will be harder to capture test execution results. Some other problems:

There will be a real tendency for ad hoc testing.
A greater potential exists for user input and observation errors.
Repeating a set of test cases is time-consuming.

On balance, unit testing with a test driver looks preferable.

Other, more subtle unit-level tests are associated with the U.S. dollar amount text box, and these are language dependent. If implemented in Visual Basic, for example, there is little need to unit test the text box. If we have a truly object-oriented implementation, we would need to verify that the inputs observed by the keyboard handler are correctly displayed on the GUI and correctly stored in the object's attributes.

## 19.3 Integration Testing for the Currency Conversion Program

Whether integration testing is appropriate for this example depends on how it is implemented. Three main choices are available for the Compute button. The first concentrates all logic in one place and simply uses the status of the option buttons as conditions in IF tests, as in the following pseudocode:

```
procedure Compute ( USDollarAmount, EquivCurrencyAmount)
dim brazilRate, canadaRate, euroRate, japanRate, USDollarAmount
As Single
If (optionBrazil)
   Then EquivCurrencyAmount = brazilRate * USDollarAmount
      If (optionCanada)
          Then EquivCurrencyAmount = canadaRate *
          USDollarAmount
          Else
              If (optionEuropeanUnion)
                  Then EquivCurrencyAmount = euroRate *
                  USDollarAmount
                  Else
                      If (optionJapan)
                          Then EquivCurrencyAmount =
                          japanRate * USDollarAmount
                          Else Output ("No country selected")
                      EndIF
              EndIF
      EndIF
EndIF
End procedure Compute
```

Because this would be thoroughly tested at the unit level, there is little need for integration testing.

A second and more object oriented choice would have methods in each option button object that send the exchange rate value for the corresponding country in response to a click event. (How the exchange rate values are set is a separate question; for now, we assume they are defined when the objects are instantiated.) Pseudocode for the instantiated objects and the revised command button procedure is as follows:

```
object optionBrazil (USDollarExchangeRate)
      private procedure senseClick
            commandCompute(USDollarExchangeRate)
      End senseClick

object optionCanada(USDollarExchangeRate)
      private procedure senseClick
            commandCompute(USDollarExchangeRate)
      End senseClick

object optionEuropeanUnion (USDollarExchangeRate)
      private procedure senseClick
            commandCompute(USDollarExchangeRate)
      End senseClick

object optionJapan (USDollarExchangeRate)
      private procedure senseClick
            commandCompute(USDollarExchangeRate)
```

```
\mount As Single
 exchangeRate * USDollarAmount
```

o test. (This is why the choice of methods as units
he integration level, and at that level, we have two
the right exchange rate values, and is the equiva-

.c Visual Basic® style. Visual Basic is now so widespread that
.. The pseudocode for a Visual Basic implementation includes a global
.. variable for the exchange rate. The event procedures for each option button
.ned value to the exchange rate variable. The result is very similar to the pure object-
.ted version. Unit testing could be replaced by simple code reading for the option button event procedures, and the testing of the commandCompute procedure is similarly trivial. Notice that in all three variations, there is little need for integration testing. This is true for small GUI applications, where a good definition of "small" is an application implemented by one person.

```
Public exchangeRate As Single

Private Sub optBrazil_Click()
     exchangeRate = 1.56
End Sub

Private Sub optCanada_Click()
     exchangeRate = 1.35
End Sub

Private Sub optEuropeanUnion_Click()
     exchangeRate = 0.93
End Sub

Private Sub optJapan_Click()
     exchangeRate = 2.04
End Sub

Private Sub cmdCompute ()
     EquivCurrencyAmount = exchangeRate *
     Val(txtUSDollarAmount.text)
End Sub
```

## 19.4 System Testing for the Currency Conversion Program

As we have seen, unit and integration testing, at least for small GUI applications, are minimally

**Table 19.1 Port Input Events**

| | *Input Events* | | *Input Events* |
|---|---|---|---|
| p1 | Enter U.S. dollar amount | p5 | Click on Japan |
| p2 | Click on Brazil | p6 | Click on Compute button |
| p3 | Click on Canada | p7 | Click on Clear button |
| p4 | Click on European Union | p8 | Click on Quit button |

list the port input and output events in the currency conversion GUI; they are nearly identical to the Visual Basic events in Table 16.1a and b. Table 19.3 lists the atomic system functions (ASFs), and Table 19.4 lists the data places that we need to make the EDPN of the GUI. The events relating to the country flags are deleted to simplify both the discussion and figures.

There is a subtle change in Table 19.4: the data place d2 is named "Country selected." It is more precise to have a data place for each country, but this really complicates the subsequent drawings. As it stands, it still represents a workable implementation.

The next step in building an EDPN description of the currency conversion GUI is to develop the EDPNs for the individual atomic system functions. These are shown in Figure 19.2. As we saw in Chapter 14, system-level threads are built up by composing atomic system functions into sequences. One such thread is shown in Figure 19.3.

We are finally in a position to describe various sets of system-level test cases for the currency conversion GUI. The lowest level is to simply exercise every atomic system function. This is a little artificial, because many atomic system functions have data outputs that are not visible system-level outputs. Even worse, the possibility always exists that an atomic system function has no port outputs, as with s1: store U.S. dollar amount. At the system testing level, we cannot tell if an amount is correctly stored, although we can look at the screen and see that the correct amount has been entered.

**Table 19.2 Port Output Events**

| | *Output Events* | | *Output Events* |
|---|---|---|---|
| p9 | Display U.S. dollar amount | p18 | Indicate Japan |
| p10 | Display Brazilian reals | p19 | Reset Canada, EU, and Japan |
| p11 | Display Canadian dollars | p20 | Reset Brazil, EU, and Japan |
| p12 | Display EU euros | p21 | Reset Brazil, Canada, and Japan |
| p13 | Display Japanese yen | p22 | Reset Brazil, Canada, and EU |
| p14 | Display ellipses | p23 | Reset U.S. dollar amount |
| p15 | Indicate Brazil | p24 | Reset equivalent currency amount |
| p16 | Indicate Canada | p25 | End application |

**Table 19.3 Atomic System Functions**

| | *Atomic System Functions* | | *Atomic System Functions* |
|---|---|---|---|
| s1 | Store U.S. dollar amount | s5 | Sense click on Japan |
| s2 | Sense click on Brazil | s6 | Sense click on Compute button |
| s3 | Sense click on Canada | s7 | Sense click on Clear button |
| s4 | Sense click on EU | s8 | Sense click on Quit button |

The next level of system testing is to exercise a suitable set of threads. This begs the question of what constitutes suitable. Here are some easy possibilities; exercise a set of threads that:

Use every atomic system function
Use every port input
Use every port output

Beyond this, we can look at a directed graph of atomic system functions, as shown in Figure 19.4. This is only a partial graph; it shows mainline behavior. The thread <s1, s4, s6, s7> (the one in Figure 19.3) is one of 12 paths. Four of these end with the Clear button clicked; the other eight end with the Quit button. Consider the set T of threads {T1, T2, T3, T4}, where their ASF sequences are listed next:

T1 = <s1, s4, s6, s7>
T2 = <s1, s2, s6, s7>
T3 = <s3, s1, s6, s8>
T4 = <s5, s1, s7, s8>

The threads in set T have the following coverages:

Every atomic system function
Every port input
Every port output

As such, set T constitutes a reasonable minimum level of system testing for the currency conversion GUI. We could go into more detail by exploring some next-level user behavior. The thread T5 = <s1, s2, s6, s3, s6, s4, s6, s5, s6, s7, s8> is a good example, where the user converts a U.S. dollar amount to each of the four currencies, then clears the screen and quits. We could also explore some of the many abnormal user behavior sequences (the sequences are abnormal, not the users), such as T6 = <s1, s2, s3, s4, s5, s6, s7, s8>. In T6, the user changes his or her mind about which currency

**Table 19.4 Data Places**

| | *Data Places* | | *Data Places* |
|---|---|---|---|
| d1 | U.S. dollar amount entered | d2 | Country selected |

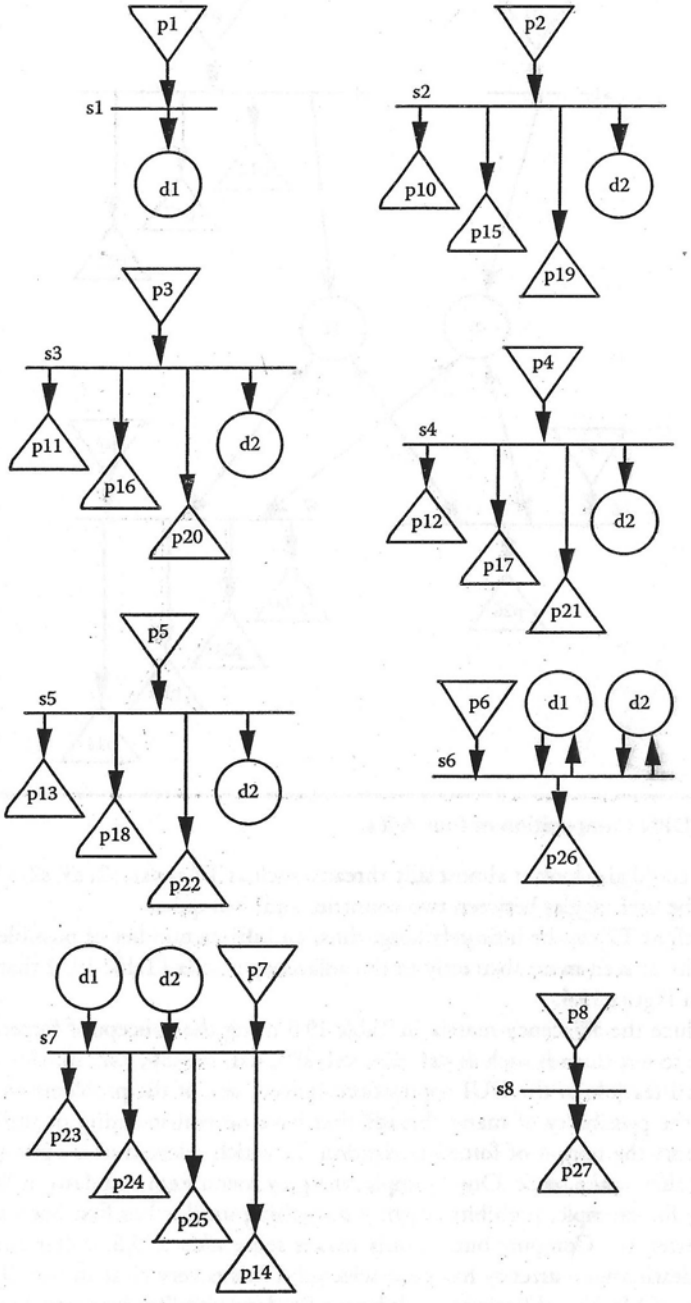



**Figure 19.2 EDPNs of atomic system functions.**

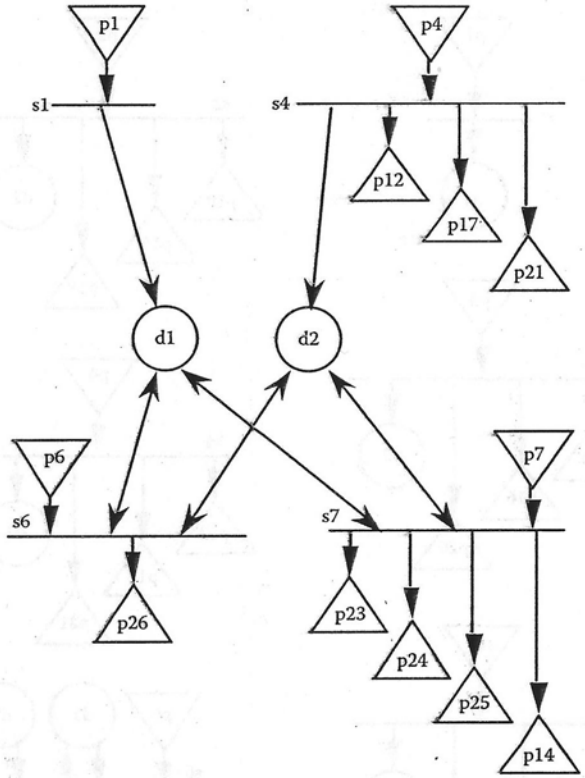



**Figure 19.3 EDPN Composition of four ASFs.**

to convert. We could also look at almost silly threads, such as T7 = <s1, s2, s3, s2, s3, s2, s3, s2, s3, s8>, in which the user toggles between two countries and then quits.

Threads such as T7 can be infinitely long; thus, an infinite number of possible threads occur in this GUI. This is seen more abstractly in the adjacency matrix (Table 19.5) that completes the partial graph in Figure 19.4.

We can reduce the adjacency matrix in Table 19.5 using the concept of forced navigation. It makes no sense to test threads such as <s1, s7>, <s1, s8>, <s1, s6, s8>, <s2, s6, s8>, and many others. These are all feasible in the GUI application; indeed, one of the problems with GUI design and testing is the possibility of many threads that have no real meaning or utility. Most GUI languages support the notion of forced navigation, in which selected user options only become available when they make sense. One example: the gray menu item standard in Windows applications, which, for example, prohibits copying something until it has first been selected. In the currency converter, the Compute button only makes sense after a U.S. dollar amount has been entered and a destination currency has been selected. (This is very clear in the EDPN of atomic system function s6.) In Visual Basic controls have a Boolean visibility property. Forced navigation in the currency converter could be implemented by making the Compute button invisible until its prerequisite events have occurred. Although forced navigation eliminates many curious threads, it

**Table 19.5 Adjacency Matrix of GUI Atomic System Functions**

| | *s1* | *s2* | *s3* | *s4* | *s5* | *s6* | *s7* | *s8* |
|---|---|---|---|---|---|---|---|---|
| s1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| s2 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| s3 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| s4 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| s5 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| s6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| s7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| s8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

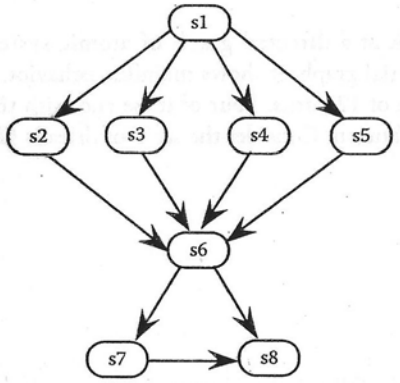



**Figure 19.4 Directed graph of intended atomic system function sequences.**

**Table 19.6 Adjacency Matrix with Forced Navigation**

| | *s1* | *s2* | *s3* | *s4* | *s5* | *s6* | *s7* | *s8* |
|---|---|---|---|---|---|---|---|---|
| *s1* | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| *s2* | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| *s3* | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| *s4* | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| *s5* | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| *s6* | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| *s7* | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| *s8* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Column *s1* forces the entry of a U.S. dollar amount before a destination currency can be selected. The rows *s2*, *s3*, *s4*, *s5*, and *s6* show that the Compute button can be selected only after a destination currency has been selected. Just these few changes greatly reduce the number of feasible paths. Mathematically, the presence of loops means that there is an infinite number of possible paths, but a tester would not test the infinite number of duplications anyway.

## Exercises

1. 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 pseudocode definition stands, a user could enter a U.S. dollar amount and then click on the Compute button without selecting a country. Similarly, the user could select a country and then click on the Compute button without inputting a dollar amount. In an object-oriented application, we can control this by being careful about when we instantiate objects. Revise the GUI class pseudocode to prevent these two errors.
2. Does it make sense to test threads such as <s1, s7>, <s1, s8>, <s1, s6, s8>, and <s2, s6, s8> in the currency converter?

# Chapter 20

# Object-Oriented System Testing

System testing is (or should be) independent of system implementation. A system tester does not really need to know if the implementation is in procedural or object-oriented code. As we saw in Chapter 14, the primitives of system testing are port input and output events, and we know how to express system-level threads as event-driven Petri nets (EDPNs). The issue is how to identify threads to be used as test cases. In Chapter 14, we used the requirements specification models, particularly the behavioral models, as the basis of thread test case identification. We also discussed pseudostructural coverage metrics in terms of the underlying behavioral models. In a sense, this chapter is very object oriented: we inherit many ideas from system testing of traditional software. The only real difference in this chapter is that we presume the system has been defined and refined with the Unified Modeling Language (UML). One emphasis, then, is finding system-level thread test cases from standard UML models.

## 20.1 Currency Converter UML Description

We will use the currency converter application as an example for system testing. Because the Unified Modeling Language from the Object Management Group is now widely accepted, we will use a rather complete UML description, in the style of Larman (1998). The terminology and UML content generally follow the Larman UML style, with the addition of pre- and postconditions in expanded essential use cases.

### *20.1.1 Problem Statement*

The currency converter application converts U.S. dollars to any of four currencies: Brazilian real, Canadian dollars, European Union euros, and Japanese yen. The user can revise inputs and perform repeated currency conversion

**Table 20.1 System Functions for the Currency Converter Application**

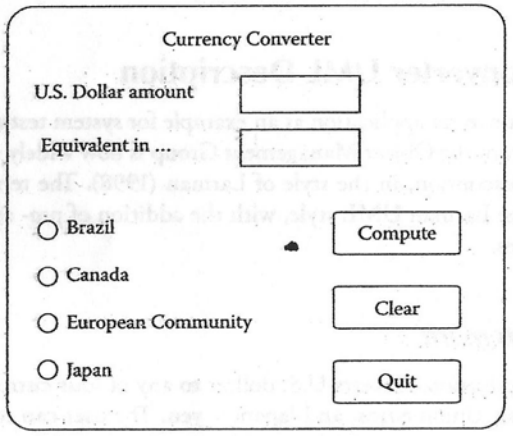
| *Reference No.* | *Function* | *Category* |
|---|---|---|
| R1 | Start application | Evident |
| R2 | End application | Evident |
| R3 | Input U.S. dollar amount | Evident |
| R4 | Select country | Evident |
| R5 | Perform conversion calculation | Evident |
| R6 | Clear user inputs and program outputs | Evident |
| R7 | Maintain exclusive-or relationship among countries | Hidden |
| R8 | Display country flag images | Frill |

### 20.1.2 System Functions

In the first step, sometimes called project inception, the customer/user describes the application in very general terms. This might take the form of user stories, which are precursors to use cases. From these, three types of system functions are identified: evident, hidden, and frill. Evident functions are the obvious ones. Hidden functions might not be discovered immediately, and frills are the "bells and whistles" that so often occur. Table 20.1 lists the system functions for the currency converter application.

### 20.1.3 Presentation Layer

Pictures are still worth a thousand words. The third step in Larman's approach is to sketch the user interface; our version is in Figure 20.1. This much information can support a customer walk-through to demonstrate that the system functions identified can be supported by the interface.




### 20.1.4 High-Level Use Cases

The use case development begins with a very high level view. Notice, as the succeeding levels of use cases are elaborated, much of the early information is retained. It is convenient to have a short, structured naming convention for the various levels of use cases. Here, for example, HLUC refers to high-level use case. Very few details are provided in a high-level use case; they are insufficient for test case identification. The main point of high-level use cases is that they capture a narrative description of something that happens in the system to be built.

| | |
|---|---|
| HLUC 1 | Start application |
| Actor(s) | User |
| Type | Primary |
| Description | The user starts the currency conversion application in Windows® |

| | |
|---|---|
| HLUC 2 | End application |
| Actor(s) | User |
| Type | Primary |
| Description | The user ends the currency conversion application in Windows |

| | |
|---|---|
| HLUC 3 | Convert dollars |
| Actor(s) | User |
| Type | Primary |
| Description | The user inputs a U.S. dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country |

| | |
|---|---|
| HLUC 4 | Revise inputs |
| Actor(s) | User |
| Type | Secondary |
| Description | The user resets inputs to begin a new transaction |

### 20.1.5 Essential Use Cases

Essential use cases add "actor" and "system" events to a high-level use case. Actors in UML are sources of system-level inputs (i.e., port input events). Actors can be people, devices, adjacent systems, or abstractions such as time. The numbering of actor actions and system responses (port output events) shows their approximate sequences in time. In EUC 3, for example,

| | | |
|---|---|---|
| EUC 1 | Start application | |
| Actor(s) | User | |
| Type | Primary | |
| Description | The user starts the currency conversion application in Windows | |
| Sequence | Actor action | System response |
| | 1. The user starts the application, either with a Run ... command or by double-clicking the application icon | 2. The currency conversion application GUI appears on the monitor and is ready for user input |

| | | |
|---|---|---|
| EUC 2 | End application | |
| Actor(s) | User | |
| Type | Primary | |
| Description | The user ends the currency conversion application in Windows | |
| Sequence | Actor action | System response |
| | 1. The user ends the application, either by clicking a Quit button or by closing the window | 2. The currency conversion application GUI disappears from the monitor |

| | | |
|---|---|---|
| EUC 3 | Convert dollars | |
| Actor(s) | User | |
| Type | Primary | |
| Description | The user inputs a U.S. dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country | |
| Sequence | Actor action | System response |
| | 1. The user enters a dollar amount | 2. The dollar amount is displayed on the GUI |
| | 3. The user selects a country | 4. The name of the country's currency is displayed |
| | | 5. The flag of the country is displayed |
| | 6. The user requests a conversion calculation | 7. The equivalent currency amount is displayed |

| | | |
|---|---|---|
| EUC 4 | Revise inputs | |
| Actor(s) | User | |
| Type | Secondary | |
| Description | The user resets inputs to begin a new transaction | |
| Sequence | Actor action | System response |
| | 1. The user enters a dollar amount | 2. The dollar amount is displayed on the GUI |
| | 3. The user selects a country | 4. The name of the country's currency is displayed |
| | | 5. The flag of the country is displayed |
| | 6. The user cancels the inputs | 7. The name of the country's currency is removed |
| | | 8. The flag of the country is no longer visible |

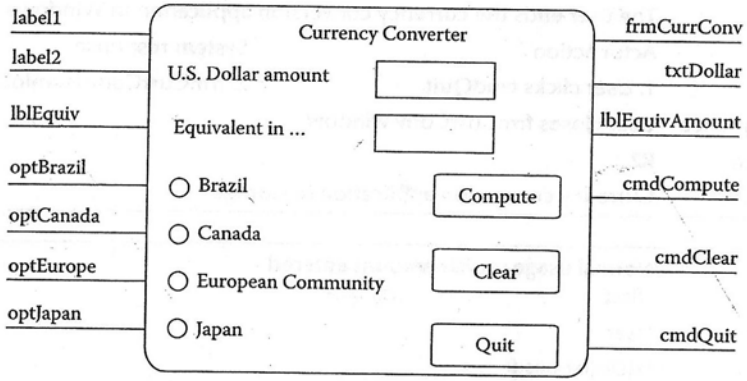
35609

### 20.1.6 Detailed GUI Definition

Once a set of essential use cases has been identified, the graphical user interface (GUI) is fleshed out with design-level detail. Here, we implement the currency converter in Visual Basic and follow the recommended (Zak, 2001) naming conventions for Visual Basic controls, as shown in Figure 20.2. (For readers not familiar with Visual Basic, this design uses four types of controls: text boxes for input, labels for output, option buttons to indicate choices, and command buttons to control the execution of the application.) These controls will be referred to in the expanded essential use cases. As in Chapter 19, we omit the country flag images.

### 20.1.7 Expanded Essential Use Cases

The expanded essential use cases (EEUCs) are the penultimate refinement of the high-level use cases. Here, we add pre- and postcondition information (not part of the Larman flavor), information about alternative sequences of events, and a cross-reference to the system functions identified



very early in the process. The other expansion is that more use cases are identified and added at this point. This is a normal part of any specification and design process: more detailed views provide more detailed insights. Note that the numeric tracing across levels of use cases is lost at this point.

The pre- and post-conditions warrant some additional comment. We are only interested in conditions that directly pertain to the expanded essential use case defined. We could always add preconditions such as "power is on," "computer is running under Windows®," and so on. As we saw in Chapter 15, threads (when expressed as EDPNs) interact via data places. The recommended practice is to record pre- and postconditions that will likely correspond to data places when the threads corresponding to expanded essential use cases are expressed as EDPNs.

| | | |
|---|---|---|
| EEUC 1 | Start application | |
| Actor(s) | User | |
| Preconditions | Currency conversion application in (disk) storage | |
| Type | Primary | |
| Description | The user starts the currency conversion application in Windows | |
| Sequence | Actor action | System response |
| | 1. User double-clicks currency conversion application icon | 2. "frmCurrConv" appears on screen |
| Alternative sequence | User opens currency conversion application with the Windows Run command | |
| Cross-reference | R1 | |
| Postconditions | Currency conversion application is in memory; txtDollar has focus | |

| | | |
|---|---|---|
| EEUC 2 | End application | |
| Actor(s) | User | |
| Preconditions | frmCurrConv is in run mode | |
| Type | Primary | |
| Description | The user ends the currency conversion application in Windows | |
| Sequence | Actor action | System response |
| | 1. User clicks cmdQuit | 2. frmCurrConv is unloaded |
| Alternative sequence | User closes frmCurrConv window | |
| Cross-reference | R2 | |
| Postconditions | Currency conversion application in storage | |

| | | |
|---|---|---|
| EEUC 3 | Normal usage (dollar amount entered first) | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Description | The user inputs a U.S. dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country | |
| Sequence | Actor action | System response |
| | 1. User enters U.S. dollar amount on keyboard | 2. Dollar amount appears in txtDollar |
| | 3. User clicks on a country button | 4. Country currency name appears in lblEquiv |
| | 5. User clicks cmdCompute button | 6. Computed equivalent amount appears in lblEqAmount |
| Alternative sequence | Actions 1 and 3 can be reversed, and consequently, responses 2 and 4 will be reversed | |
| Cross-references | R3, R4, R5 | |
| Postconditions | cmdClear has focus | |

| | | |
|---|---|---|
| EEUC 4 | Repeated conversions, same country | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Type | Primary | |
| Description | The user inputs a U.S. dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country | |
| Sequence | Actor action | System response |
| | 1. User enters U.S. dollar amount on keyboard | 2. Dollar amount appears in txtDollar |
| | 3. User clicks on a country button | 4. Country currency name appears in lblEquiv |
| | 5. User clicks cmdCompute button | 6. Computed equivalent amount appears in lblEqAmount |
| | 7. User clicks on txtDollar | 8. txtDollar has focus |
| | 9. User enters different U.S. dollar amount on keyboard | 10. Dollar amount appears in txtDollar |
| | 11. User clicks cmdCompute button | 12. Computed equivalent amount appears in lblEqAmount |
| Alternative sequence | Actions 1 and 3 can be reversed, and consequently, responses 2 and 4 will be reversed; actions 7, 9, and 11 can be repeated indefinitely, with corresponding responses 8, 10, and 12 | |
| Cross-references | R3, R4, R5 | |
| Postconditions | cmdClear has focus | |

| | |
|---|---|
| EEUC 5 | Repeated conversions, same dollar amount |

| | | |
|---|---|---|
| Preconditions | txtDollar has focus | |
| Type | Primary | |
| Description | The user inputs a U.S. dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country | |
| Sequence | Actor action | System response |
| | 1. User enters U.S. dollar amount on keyboard | 2. Dollar amount appears in txtDollar |
| | 3. User clicks on a country button | 4. Country currency name appears in lblEquiv |
| | 5. User clicks cmdCompute button | 6. Computed equivalent amount appears in lblEqAmount |
| | 7. User clicks on a different country button | 8. New country currency name appears in lblEquiv |
| | | 9. Previously selected option button is reset |
| | | 10. Currently selected option button is set |
| | 11. User clicks cmdCompute button | 12. Computed equivalent amount appears in lblEqAmount |
| Alternative sequence | Actions 1 and 3 can be reversed, and consequently, responses 2 and 4 will be reversed; actions 7 and 11 can be repeated indefinitely, with corresponding responses 8, 9, 10, and 12 | |
| Cross-references | R3, R4, R5, R7 | |
| Postconditions | cmdClear has focus | |

| | | |
|---|---|---|
| EEUC 6 | Revise inputs | |
| Actor(s) | User | |
| Preconditions | txtDollar has a nonblank amount OR a country has been selected | |
| Type | Secondary | |
| Description | The user resets inputs either to begin a new transaction or to revise existing inputs | |
| Sequence | Actor action | System response |
| | 1. The user enters a dollar amount on the keyboard | 2. The dollar amount is displayed in txtDollar |
| | 3. User clicks on a country button | 4. Country currency name appears in lblEquiv |
| | 5. The user clicks on the cmdClear button | 6. txtDollar shows the null entry |
| | | 7. The selected country option button is reset. |
| | 8. The user enters a new dollar | 9. The new dollar amount is |
| | 10. User clicks on a different country button | 11. Country currency name appears in lblEquiv |
| | | 12. The selected country option button is set |
| | 13. The user clicks on the cmdClear button | 14. txtDollar shows the null entry |
| | | 15. The selected country option button is reset |
| Alternative sequence | Actions 8, 10, and 13 can be repeated indefinitely; responses 9, 11, 12, 14, and 15 will recur | |
| Cross-references | R3, R4, R6 | |
| Postconditions | txtDollar has the focus | |

| | | |
|---|---|---|
| EEUC 7 | Abnormal case; no country selected | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Type | Hidden | |
| Description | User enters a dollar amount and clicks on the cmdCompute button without selecting a country | |
| Sequence | Actor action | System response |
| | 1. User enters U.S. dollar amount on keyboard | 2. Dollar amount appears in txtDollar |
| | 3. User clicks cmdCompute button | 4. A message box appears with the caption "must select a country" |
| | 5. User closes message box | 6. Message box no longer visible |
| Alternative sequence | n/a | |
| Cross-references | R3, R5 | |
| Postconditions | txtDollar has focus | |

| | | |
|---|---|---|
| EEUC 8 | Abnormal case; no dollar amount entered | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Type | Hidden | |
| Description | User selects a country and clicks on the cmdCompute button without entering a dollar amount | |
| Sequence | Actor action | System response |
| | 1. User clicks on a country button | 2. Country currency name appears in lblEquiv |
| | 3. User clicks cmdCompute button | 4. A message box appears with the caption "must enter a |

| | | |
|---|---|---|
| | 5. User closes message box | 6. Message box no longer visible |
| Alternative sequence | n/a | |
| Cross-references | R3, R5 | |
| Postconditions | txtDollar has focus | |

| | | |
|---|---|---|
| EEUC 9 | Abnormal case; no dollar amount entered and no country selected | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Type | Hidden | |
| Description | User clicks on the cmdCompute button without entering a dollar amount and without selecting a country | |
| Sequence | Actor action | System response |
| | 1. User clicks cmdCompute button | 2. A message box appears with the caption "must enter a dollar amount and select a country" |
| | 3.User closes message box | 4. Message box no longer visible |
| Alternative sequence | n/a | |
| Cross-reference | R5 | |
| Postconditions | txtDollar has focus | |

### 20.1.8 Real Use Cases

In Larman's terms, real use cases are only slightly different from the expanded essential use cases. Phrases such as "enter a U.S dollar amount" must be replaced by the more specific "enter 125 in txtDollar." Similarly, "select a country" would be replaced by "click on the optBrazil button." In the interest of space (and reduced reader boredom), real use cases are omitted. Note that system-level test cases could be mechanically derived from real use cases.

## 20.2 UML-Based System Testing

Our formulation lets us be very specific about system-level testing; there are at least four identifiable levels with corresponding coverage metrics for GUI applications. Two of these are naturally dependent on the UML specification; we saw the other two in Chapter 19.

The first level is to test the system functions given as the first step in Larman's UML approach. These are cross-referenced in the extended essential use cases, so we can easily build an incidence matrix such as Table 20.2.

Examining the incidence matrix, we can see several possible ways to cover the seven system functions. One way would be to derive test cases from real use cases that correspond to extended essential use cases 1, 2, 5, and 6. These will need to be real use cases as opposed to the expanded

**Table 20.2 Use Case Incidence with System Functions**

| *EEUC* | *R1* | *R2* | *R3* | *R4* | *R5* | *R6* | *R7* |
|---|---|---|---|---|---|---|---|
| 1 | X | – | – | – | – | – | – |
| 2 | – | X | – | – | – | – | – |
| 3 | – | – | X | X | X | – | – |
| 4 | – | – | X | X | X | – | – |
| 5 | – | – | X | X | X | – | X |
| 6 | – | – | X | X | – | X | X |
| 7 | – | – | X | – | X | – | – |
| 8 | – | – | X | – | X | – | – |
| 9 | – | – | – | – | X | – | – |

higher-level statements such as "click on a country button" and "enter a dollar amount." Deriving system test cases from real use cases is mechanical: the use case preconditions are the test case preconditions, and the sequences of actor actions and system responses map directly into sequences of user input events and system output events. The set of extended essential use cases 1, 2, 5, and 6 is a nice example of a set of regression test cases; taken together, they cover all seven system functions.

The second level is to develop test cases from all of the real use cases. Assuming that the customer approved of the original expanded essential use cases, this is the minimally acceptable level of system test coverage. Here is a sample system-level test case derived from the real use case based on extended essential use case EEUC 3. (Assume the exchange rate for one euro is U.S.$1.38.)

| | | |
|---|---|---|
| RUC 3 | Normal usage (dollar amount entered first) | |
| Actor(s) | User | |
| Preconditions | txtDollar has focus | |
| Type | Primary | |
| Description | The user inputs a U.S. $10 amount and selects the European Community; the application computes and displays the equivalent: 7.25 euros | |
| Sequence | Actor action | System response |
| | 1. User enters 10 on the keyboard | 2. "10" appears in txtDollar |
| | 3. User clicks on the European Community button | 4. "Euros" appears in lblEquiv |
| | 5. User clicks cmdCompute button | 6. "7.25" appears in lblEqAmount |
| Alternative sequence | Actions 1 and 3 can be reversed, and consequently, responses 2 and 4 will be reversed | |
| Cross-references | R3, R4, R5 | |

| | | |
|---|---|---|
| SysTC 3 | Normal usage (dollar amount entered first) | |
| Test operator | Paul Jorgensen | |
| Preconditions | txtDollar has focus | |
| Test operator sequence | Tester inputs | Expected system response |
| | 1. Enters 10 on the keyboard | 2. Observe "10" appears in txtDollar |
| | 3. Click on the European Community button | 4. Observe "euros" appears in lblEquiv |
| | 5. Clicks cmdCompute button | 6. Observe "7.25" appears in lblEqAmount |
| Postconditions | cmdClear has focus | |
| Test result | Pass/fail | |
| Date run | August 8, 2007 | |

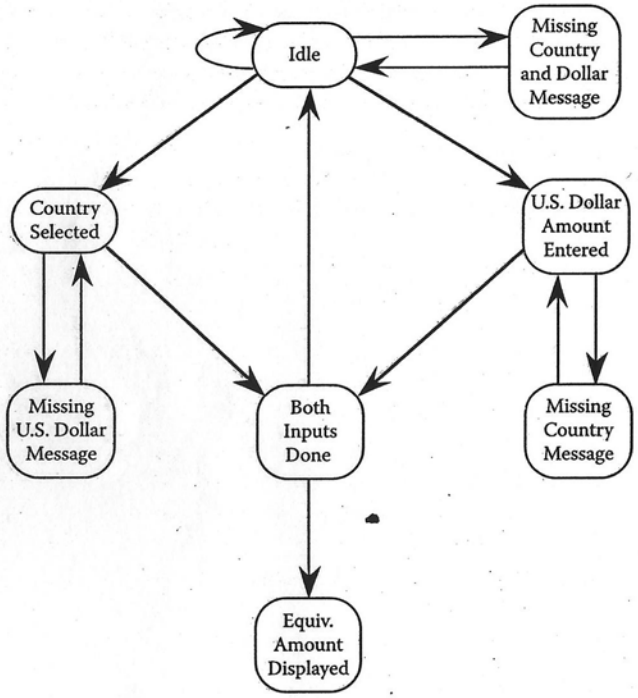



**Table 20.3 Test Cases Derived from a Finite State Machine**

| *State* | *TC1* | *TC2* | *TC3* | *TC4* | *TC5* | *TC6* | *TC7* | *TC8* | *TC9* |
|---|---|---|---|---|---|---|---|---|---|
| Idle | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1, 3 |
| Missing country and dollar message | | | | | | | | 2 | 2 |
| Country selected | | 2 | 2, 4 | | | 2 | | | 4, 6 |
| U.S. dollar amount entered | | | | 2 | 2, 4 | | 2 | | |
| Missing U.S. dollar message | | | 3 | | | | | | 5 |
| Both inputs done | | 3 | 5 | 3 | 5 | 3 | 3 | | 7 |
| Missing country message | | | | | 3 | | | | |
| Equivalent amount displayed | | 4 | 6 | 4 | 6 | | | | |
| Idle | 2 | 5 | 7 | 5 | 7 | 1 | 1 | 1 | 1 |

The third level is to derive test cases from the finite state machines derived from a finite state machine description of the external appearance of the GUI, as shown in Figure 20.3 (repeated from Chapter 19). A test case from this formulation is a circuit (a path in which the start node is the end node, usually the idle state). Nine such test cases are shown in Table 20.3, where the numbers show the sequence in which the states are traversed by the test case. Many other cases exist, but this shows how to identify them.

The fourth level is to derive test cases from state-based event tables (see Chapter 19, Table 19.1 and Figure 19.2). This would have to be repeated for each state. We might call this the exhaustive level, because it exercises every possible event for each state. It is not truly exhaustive, however, because we have not tested all sequences of events across states. The other problem is that it is an extremely detailed view of system testing that is likely very redundant with integration- and even unit-level test cases.

## 20.3 StateChart-Based System Testing

A caveat is required here. StateCharts are a fine basis for system testing; we saw this in Chapter 19. The problem is that StateCharts are prescribed to be at the class level in UML. There is no easy way to compose StateCharts of several classes to get a system-level StateChart. A possible workaround is to translate each class-level StateChart into a set of EDPNs, and then compose the EDPNs as we did in Chapter 15.

## References

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, Upper Saddle River, NJ, 1998.

Zak, D., *Programming with Microsoft Visual Basic 6.0: Enhanced Edition*, Course Technology, Inc., Boston