UNIT-V

Object-Oriented Integration Testing

MADHESWARI.K
AP/CSE
SSNCE

Topics to be covered?

- **Object-Oriented Integration testing**

  - ***UML Support for Integration Testing***
  - ***MM-Paths for Object-Oriented Software***
  - ***A Framework for Object-Oriented Data Flow Testing***
    - *Event-/Message-Driven Petri Nets*
    - *Inheritance-Induced Data Flow*
    - *Message-Induced Data Flow*
    - *Slices?*

Object-Oriented Integration testing

- As with traditional procedural software, object oriented integration testing presumes complete unit-level testing.
- Both unit choices have implications for object-oriented integration testing.

Method as Unit

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.

Object-Oriented Integration testing

class as Unit

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.

UML Support for Integration Testing

- In UML-defined, object-oriented software, collaboration and sequence diagrams are the basis for integration testing.
- Once this level is defined, integration-level details are added
- A collaboration diagram shows (some of) the message traffic among classes.
- collaboration diagram supports both the pairwise and neighborhood approaches to integration testing.

Pairwise Integration

- 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 we saw pairwise integration of procedural units to be.

UML Support for Integration Testing

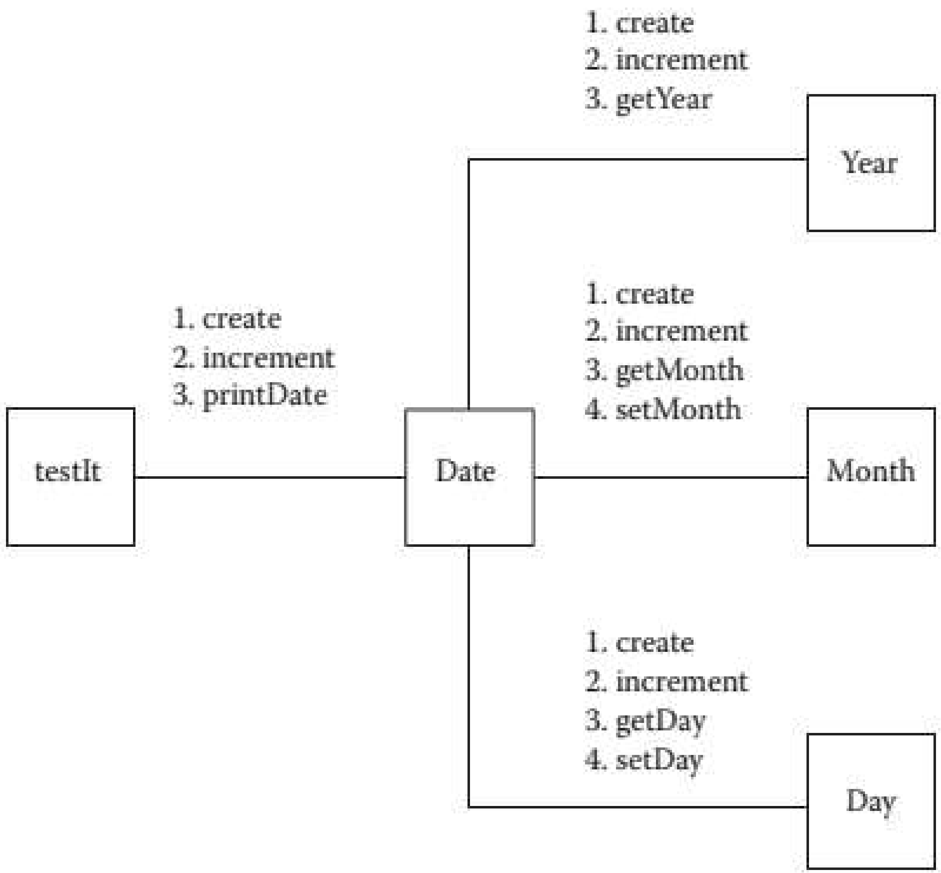
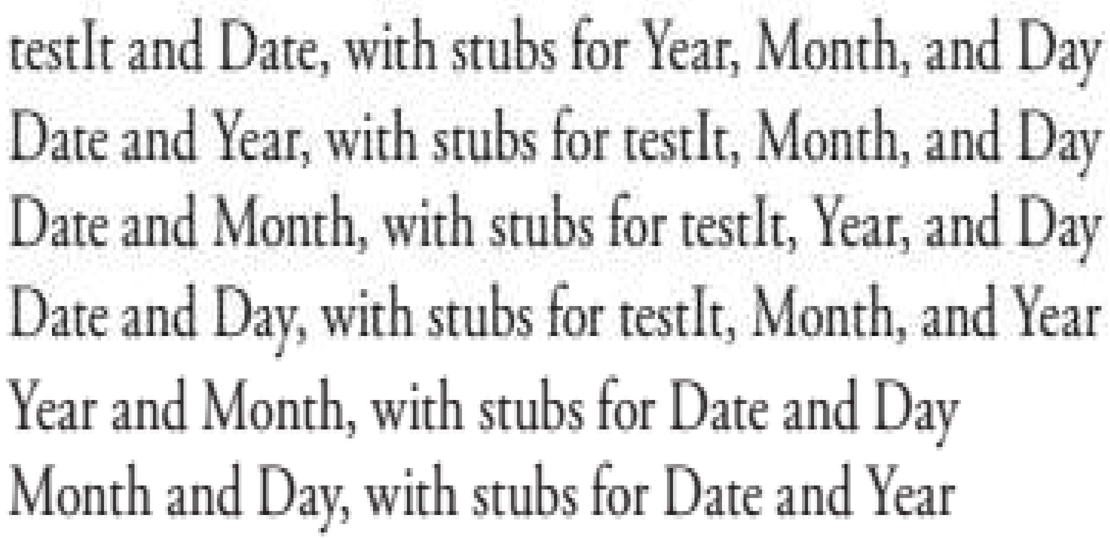


Figure 15.9 Collaboration diagram for ooCalendar.

UML Support for Integration Testing

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

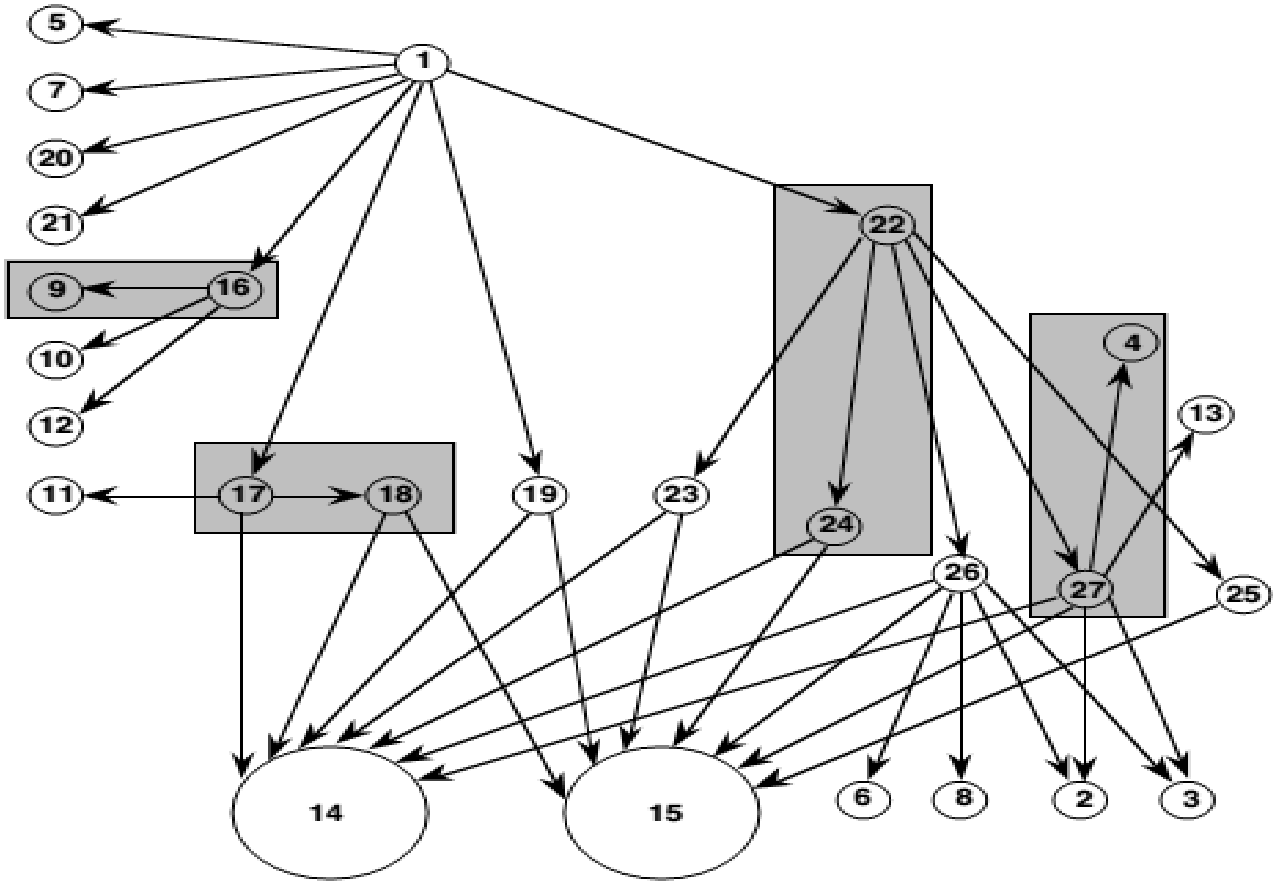
Date and Day, with stubs for testIt, Month, and Year

Year and Month, with stubs for Date and Day

Month and Day, with stubs for Date and Year

UML Support for Integration Testing

Pair-wise integration example



UML Support for Integration Testing

Advantages of pair-wise integration testing

- Eliminate need for developing stubs / drivers
- Use actual code instead of stubs/drivers

UML Support for Integration Testing

Neighborhood integration testing

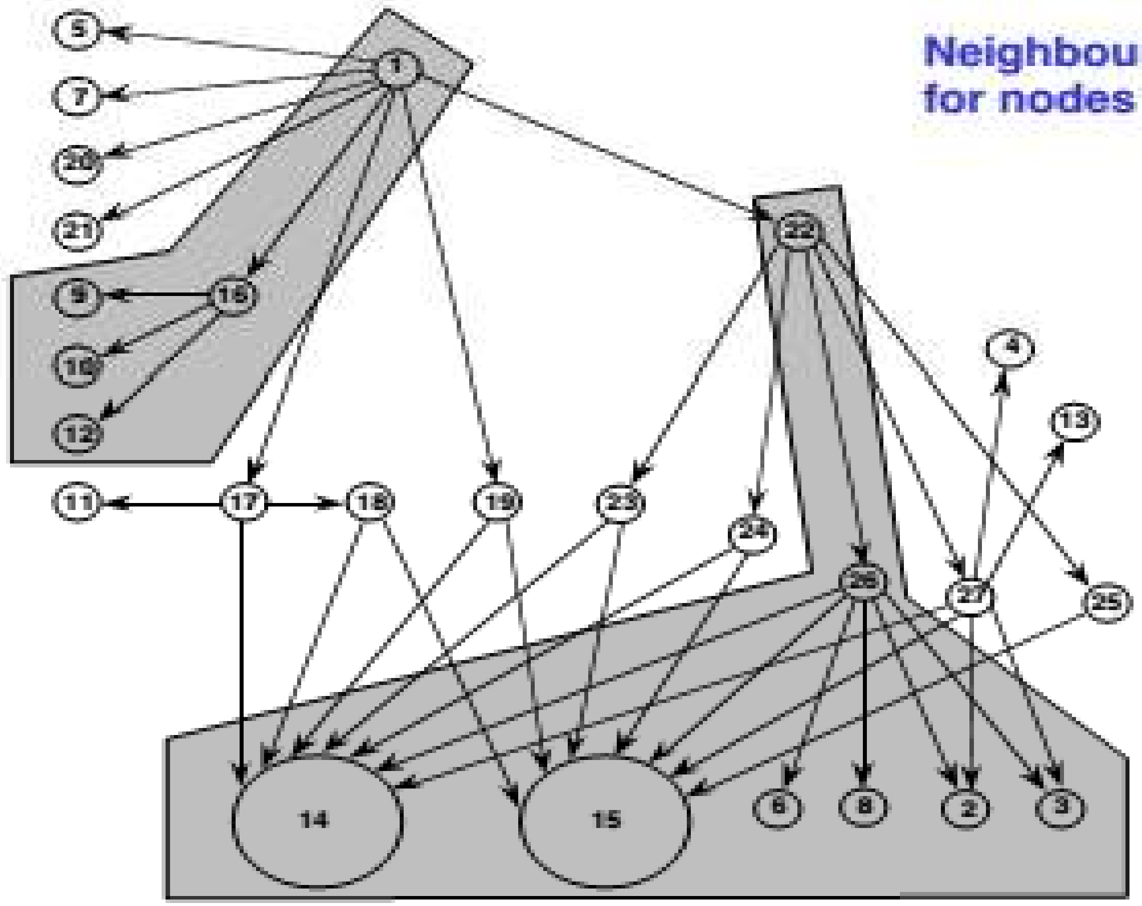
- start with the ultracenter and the neighborhood of nodes one edge away, then add the nodes two edges away, and so on.
- The set of nodes that are one edge away from the given node is called neighborhood nodes (first level of testing)
- The set of nodes that are two edges away from the given node is called neighborhood nodes (second level of testing)

Advantges

- 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.

UML Support for Integration Testing

Neighborhood integration testing - example



Neighbourhoods
for nodes 16 & 26

UML Support for Integration Testing

A sequence diagram traces an execution-time path through a collaboration diagram. (In UML, a sequence diagram has two levels: at the system/use case level and at the class interaction level.) Thick, vertical lines represent either a class or an instance of a class, and the arrows are labeled with the messages sent by (instances of) the classes in their time order. The portion of the ooCalendar application that prints out the new date is shown as a sequence diagram in Figure 15.10.

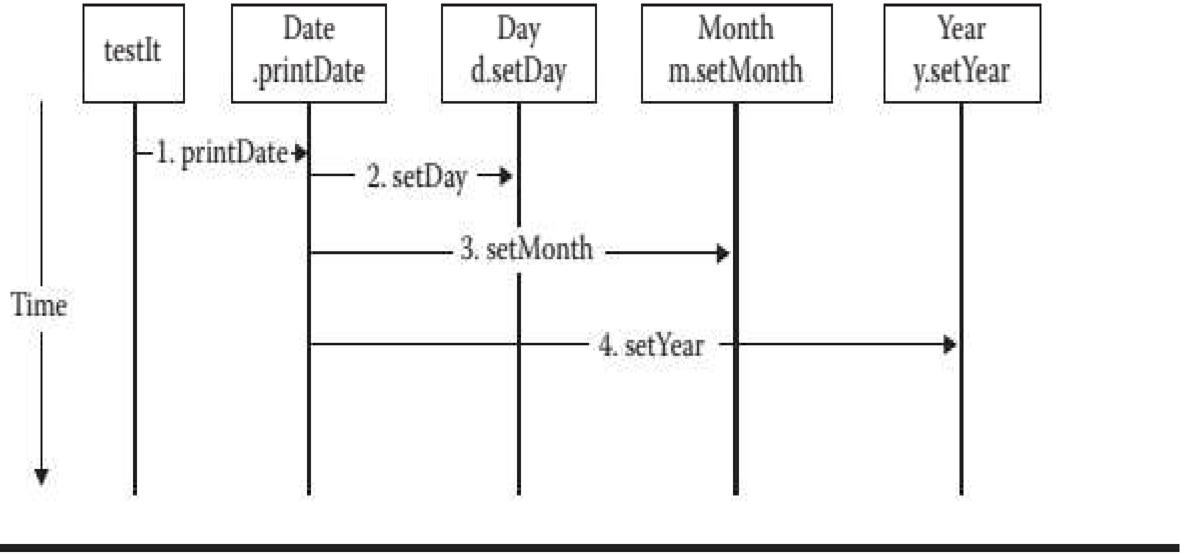
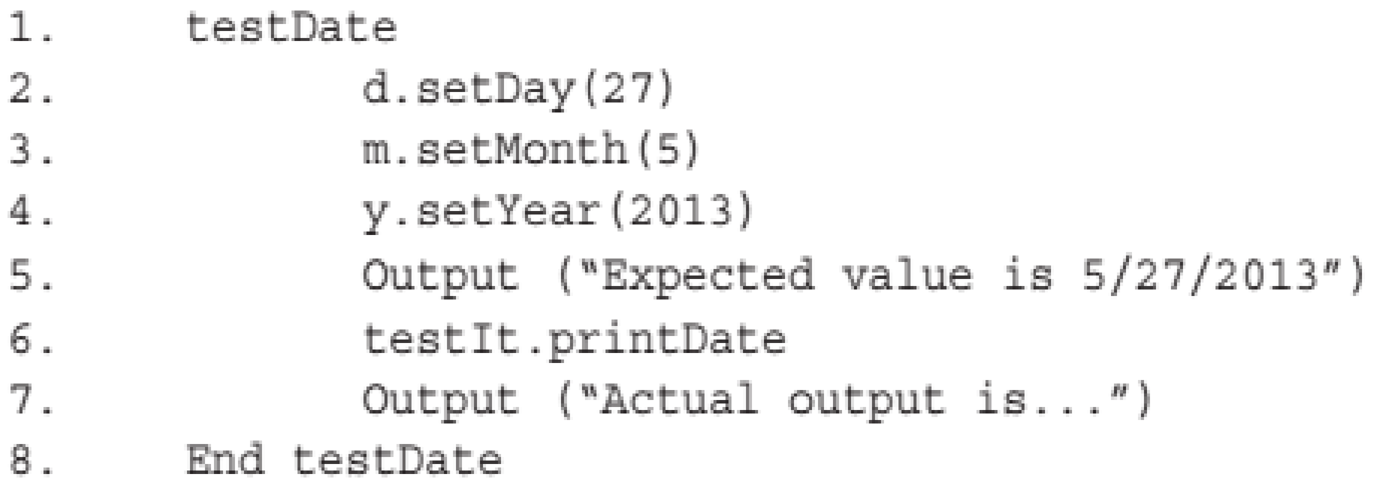


Figure 15.10 Sequence diagram for printDate.

UML Support for Integration Testing

An actual test for this sequence diagram would have pseudocode similar to this:



```
testDate
    d.setDay(27)
    m.setMonth(5)
    y.setYear(2013)
    Output ("Expected value is 5/27/2013")
    testIt.printDate
    Output ("Actual output is...")
End testDate
```

UML Support for Integration Testing

- Statements 2, 3, and 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.

UML Support for Integration Testing

Collaboration diagram and sequence diagram is suboptimal for Integration testing Why?

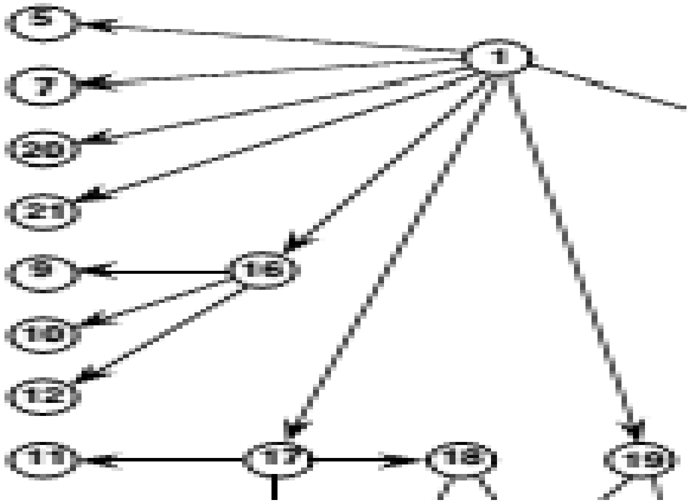
- Using collaboration diagrams or sequence diagrams as a basis for object-oriented integration testing is suboptimal.
- Collaboration diagrams force a pairwise approach to integration testing.
- Sequence diagrams are a little better, but somehow the integration tester needs all the sequence diagrams that pertain to a particular set of units to be integrated.

UML Support for Integration Testing

A third, non-UML strategy is to use the call graph

call graph Is a directed, labeled graph

☐ Vertices are methods
☐ A directed edge joins calling vertex to the called vertex



nodes in a call graph can be either procedural units or object-oriented methods. For object-oriented integration, edges represent messages from one method to another

MM-Paths for Object-Oriented Software

MM-Paths for Object-Oriented Software

Definition

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

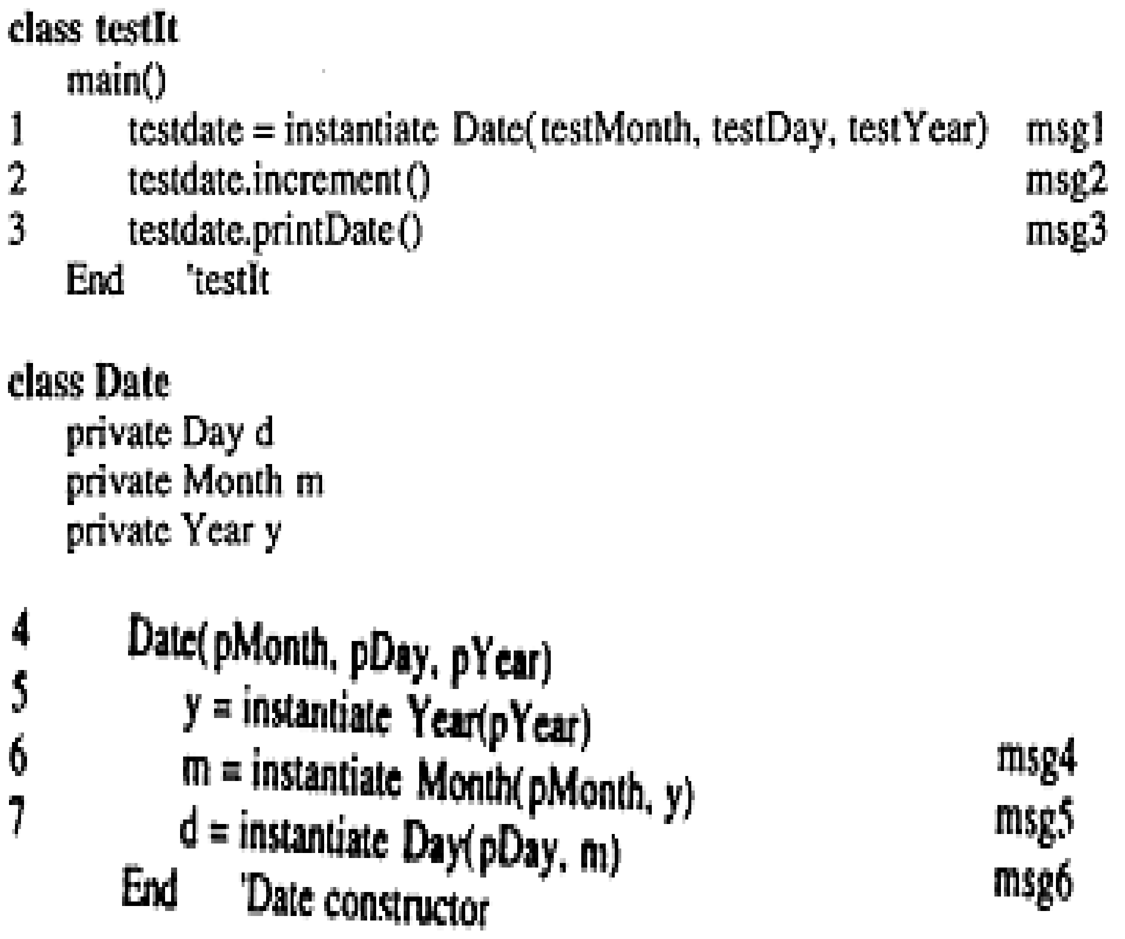
M-Method
M-message
Method/Message path (MM-path)

“message” to refer to the invocation among separate units (modules)

An MM-path starts with a method and ends when it reaches a method that does not issue any messages of its own; this is the point of message quiescence.

MM-Paths for Object-Oriented Software

Pseudo code for OO calendar

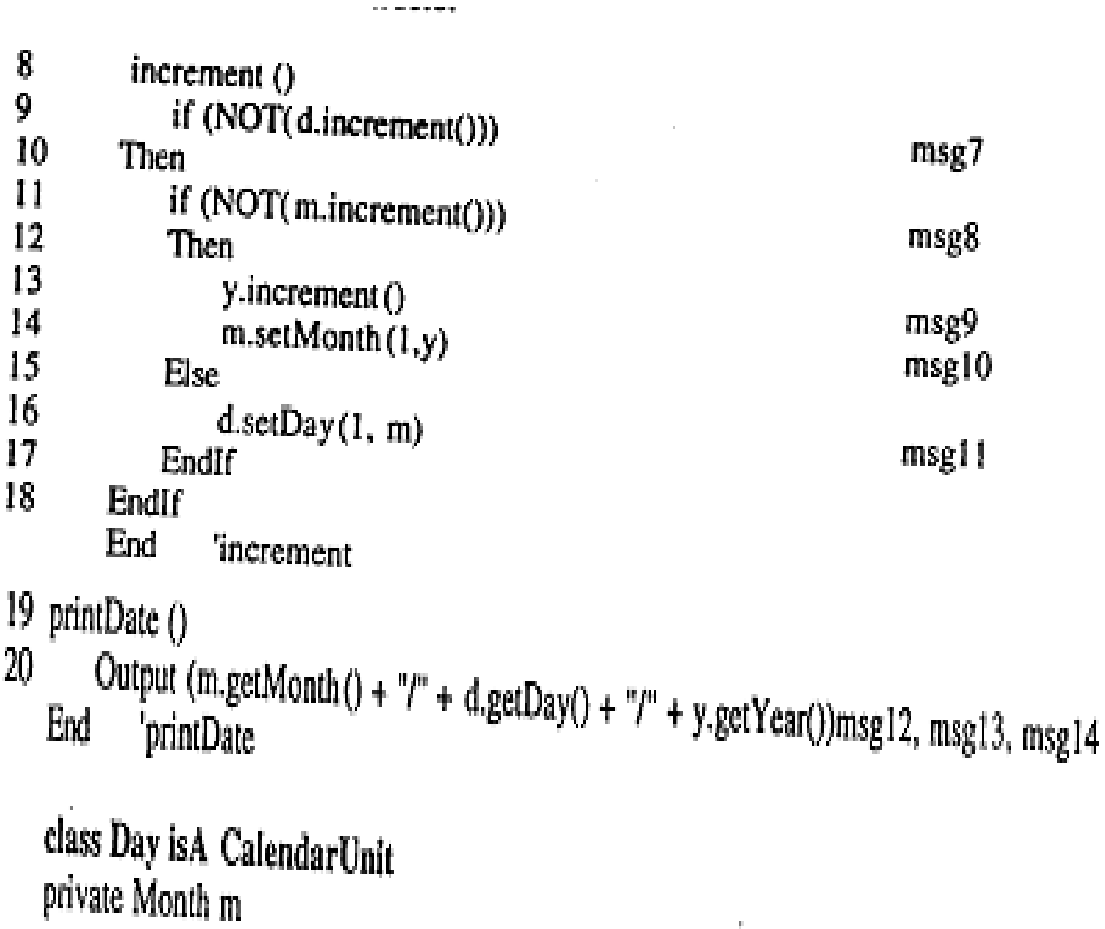


```
class testIt
    main()
1       testdate = instantiate Date(testMonth, testDay, testYear)   msg1
2       testdate.increment()                                         msg2
3       testdate.printDate()                                         msg3
    End     'testIt

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
```

MM-Paths for Object-Oriented Software

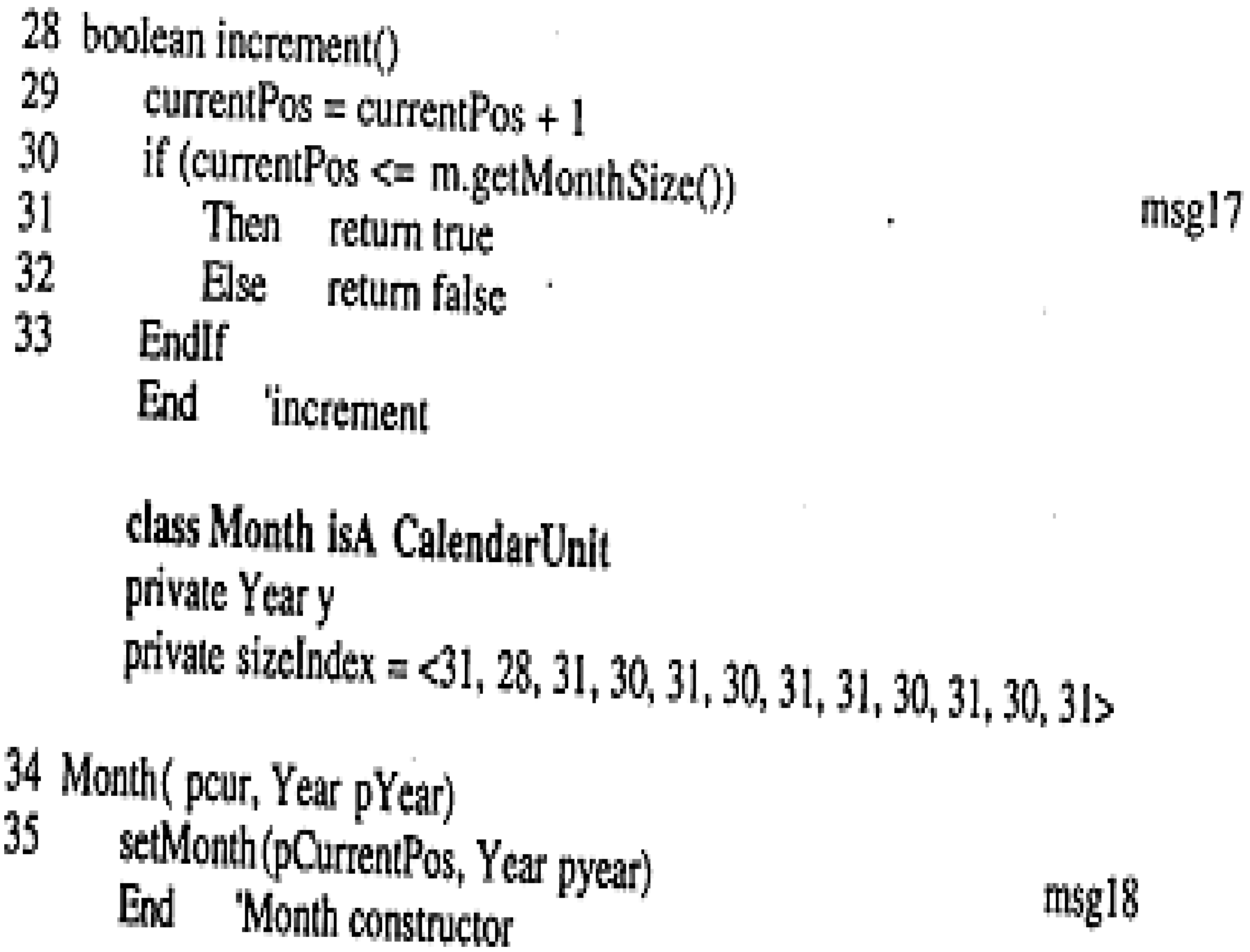


```
8       increment ()
9          if (NOT(d.increment()))
10      Then                                            msg7
11         if (NOT( m.increment()))
12         Then                                         msg8
13             y.increment ()
14             m.setMonth (1,y)                     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

MM-Paths for Object-Oriented Software

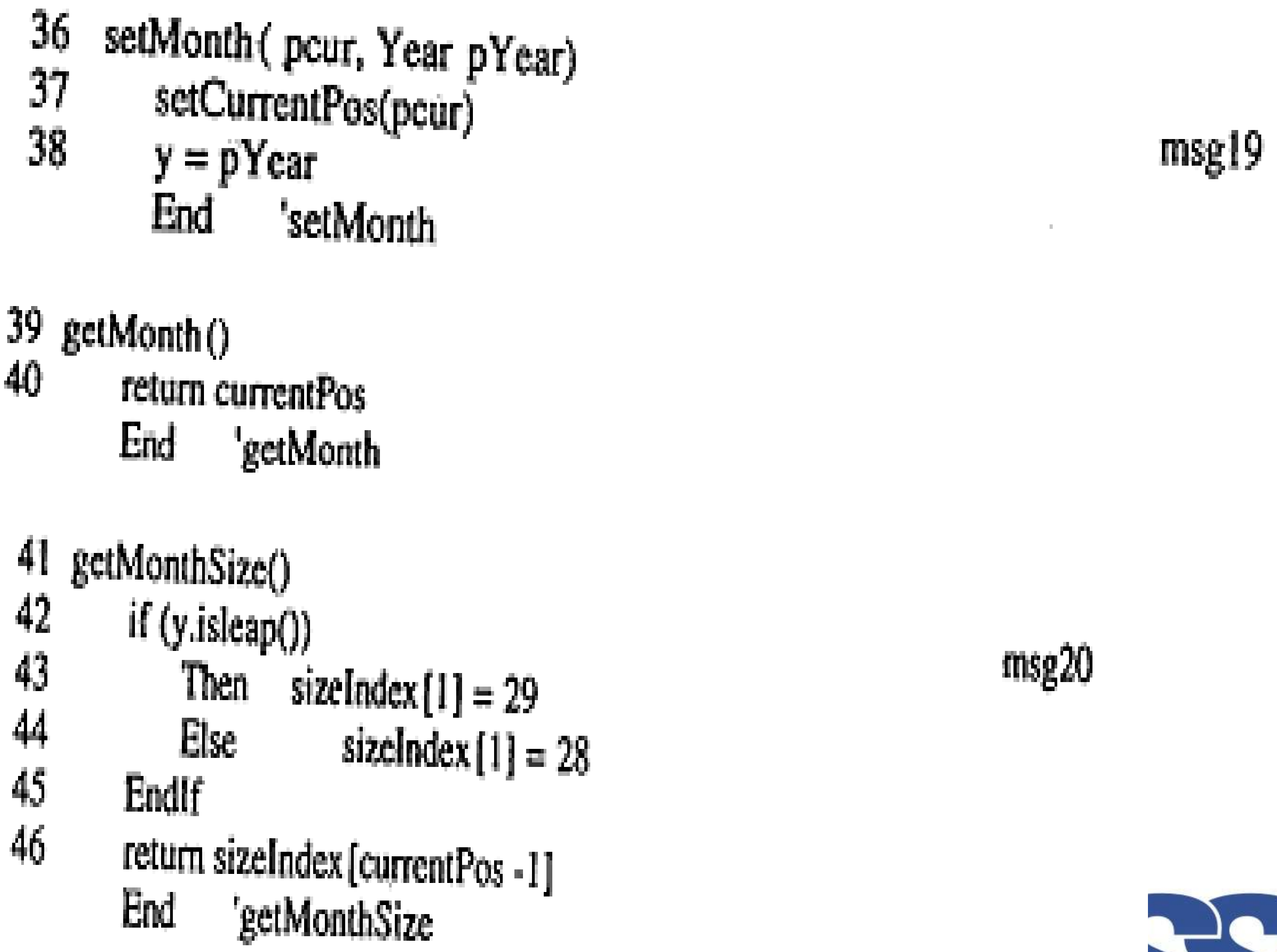


```
28 boolean increment()
29        currentPos = currentPos + 1
30        if (currentPos <= m.getMonthSize())                                msg17
31            Then    return true
32            Else    return false
33        EndIf
          End     'increment

          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)
          End     'Month constructor                                       msg18
```

MM-Paths for Object-Oriented Software

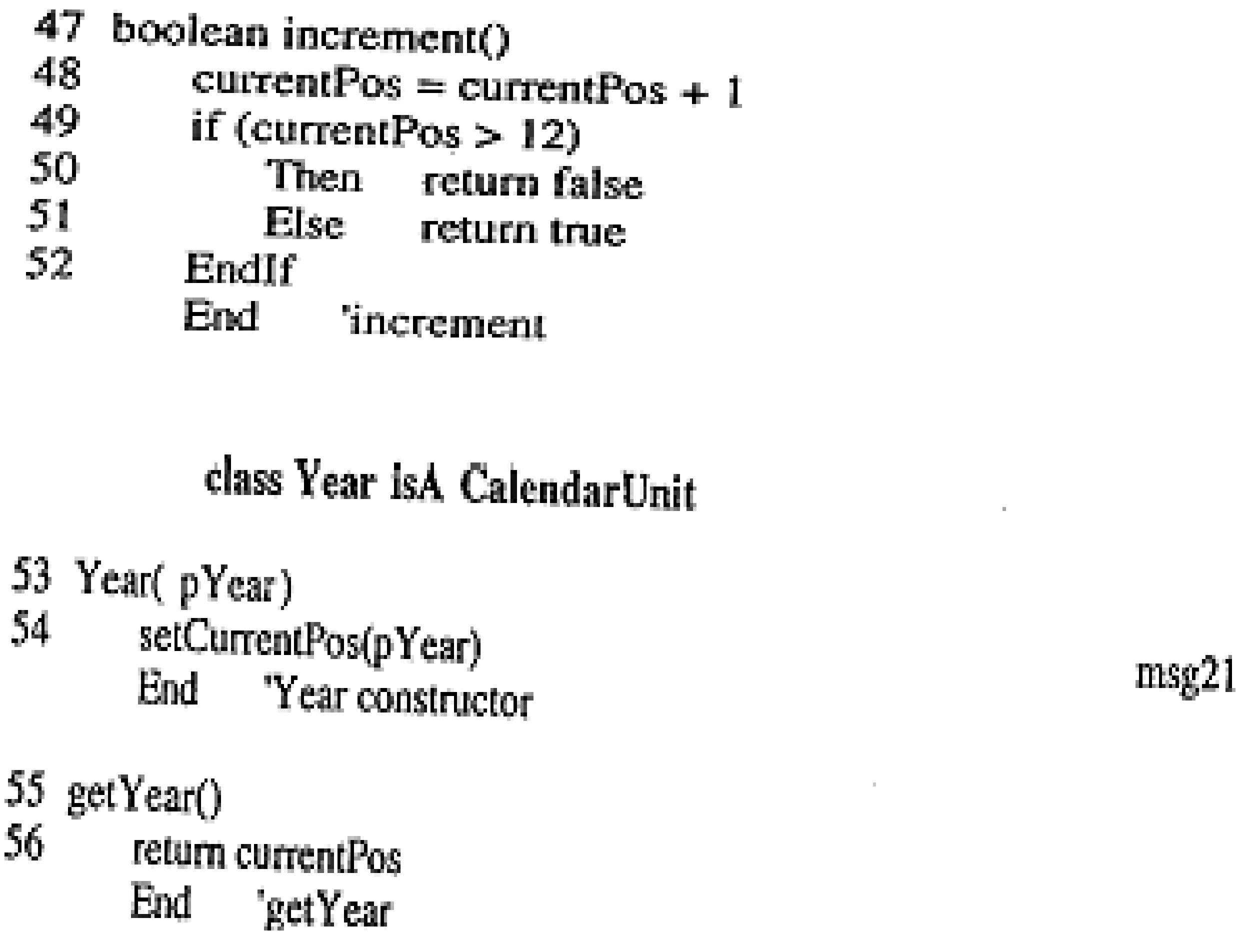


```
36  setMonth( pcur, Year pYear)
37      setCurrentPos(pcur)
38      y = pYear                                        msg19
        End     'setMonth

39  getMonth()
40      return currentPos
        End     'getMonth

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

MM-Paths for Object-Oriented Software



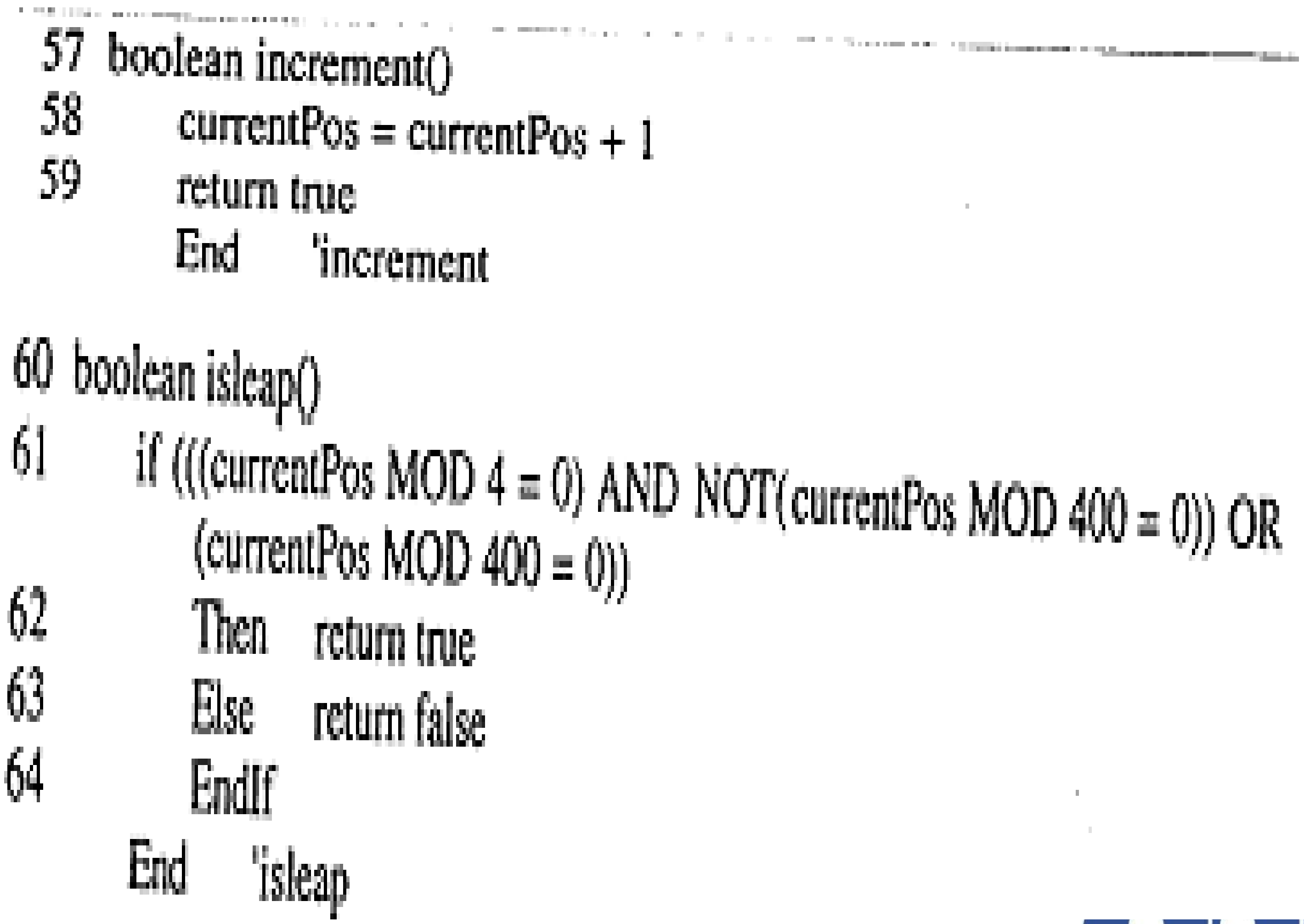
```
47  boolean increment()
48        currentPos = currentPos + 1
49        if (currentPos > 12)
50              Then    return false
51              Else    return true
52        EndIf
          End     'increment

          class Year isA CalendarUnit

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

55  getYear()
56        return currentPos
          End     'getYear
```

MM-Paths for Object-Oriented Software



```
boolean increment()
    currentPos = currentPos + 1
    return true
    End    'increment

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

MM-Paths for Object-Oriented Software

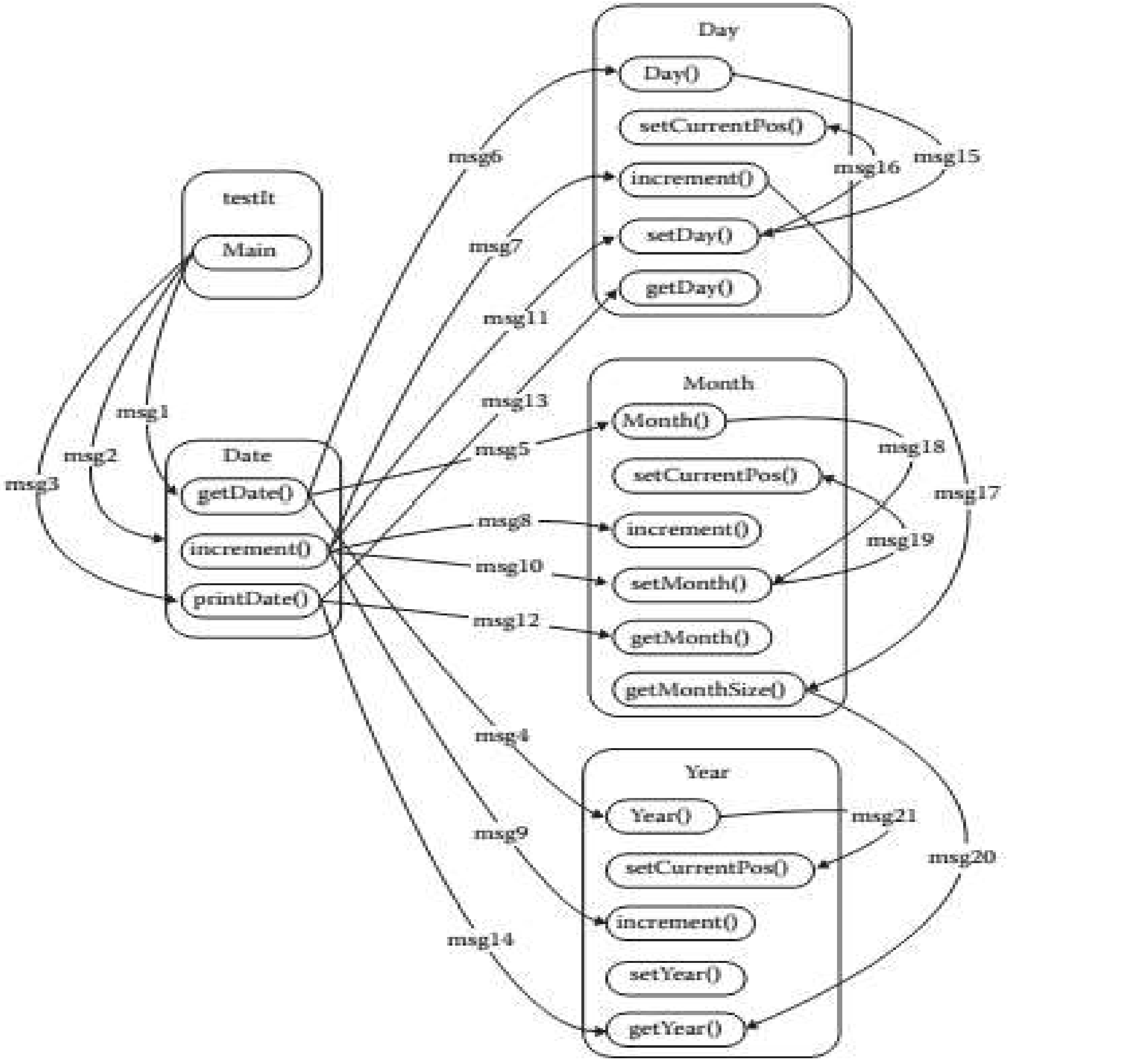
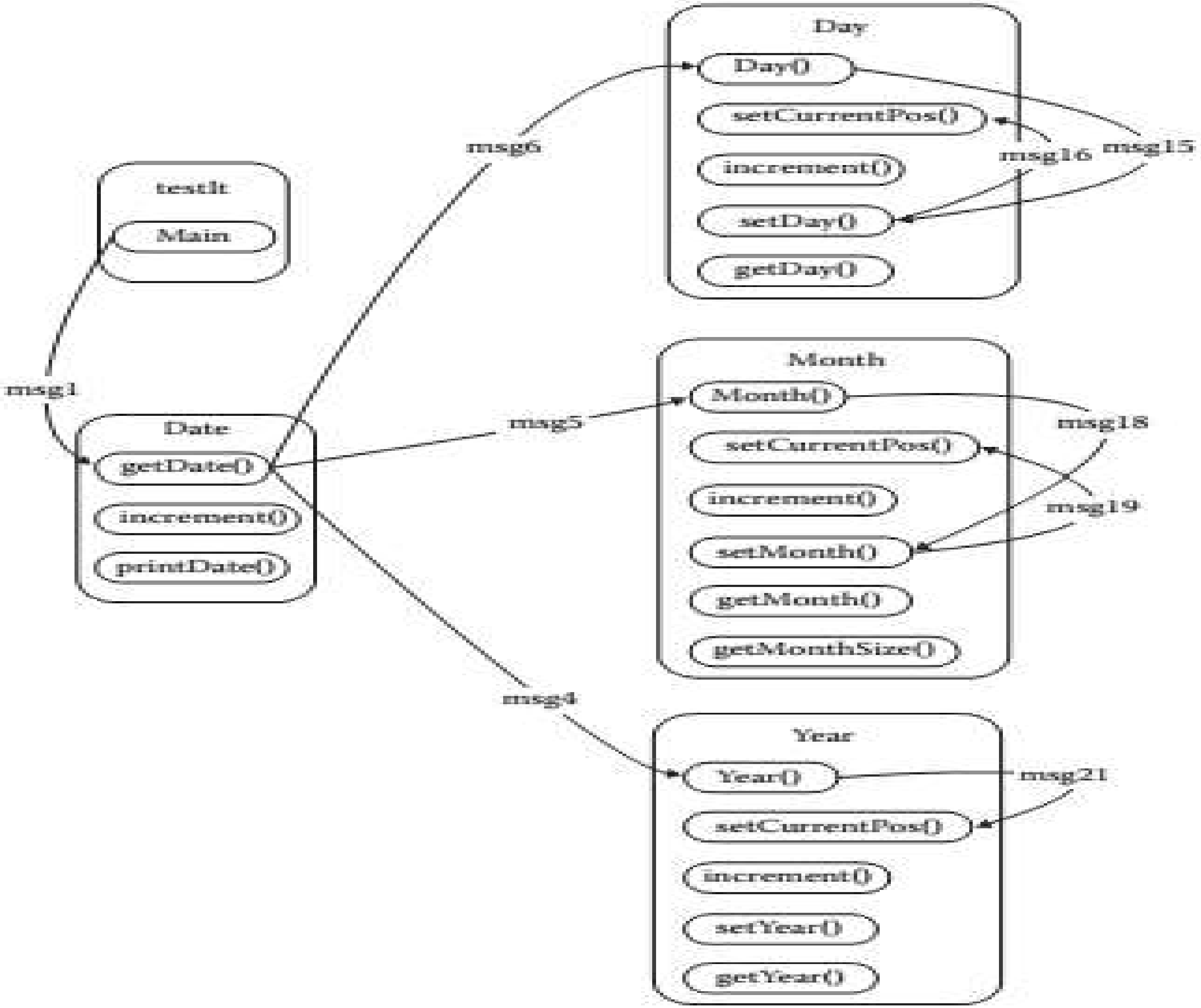
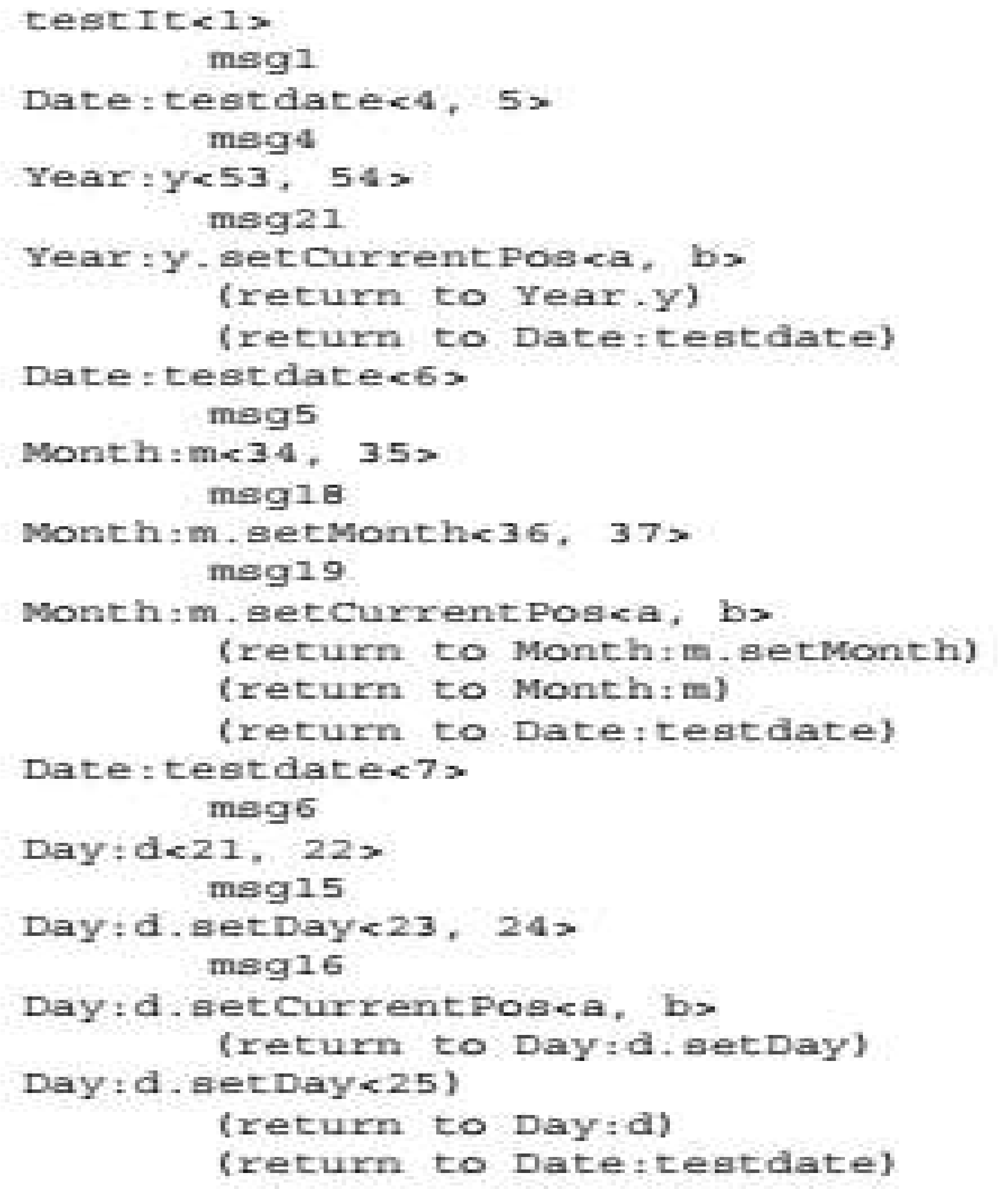


Figure 15.11 **Potential message flow in ooNextDate.**

MM-path for January 3, 2013.



Test CASE.



```
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
Day:d<21, 22>
        msg15
Day:d.setDay<23, 24>
        msg16
Day:d.setCurrentPos<a, b>
        (return to Day:d.setDay)
Day:d.setDay<25)
        (return to Day:d)
        (return to Date:testdate)
```

look for MM-paths to make sure that every message (edge) in the graph is traversed