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

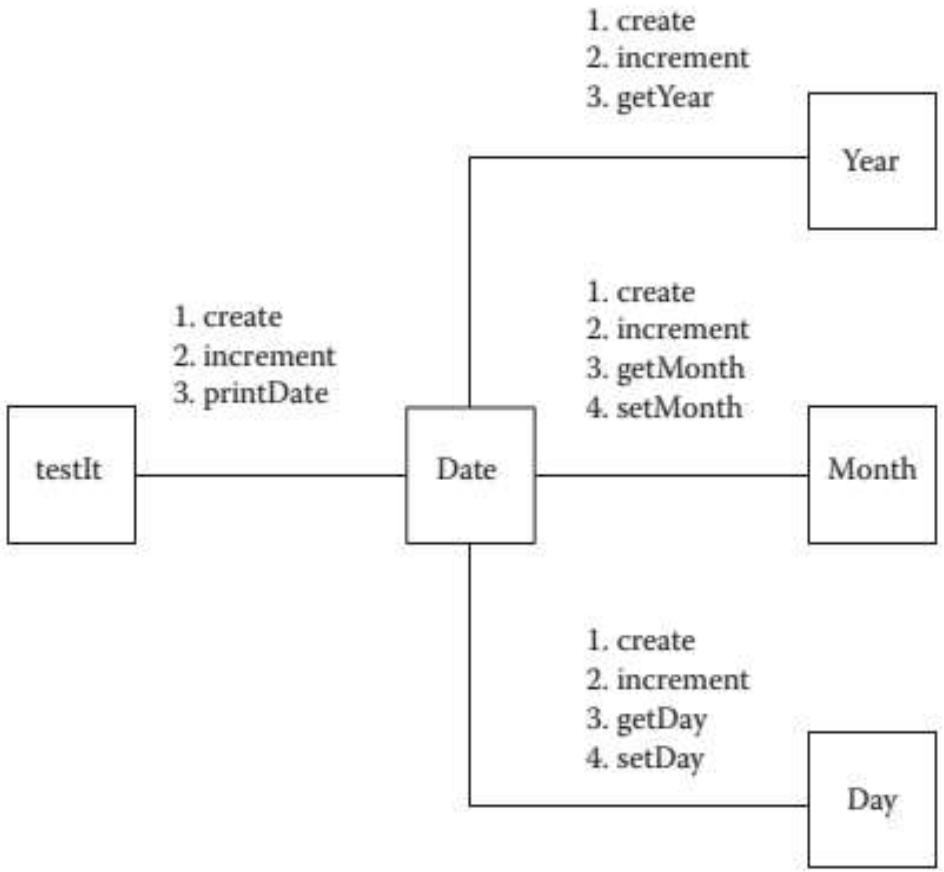


Figure 15.9 Collaboration diagram for ooCalendar.

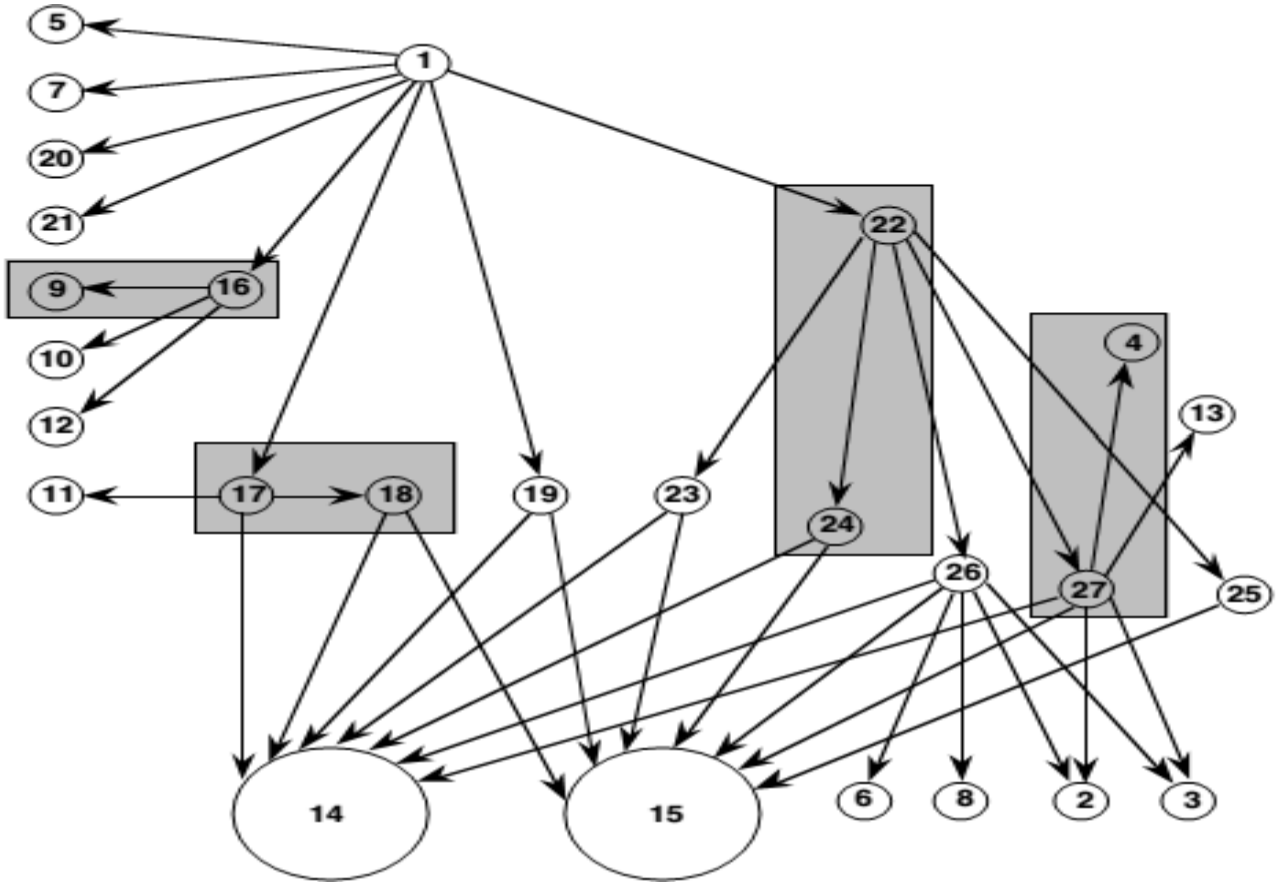
UML Support for Integration Testing

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

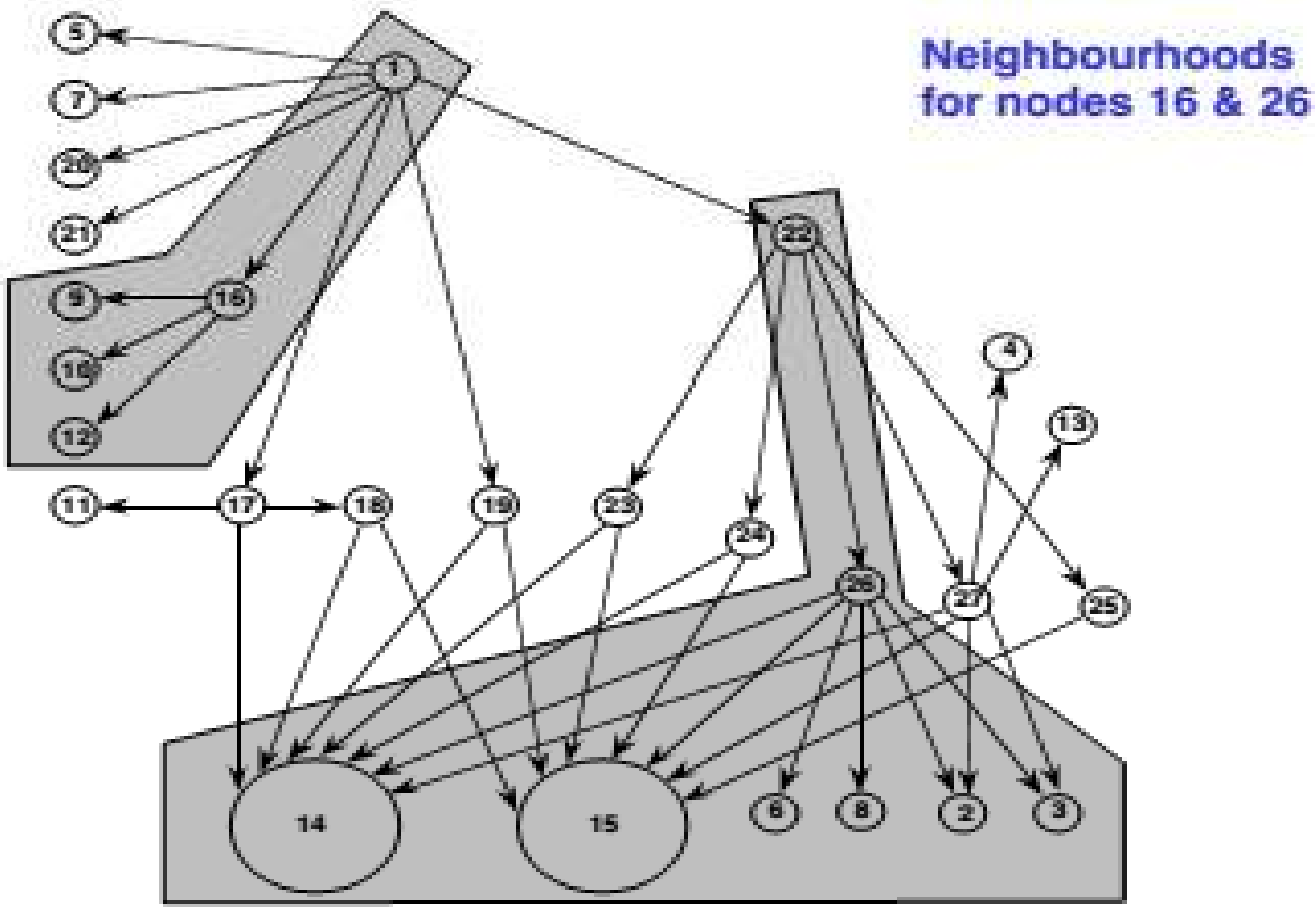
- start with the ultracenter and the neighborhood of nodes one edge away, then add the nodes two edges away, and so on.
- The set of nodes that are one edge away from the given node is called neighborhood nodes (first level of testing)
- The set of nodes that are two edges away from the given node is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



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.

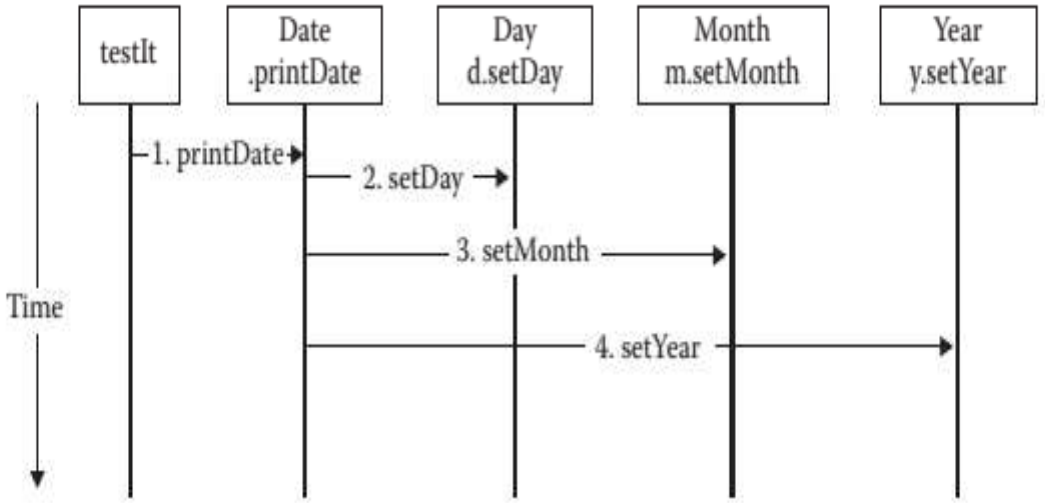


Figure 15.10 Sequence diagram for printDate.

UML Support for Integration Testing

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

```
1.    testDate
2.          d.setDay(27)
3.          m.setMonth(5)
4.          y.setYear(2013)
5.          Output ("Expected value is 5/27/2013")
6.          testIt.printDate
7.          Output ("Actual output is...")
8.    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?

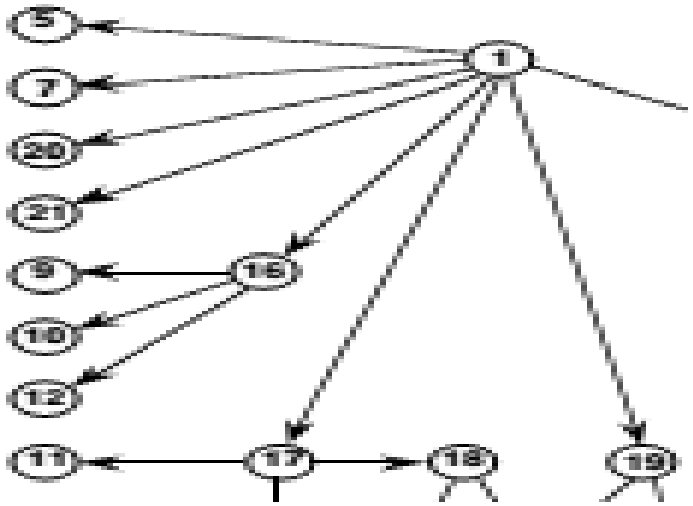
- Using collaboration diagrams or sequence diagrams as a basis for object-oriented integration testing is suboptimal.
- Collaboration diagrams force a pairwise approach to integration testing.
- Sequence diagrams are a little better, but somehow the integration tester needs all the sequence diagrams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