Chapter 15

Object-Oriented Testing
Issues in Testing Object-Oriented Software

• How is object-oriented (o-o) software different/special?
  – Need to clarify unit
  – Composition vs. decomposition
  – No reason for integration testing based on functional decomposition tree
  – Can still use the (better) idea of Call Graph based integration

• Properties of an o-o programming language
  – Inheritance
  – Encapsulation
  – Polymorphism

• Message communication
  – Message quiescence
  – Event quiescence
# Levels of Object-Oriented Testing

<table>
<thead>
<tr>
<th>Level</th>
<th>Item</th>
<th>Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>Method of an object?</td>
<td>Program graph</td>
</tr>
<tr>
<td></td>
<td>Class?</td>
<td></td>
</tr>
<tr>
<td>Integration</td>
<td>MM-Path</td>
<td>Message quiescence</td>
</tr>
<tr>
<td></td>
<td>Atomic System Function</td>
<td>Event quiescence</td>
</tr>
<tr>
<td>System</td>
<td>Thread</td>
<td>Source to sink ASF</td>
</tr>
<tr>
<td></td>
<td>Thread Interaction</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Notice the cascading levels of interaction:
- unit testing covers statement interaction,
- MM-Path testing covers method interaction,
- ASF testing covers MM-Path interaction,
- thread testing covers object interaction, and all of this culminates in thread interaction.
## Re-usable Testing Techniques

<table>
<thead>
<tr>
<th>Level</th>
<th>Item</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>Method of an object</td>
<td>Traditional specification and/or code based</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>StateChart-based</td>
</tr>
<tr>
<td>Integration</td>
<td>MM-Path</td>
<td>New definition?</td>
</tr>
<tr>
<td></td>
<td>Atomic System Function</td>
<td>New definition?</td>
</tr>
<tr>
<td>System</td>
<td>Thread</td>
<td>New definition?</td>
</tr>
<tr>
<td></td>
<td>Thread Interaction</td>
<td>(StateCharts)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(as before)</td>
</tr>
</tbody>
</table>
Units in Object-Oriented Software

• Guidelines for units
  – A unit is the smallest software component that can be compiled and executed. (traditional view, usually in procedural code)
  – A unit is a software component that would never be assigned to more than one designer to develop. (project management view, appropriate for both procedural and o-o software)
  – Unit developer does detailed design, coding, and unit level testing

• Candidate o-o units
  – A single method
  – A class (generally accepted view)
Single Methods as Units

- Appropriate for large classes
- Consistent with the one designer/one unit practice
- Single method testing
  - reduces to unit testing for procedural units
  - o-o units are typically less complex (cyclomatic) than procedural units
- This choice mandates “intra-class” integration testing.
Classes as Units

• Generally accepted view
• In UML, the *de facto* behavioral model for a class is a StateChart.
• BUT, StateChart composition is possible only in extremely simple cases.
• Testing complexity shifts from unit level to integration level
Composition and Encapsulation

• Composition
  – normal approach for o-o design
  – mandates strong unit level testing
  – traditional precepts of coupling and cohesion are good practice for o-o units

• Encapsulation
  – based on coupling and cohesion
  – shifts complexity and testing to integration level
Implications of Inheritance

Account

accountNumber
Balance

getBalance()
setBalance()

checkingAccount

checkProcessingCharge
checkNumber

postCharge()

savingsAccount

interestRate

postInterest()
Implications of Inheritance—Flattened Classes

<table>
<thead>
<tr>
<th>checkingAccount</th>
<th>savingsAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountNumber</td>
<td>accountNumber</td>
</tr>
<tr>
<td>Balance</td>
<td>Balance</td>
</tr>
<tr>
<td>checkProcessingCharge</td>
<td>interestRate</td>
</tr>
<tr>
<td>checkNumber</td>
<td></td>
</tr>
<tr>
<td>getBalance()</td>
<td>getBalance()</td>
</tr>
<tr>
<td>setBalance()</td>
<td>setBalance()</td>
</tr>
<tr>
<td>postCharge()</td>
<td>postInterest()</td>
</tr>
</tbody>
</table>
Example—O-O Windshield Wiper

Class lever(leverPosition;
   private senseLeverUp(),
   private senseLeverDown() )

Class dial(dialPosition;
   private senseDialUp(),
   private senseDialDown() )

Class wiper(wiperSpeed;
   setWiperSpeed(newSpeed) )

Which unit controls the interaction between the Lever and the Dial?
Saturn Windshield Wiper Objects

```
[Diagram showing:
  Wiper
  Speed
  Lever Position
  Dial Position
  Compute Speed

  Lever
  position
  Sense Up
  Sense Down

  Dial
  position
  Sense Left
  Sense Right]
```
Windshield Wiper Class Behavior

Lever
- Off
  - leverUp
  - leverDown
- Int
  - leverUp
  - leverDown
- Low
  - leverUp
  - leverDown
- High

Dial
- 1
  - dialUp
  - dialDown
- 2
  - dialUp
  - dialDown
- 3
  - dialUp
  - dialDown

Wiper
- 0 wipes/minute
- 6 wipes/minute
- 12 wipes/minute
- 20 wipes/minute
- 30 wipes/minute
- 60 wipes/minute
Unit Testing for Windshield Wiper Classes

- Can test only the Lever and Dial classes directly.
- Test cases to make sure event-oriented methods make correct changes to the position attributes.
- Need “mock objects” (o-o analog of drivers for procedural code) to provide inputs to test the wiper class.
- Note how the “line” between unit and integration testing is blurred!
## Sample Unit Test Case for Lever

<table>
<thead>
<tr>
<th>Test Case Name</th>
<th>Test Sense Up method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
<td>Lever-1</td>
</tr>
<tr>
<td>Description</td>
<td>Verify that the LeverUp method correctly changes the value of the position attribute</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>1. position = Off</td>
</tr>
<tr>
<td>Input(s)</td>
<td>Expected Output(s)</td>
</tr>
<tr>
<td>1. move lever to Int</td>
<td>2. position = Int</td>
</tr>
<tr>
<td>3. move lever to Low</td>
<td>4. position = Int</td>
</tr>
<tr>
<td>5. move lever to High</td>
<td>6. position = Int</td>
</tr>
<tr>
<td>Post condition(s)</td>
<td>1. position = High</td>
</tr>
<tr>
<td>Test Result?</td>
<td>Pass/Fail</td>
</tr>
<tr>
<td>Test operator</td>
<td>Paul Jorgensen</td>
</tr>
<tr>
<td>Test Date</td>
<td>Dec. 8, 2013</td>
</tr>
</tbody>
</table>
StateChart-Based Testing for the Windshield Wiper

- Need an engine to execute the StateChart
  - execute “interesting” scenarios
  - save paths as skeletons of system test cases
- The InState events show how the Wiper class can be tested once the Lever and Dial classes have been tested.
### Sample System Level Test Case

<table>
<thead>
<tr>
<th>Test Case Name</th>
<th>Exercise all wiper speeds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
<td>WindshieldWiper-1</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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</td>
</tr>
<tr>
<td><strong>Pre-Conditions</strong></td>
<td>1. The Lever is in the OFF position</td>
</tr>
<tr>
<td></td>
<td>2. The Dial is at the 1 position</td>
</tr>
<tr>
<td></td>
<td>3. The wiper speed is 0</td>
</tr>
<tr>
<td>Input events</td>
<td>Output events</td>
</tr>
<tr>
<td>1. move lever to INT</td>
<td>2. speed is 6</td>
</tr>
<tr>
<td>3. move dial to 2</td>
<td>4. speed is 12</td>
</tr>
<tr>
<td>5. move dial to 3</td>
<td>6. speed is 20</td>
</tr>
<tr>
<td>7. move lever to LOW</td>
<td>8. speed is 30</td>
</tr>
<tr>
<td>9. move lever to HIGH</td>
<td>10. speed is 60</td>
</tr>
<tr>
<td><strong>Post conditions</strong></td>
<td>1. The Lever is in the HIGH position</td>
</tr>
<tr>
<td></td>
<td>2. The Dial is at the 3 position</td>
</tr>
<tr>
<td></td>
<td>3. The wiper speed is 60</td>
</tr>
</tbody>
</table>
Integration Testing for O-O Software

• In general, o-o methods have low cyclomatic complexity, BUT

• The complexity shifts to the integration level.

• Possibilities
  – Call Graph based integration (very similar to procedural code strategies—pairwise or neighborhood)
  – MM-Paths for o-o code

• Not much help from UML

• UML collaboration diagram predisposes pairwise integration among classes
Testing Object-Oriented NextDate

- Example—our familiar NextDate
- Rewritten here in object-oriented style
- One Abstract Class—CalendarUnit
- Classes
  - testIt
  - Date
  - Day
  - Month
  - Year
- (See text for pseudo-code)
- Program graphs in next few slides
Program Graphs of Date Methods

testIt
1
2
3

Date.constructor
4
5
6
7

Date.increment
8
9
10
11
12
13
14
15
16
17
18

Date.printDate
19
20
Program Graphs of Day Methods

Day.constructor

Day.setDay

Day.increment

Day.setCurrentPos

Day.getDay

21 → 22

23 → 24 → 25

28 → 29 → 30 → 31 → 32

a → b

26 → 27
Program Graphs of Month Methods

Month.constructor
34
↓
35

Month.setMonth
36
↓
37
↓
38

Month.setCurrentPos
a
↓
b

Month.getMonthsize
41
↓
42
↓
43
↓
44
↓
45
↓
46

Month.getMonth
39
↓
40

boolean.increment
47
↓
48
↓
49
↓
50
↓
51
↓
52
Program Graphs of Year Methods

Year.setCurrentPos

a

b

Year.constructor

53

54

Year.getYear

55

56

boolean.increment

57

58

59

boolean.isLeap

60

61

62

63

64
ooNextDate Collaboration Diagram

- **testIt**:
  - 1. create
  - 2. increment
  - 3. printDate

- **Date**:
  - 1. create
  - 2. increment
  - 3. getYear
  - 4. setMonth

- **Year**:
  - 1. create

- **Month**:
  - 1. create
  - 2. increment
  - 4. setMonth

- **Day**:
  - 1. create
  - 2. increment
  - 3. getDay

- **ooNextDate**:
  - 1. create
  - 2. increment
  - 3. getYear
printDate Sequence Diagram

- **testIt**
  - 1. printDate

- **Date**
  - 2. setDay

- **Day**
  - 3. setMonth

- **Month**
  - 4. setYear

- **Year**
MM-Paths for O-O Software

• Definition: An object-oriented MM-Path is a sequence of method executions linked by messages.

• Call Graph slightly revised
  – nodes are methods
  – edges are messages

• Next three slides show sample Call Graphs for ooNextDate
All MM-Paths as a Call Graph

Chapter 15 O-O Testing
MM-Path for January 3, 2013
MM-Path for April 30, 2013
Integration Test Coverage Metrics

• Coverage metrics with respect to Call Graph
• Given a set of test cases that
  – covers each method (node coverage)
  – covers each message (edge coverage)
  – covers each path
System Testing for O-O Software

• Nearly equivalent to system testing for procedural software.
• Some possibilities for sources of system test cases...
  – Use cases
  – Model-Based system testing
Second Example (event-driven system testing)

Currency Converter

U.S. Dollar amount
Equivalent in ...

- Brazil
- Canada
- European community
- Japan

Compute
Clear
Quit
Second Example (continued)

![Currency Converter Diagram]

- **label1**: Currency Converter
- **label2**: U.S. Dollar amount
- **label3**: Equivalent in ...
- **optBrazil**: Brazil
- **optCanada**: Canada
- **optEU**: European community
- **optJapan**: Japan
- **frmCurrConv**: Form
- **txtDollar**: Text Box
- **lblEquivAmount**: Label
- **cmdCompute**: Command Button
- **cmdClear**: Command Button
- **cmdQuit**: Command Button
## Input Events for the Currency Converter

<table>
<thead>
<tr>
<th>Input Events</th>
<th>Input Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip1 Enter US Dollar amount</td>
<td>ip2.4 Click on Japan</td>
</tr>
<tr>
<td>ip2 Click on a country button</td>
<td>ip3 Click on Compute button</td>
</tr>
<tr>
<td>ip2.1 Click on Brazil</td>
<td>ip4 Click on Clear button</td>
</tr>
<tr>
<td>ip2.2 Click on Canada</td>
<td>ip5 Click on Quit button</td>
</tr>
<tr>
<td>ip2.3 Click on European Community</td>
<td>ip6 Click on OK in error message</td>
</tr>
</tbody>
</table>
### Output Events for the Currency Converter

<table>
<thead>
<tr>
<th>Output Events</th>
<th>Output Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1 Display US Dollar Amount</td>
<td>op4 Reset selected country</td>
</tr>
<tr>
<td>op2 Display currency name</td>
<td>op4.1 Reset Brazil</td>
</tr>
<tr>
<td>op2.1 Display Brazilian Reals</td>
<td>op4.2 Reset Canada</td>
</tr>
<tr>
<td>op2.2 Display Canadian Dollars</td>
<td>op4.3 Reset European Community</td>
</tr>
<tr>
<td>op2.3 Display European Community Euros</td>
<td>op4.4 Reset Japan</td>
</tr>
<tr>
<td>op2.4 Display Japanese Yen</td>
<td>op5 Display foreign currency value</td>
</tr>
<tr>
<td>op2.5 Display ellipsis</td>
<td>op6 Error Msg: Must select a country</td>
</tr>
<tr>
<td>op3 Indicate selected country</td>
<td>op7 Error Msg: Must enter US Dollar amount</td>
</tr>
<tr>
<td>op3.1 Indicate Brazil</td>
<td>op8 Error Msg: Must select a country and enter US Dollar amount</td>
</tr>
<tr>
<td>op3.2 Indicate Canada</td>
<td>op9 Reset US Dollar Amount</td>
</tr>
<tr>
<td>op3.3 Indicate European Community</td>
<td>op10 Reset equivalent currency amount</td>
</tr>
<tr>
<td>op3.4 Indicate Japan</td>
<td></td>
</tr>
</tbody>
</table>
StateChart for the Currency Converter

- **In Storage**
  - ip 5: Quit
  - End Application

- **Idle**
  - Ip 4
  - op 2, op 4, op 9, op 10

- **Executing**
  - Ip 4

- **Country Selected**
  - Ip 2 / op 2, op 3
  - Ip 3 / op 7
  - Missing U.S. Dollar Message

- **U.S. Dollar Amount Selected**
  - Ip 1 / op 1
  - Ip 2 / op 2, op 3
  - Both Inputs Done

- **Missing Country and Dollar Message**
  - Ip 6
  - Ip 3 / op 7

- **Missing U.S. Dollar Message**
  - Ip 6

- **Missing Country Message**
  - Ip 6

- **Equiv. Amount Displayed**
  - Ip 1 / op 1
  - Ip 3 / op 5
  - Ip 2 / op 2, op 3
# Sample System Level Test Case

<table>
<thead>
<tr>
<th>System test Case -3</th>
<th>Normal usage (dollar amount entered first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test performed by</td>
<td>Paul Jorgensen</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>txtDollar has focus</td>
</tr>
<tr>
<td>Input events</td>
<td>Output events</td>
</tr>
<tr>
<td>1. Enter 10 on the keyboard</td>
<td>2. Observe 10 appears in txtDollar</td>
</tr>
<tr>
<td>3. Click on optEU button</td>
<td>4. Observe “euros” appears in label3</td>
</tr>
<tr>
<td>5. Click cmdCompute button</td>
<td>6. Observe “7.60” appears in lblEquivAmount</td>
</tr>
<tr>
<td>Post-Conditions</td>
<td>cmdClear has focus</td>
</tr>
<tr>
<td>Test result</td>
<td>Pass (on first attempt)</td>
</tr>
<tr>
<td>Date run</td>
<td>May 27, 2013</td>
</tr>
</tbody>
</table>
Constructs for Event- and Message-Driven Petri Nets

- Port Input Event
- Port Output Event
- Date (place)
- Message send/return
- Method execution path
Framework for Object-Oriented Dataflow Testing

- Need a construct to express the role of messages
- An extension of Event-Driven Petri Nets
- Event/Message–Driven Petri Nets
Elements of Event/Message–Driven Petri Nets

- Port Input Event
- Port Output Event
- Date (place)
- Message send/return
- Method execution path
Message from object A to object B
Data Flow from Message Passing
Data Flow from Inheritance

Diagram:

```
(1) ----> (2) ----> (3) ----> (4)
isA    isA    isA
```

Chapter 15 O-O Testing
Observations and Conclusions

• Classes/objects are more complicated than procedures
  – need to deal with inheritance, polymorphism
  – good encapsulation in o-o design helps
• Complexity shifts from methods to integration
• Dataflow testing seems appropriate for o-o integration testing.
• Not much help for Model-Based Testing at the integration level.