Section 6.2
Unit Testing

1. Overview
2. Techniques for unit testing

6.2.1 Overview

- Focus of unit testing
  - the objects and subsystems in individual components
  - groups of objects can be tested **after** individual objects are tested
- Characteristics of unit testing
  - reduces the complexity of the testing process
  - facilitates the finding and correcting of faults
  - allows for parallelism in the testing process

Overview (cont.)

- Candidate units
  - selected from the object model and subsystem decomposition
  - all objects should be tested
  - at minimum, participating objects in use cases should be tested
  - subsystems can be tested after all its classes have been tested

6.2.2 Techniques for Unit Testing

- Equivalence testing
- Boundary testing
- Path testing
- State-based testing
- Polymorphism testing

Equivalence Testing

- What is equivalence testing?
  - it's a blackbox technique
  - it minimizes the number of test cases
  - the input is partitioned into *equivalence classes*
    - testing will behave similarly for all members of an equivalence class
  - only one member of each equivalence class is tested

Equivalence Testing (cont.)

- Equivalence testing strategy
  - identify the equivalence classes for the input to test component
  - criteria for identifying equivalence classes
    - coverage:
      - every input belongs to an equivalence class
    - disjointedness:
      - no input belongs to more than one equivalence class
    - representation:
      - any error occurring with one member occurs for all members
  - select test input
    - for each equivalence class, select one valid and one invalid input

Equivalence Testing (cont.)

```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth(int month, int year) {…}
...
}
```

Figure 11-10 Interface for a method computing the number of days in a given month (in Java). The `getNumDaysInMonth()` method takes two parameters, a month and a year, both specified as integers.

Table 11-2 Equivalence classes and selected valid inputs for testing the `getNumDaysInMonth()` method.

Equivalence class	Value for month input	Value for year input
Months with 31 days, non–leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non–leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non–leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

Boundary Testing

- What is boundary testing?
  - it's a special case of equivalence testing
  - focus on conditions at the boundary of equivalence classes

- Disadvantage
  - some kinds of errors will not be detected

Table 11-3 Additional boundary cases selected for the `getNumDaysInMonth()` method.

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non–leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

Path Testing

- What is path testing?
  - it's a whitebox technique
  - it identifies faults by exercising all possible control flow paths through the code
  - strategy
    - construct flow graph for the test component
    - design test cases so that every edge is traversed at least once
  - does not detect faults associated with:
    - code omissions
    - invariants of data structures

Path Testing (cont.)

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

Figure 11-11 An example of a (faulty) implementation of the `getNumDaysInMonth()` method (Java).

Path Testing (cont.)

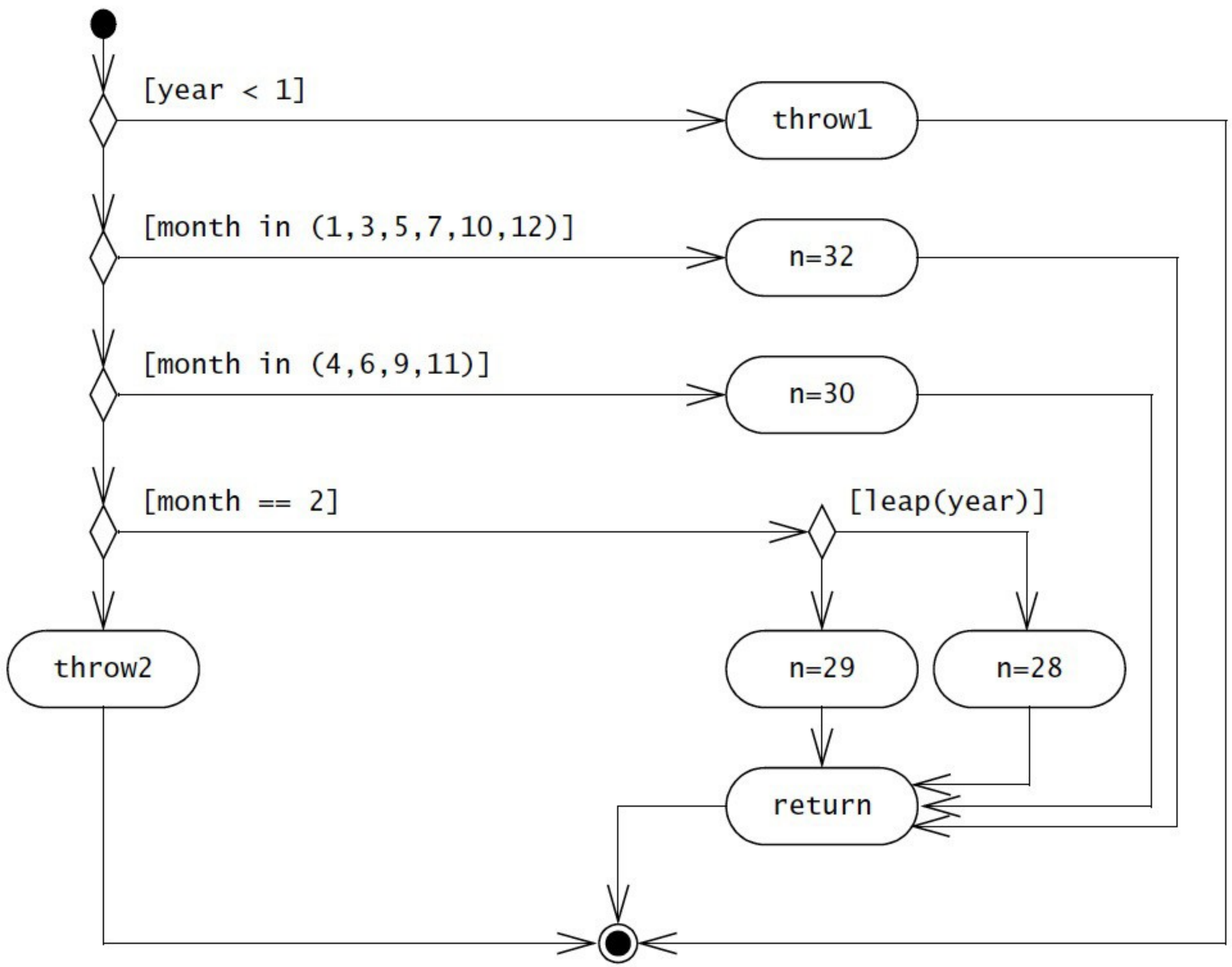


Figure 11-12 Equivalent flow graph for the (faulty) implementation of the `getNumDaysInMonth()` method of Figure 11-11 (UML activity diagram).

Path Testing (cont.)

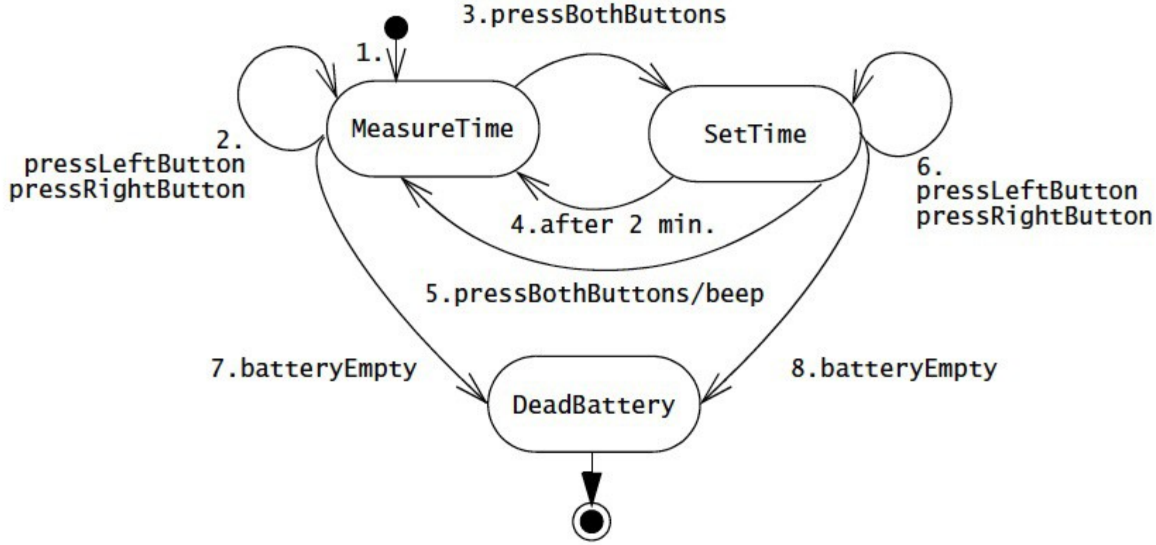
Table 11-4 Test cases and their corresponding path for the activity diagram depicted in Figure 11-12.

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2)	{n=28 return}
(year = 1904, month = 2)	{n=29 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

State-Based Testing

- What is state-based testing?
  - it compares *resulting* state of the system against *expected* state
  - it is class-based
  - strategy
    - for each state in the state machine diagram, derive a representative set of stimuli for each transition
  - it is similar to equivalence testing
  - issue: achieving a given state can be complex

State-Based Testing (cont.)



Stimuli	Transition tested	Predicted resulting state
Empty set	1. *Initial transition*	MeasureTime
Press left button	2.	MeasureTime
Press both buttons simultaneously	3.	SetTime
Wait 2 minutes	4. *Timeout*	MeasureTime
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	SetTime
Press both buttons simultaneously	5.	SetTime->MeasureTime
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	SetTime
Press left button	6. Loop back onto MeasureTime	MeasureTime

Figure 11-14 UML state machine diagram and resulting tests for 2Bwatch SetTime use case. Only the first eight stimuli are shown.

Polymorphism Testing

- What is polymorphism testing?
  - all possible dynamic bindings must be tested
  - this introduces a new challenge to testing
  - strategy
    - expand source code to:
      - typecast polymorphic object into each possible subclass
      - invoke operation on subclass
    - construct the flow graph
    - perform path testing

Polymorphism Testing (cont.)

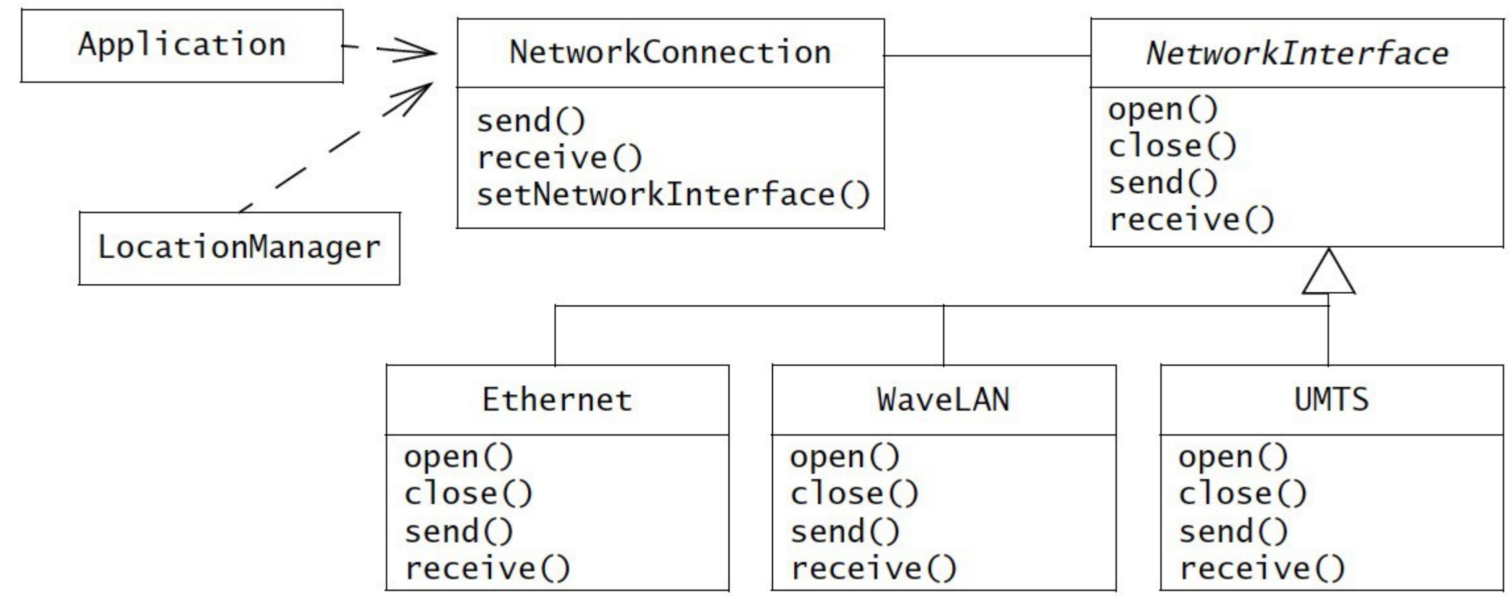


Figure 11-15 A Strategy design pattern for encapsulating multiple implementations of a NetworkInterface (UML class diagram).

Polymorphism Testing (cont.)

```java
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    if (nif.isReady()) {
        nif.send(queue);
        queue.setLength(0);
    }
}
}
```

```java
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    boolean ready = false;
    if (nif instanceof Ethernet) {
        Ethernet eNif = (Ethernet)nif;
        ready = eNif.isReady();
    } else if (nif instanceof WaveLAN) {
        WaveLAN wNif = (WaveLAN)nif;
        ready = wNif.isReady();
    } else if (nif instanceof UMTS) {
        UMTS uNif = (UMTS)nif;
        ready = uNif.isReady();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            Ethernet eNif = (Ethernet)nif;
            eNif.send(queue);
        } else if (nif instanceof WaveLAN){
            WaveLAN wNif = (WaveLAN)nif;
            wNif.send(queue);
        } else if (nif instanceof UMTS){
            UMTS uNif = (UMTS)nif;
            uNif.send(queue);
        }
        queue.setLength(0);
    }
}
}
```

Figure 11-16 Java source code for the NetworkConnection.send() message (left) and equivalent Java source code without polymorphism (right). The source code on the right is used for generating test cases.

Polymorphism Testing (cont.)

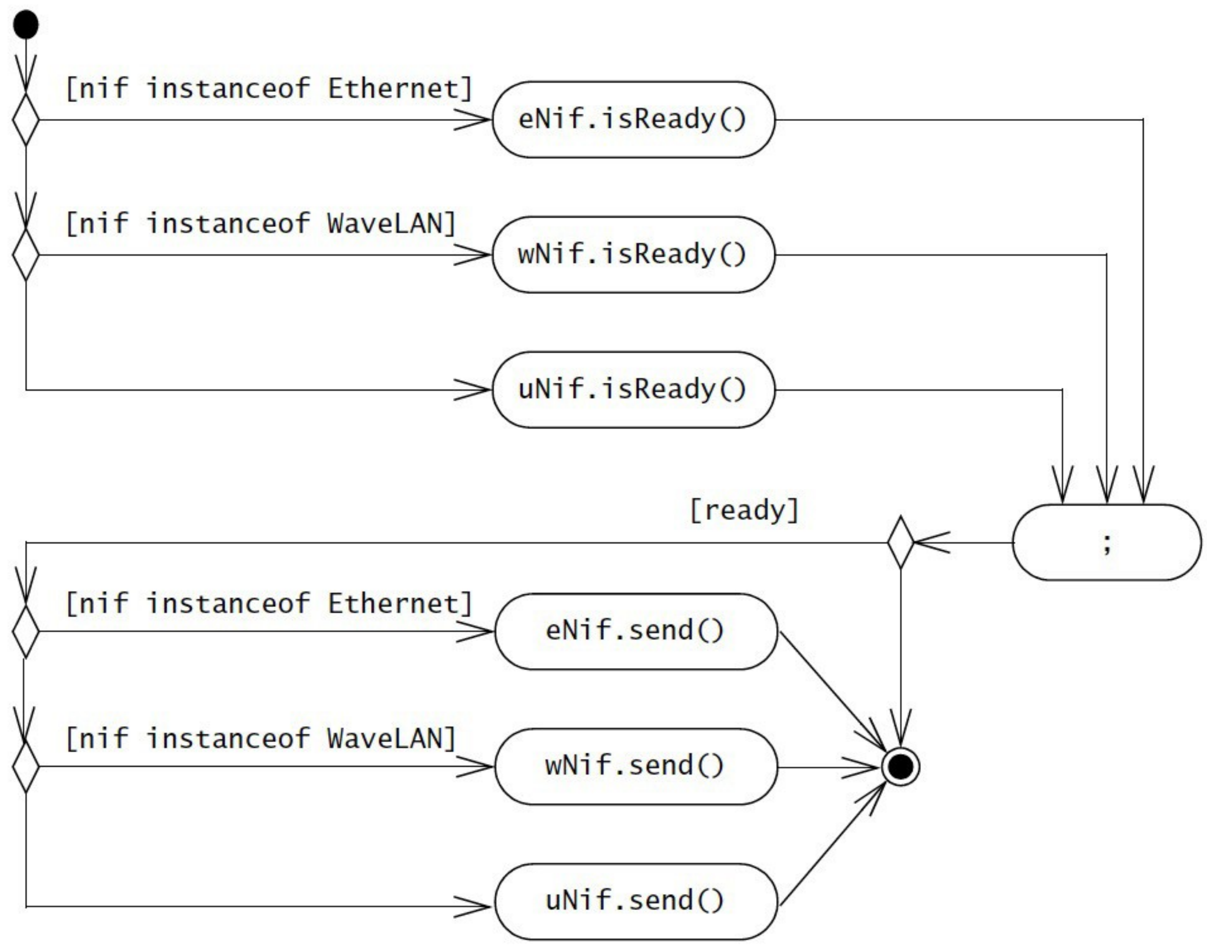


Figure 11-17 Equivalent flow graph for the expanded source code of the `NetworkConnection.send()` method of Figure 11-16 (UML activity diagram).