Building Tests and hw5

10-17-2012
Section 4

by Kellen Donohue, with material from Krysta Yousoufian
Agenda

• Assignments
  – hw2 will be returned soon
  – hw3 being returned
  – hw4 due tonight
  – hw5 released

• Building a test suite

• HW5 warm-up
Unit Test Best Practices

How to craft well-written JUnit tests
#1: Use descriptive asserts, test names

- When a test fails, JUnit tells you:
  - Name of test method
  - Message passed into failed assertion
  - Expected and actual values of failed assertion
  - Stack trace

- The more descriptive this information is, the easier it is to diagnose failures

- Avoid System.out.println()
  - Want any diagnostic info to be captured by JUnit and associated with that test method
#1: Use descriptive asserts, test names

- **Test name**: describe what’s being tested
  - Good: “testAddDaysWithinMonth,” ...
  - Not so good: “testAddDays1,” “testAddDays2,” ...
  - Useless: “test1,” “test2,” ...
  - Overkill:
    “testAddDaysOneDayAndThenFiveDaysThenNegativeFourDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary(“)
#1: Use descriptive asserts, test names

- **Test name:** describe what’s being tested
  - Good: “testAddDaysWithinMonth,” ...
  - Not so good: “testAddDays1,” “testAddDays2,” ...
  - Useless: “test1,” “test2,” ...
  - Overkill: “testAddDaysOneDayAndThenFiveDaysThenNegativeFourDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary(”
#1: Use descriptive asserts, test names

- **Assertions**: take advantage of expected & actual values
- Make sure you have the right order:

  ```java
  assertEquals(message, expected, actual)
  ```

- Use the right assert for the occasion:

  ```java
  assertEqual(expected, actual) instead of assertEquals(expected.equals(actual))
  ```

  ```java
  or assertEquals(expected==actual)
  ```

  ```java
  assertTrue(b) instead of assertEquals(true, b)
  ```
#1: Use descriptive asserts, test names

- **Assertion message**: contribute new information
  - No need to repeat expected/actual values or info in test name
  - e.g. details of what happened before the failure

Example:

```java
@Test
public void test_addDays_wrapToNextMonth() {
    Date actual = new Date(2050, 2, 15);
    actual.addDays(14);
    Date expected = new Date(2050, 3, 1);
    assertEquals("date after +14 days", expected, actual);
}
```
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

}
public class DateTest {

   ...

   // Test addDays when it causes a rollover between months
   @Test
   public void testAddDaysWrapToNextMonth() {
      Date actual = new Date(2050, 2, 15);
      actual.addDays(14);
      Date expected = new Date(2050, 3, 1);
      assertEquals("date after +14 days", expected, actual);
   }
}
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

    Use assertion to check expected results
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
public class DateTest {

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
Let’s put it all together!

```java
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);

        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

    That's it! Test is short & sweet
    }

```
#2: Keep tests small

• Ideally, each test only tests one “thing”
  – One “thing” usually means one method under one input condition

• Where possible, only test one method at a time
  – Not always possible – but if you test \( x() \) using \( y() \), try to test \( y() \) in isolation in another test
  – E.g. if you test \( \text{add()} \) using \( \text{contains()} \), separately test \( \text{contains()} \) before any items are added
#2: Keep tests small

- Only a few (likely one) assert statements per test
  - Test halts after first failed assertion
  - Don’t know whether later assertions would have failed

- Low-granularity tests help you isolate bugs
  - Tell you exactly what failed and what didn’t
What NOT to do

- IntArrayTest
- What’s wrong?
What NOT to do

• IntArrayTest
• What’s wrong?
What NOT to do

- IntArrayTest
- What’s wrong?

- testIntArray tests way too many things
  - Too many methods, array states
- Solution: break down by method being tested and/or state of array
- IntArrayTestBetter
#3: Choose the right tests

• Given a finite number of tests, want reasonable confidence in an infinite number of inputs

• Input = initial state of object + method arguments + ...
#3: Choose the right tests

- For each method, ask: what are the equivalence classes?
  - Items in a collection: none, one, many

- Write a test for each equivalence class
#3: Choose the right tests

- Consider common input categories
  - `Math.abs()`: negative, zero, positive values

- Consider boundary cases
  - Inputs on the boundary between equivalence classes
    - `Person.isMinor()`: age < 18, **age == 18**, age > 18

- Consider edge cases
  - -1, 0, 1, empty list, `arr.length, arr.length-1`

- Consider error cases
  - Empty list, null object
#3: Choose the right tests

- Consider common input categories
  - Math.abs(): negative, zero, positive values

- Consider boundary cases
  - Inputs on the boundary between equivalence classes
  - Person.isMinor(): age < 18, \texttt{age == 18}, age > 18

- Consider edge cases
  - -1, 0, 1, empty list, arr.length, arr.length-1

- Consider error cases
  - Empty list, null object
Other guidelines

• Test all methods
  – Caveat: constructors don’t necessarily need explicit testing

• Keep tests simple – avoid complicated logic
  – minimize if/else, loops, switch, etc.
  – Don’t want to debug your tests!

• Tests should always have at least one assert
  – Unless testing that an exception is thrown
  – Simply testing that an exception is not thrown is not necessary
  – assertTrue(true); doesn’t count!
Other guidelines

• Tests should be *isolated*
  – Not dependent on side effects of other tests
  – Should be able to run in any order

• Use helper methods to factor out common operations
  – E.g. setting up initial state of an object
Setup and Teardown

• Methods to run before/after each test case method is called:

```java
@Before
public void name() { ... }

@After
public void name() { ... }
```

• Methods to run once before/after the entire test class runs:

```java
@BeforeClass
public static void name() { ... }

@AfterClass
public static void name() { ... }
```
Example: Date

- public Date(int year, int month, int day)
- public Date()
  // today
- public int getDay(), getMonth(), getYear()
- public void addDays(int days)
  // advances by days
- public int daysInMonth()
- public String dayOfWeek()
  // e.g. "Sunday"
- public boolean equals(Object o)
- public boolean isLeapYear()
- public void nextDay()
  // advances by 1 day
- public String toString()

• Come up with unit tests to check the following:
  – That no Date object can ever get into an invalid state.
  – That the addDays method works properly.
  – It should be efficient enough to add 1,000,000 days in a call.
Example: **IntStack**

- What tests should we write?
More examples

• How would we test the following Collections interface methods:
  • `Collections.binarySearch`
  • `Collections.sort`
  • …
  • (Assume the List we pass in has already been tested)
JUnit Summary

• Tests need *failure atomicity* (ability to know exactly what failed).
  – Each test should have a descriptive name.
  – Assertions should have clear messages to know what failed.
  – Write many small tests, not one big test.
• Test for expected errors / exceptions.
• Choose a descriptive assert method, not always `assertTrue`.
• Choose representative test cases from equivalent input classes.
• Avoid complex logic in test methods if possible.
• Use helpers, `@Before` to reduce redundancy between tests.
Homework 5

• Design, spec, build, and test your own Graph ADT
• No starter source code
• Unique testing framework
Graph Explanation

1

2

3

4

A

B

C

D

E
HW 5 Explanation

• Specification
  – Design your classes, how they fit together, what operations look like
  – Don’t write a “kitchen sink” or “god” class
HW 5 Testing

• Specification vs. Implementation Tests
  – Implementation tests
  – JUnit tests
  – Black box & White box
  – Specification tests
  – We want to see if your program actually implements a Graph properly
  – Issue commands like AddNode, AddEdge, ListNode, ListEdge, checked externally
  – Black box by necessity
HW5TestDriver

• Specification Tests
  – Commands run on your program
  – For each test
  – Run the commands in the file ending in .test
  – Save output in .actual
  – Compared to .expected

• Demo in Eclipse
Design Brainstorming

- Work by yourself first, then compare with neighbors

- Two implementation strategies
  - As an incidence list, in which each vertex stores its edges and each edge stores its connected vertices.
  - As an adjacency matrix, which explicitly represents, for every pair \( \langle A, B \rangle \) of edges, whether there is a link from \( A \) to \( B \), and how many.
Design Review

• Share what you came up with, RI, and AF
• Runtime/Space complexity of various operations
  – Which is faster for
  – Seeing if two vertices are adjacent?
  – Adding a vertex?
  – Adding an edge?
  – Which takes more memory on sparse/dense graphs