UNIT TESTING

Krysta Yousoufian
CSE 331 Section
Jan. 19, 2012

With material from Marty Stepp, David Notkin, and The Pragmatic Programmer
Plan for today / Thursday

• HW3
• Background on unit testing
• JUnit mechanics
• JUnit best practices
Background

What unit testing is and why it matters
Kinds of Testing

- **Unit testing**: test each module of a program in isolation
  - “Module” usually means one class

- Integration testing
  - Do the modules “play well” together?

- Validation testing
  - Does the system do what the client wants and needs?
  - Aside: “what the client wants” != “what the client asked for”

- System testing
  - Overall functionality and performance of the system
  - Usable, meets requirements, good performance, etc.
Unit Testing

- **Unit testing**: test each module of a program in isolation
  - “Module” usually means one class
- Helps to catch errors at their source
- For Java, we use the **JUnit** library
  - Framework for automated testing
  - Can quickly run lots (1000s?!?) of tests and see which failed
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it
  - Write “test case” methods in `FooTest` for behavior of `Foo`
  - Each method expects certain results and passes/fails accordingly
Test-Driven Development

• Write the tests before you write the coding being tested!
• Traditional development model for a module Foo
  1. **Design:** specify Foo’s public interface
  2. **Implement:** fill in those methods
  3. **Test:** write & run unit tests
Test-Driven Development

- Write the tests before you write the coding being tested!
- Traditional development model for a module Foo
  1. **Design**: specify Foo’s public interface
  2. **Implement**: fill in those methods
  3. **Test**: write & run unit tests
Test-Driven Development

• Write the tests before you write the coding being tested!
• Traditional development model for a module Foo:
  1. **Design**: specify Foo’s public interface
  2. **Implement**: fill in those methods
  3. **Test**: write & run unit tests

“write tests or add more features?”
Test-Driven Development

• Write the tests before you write the coding being tested!
• Traditional development model for a module Foo:
  1. **Design**: specify Foo’s public interface
  2. **Implement**: fill in those methods
  3. **Test**: write & run unit tests

“looks ok to me…”

“write tests or add more features?”

FEATURE REQUEST
Test-Driven Development

- Write the tests before you write the coding being tested!
- Traditional development model for a module Foo:
  1. **Design**: specify Foo’s public interface
  2. **Implement**: fill in those methods
  3. **Test**: write & run unit tests

"looks ok to me…"

"write tests or add more"

**FEATURE REQUEST** DEADLINE!
Test-Driven Development

- Write the tests before you write the coding being tested!
- Traditional development model for a module $\text{Foo}$
  1. **Design**: specify $\text{Foo}$’s public interface
  2. **Implement**: fill in those methods
  3. **Test**: write & run unit tests

😊
Test-Driven Development

• Write the tests before you write the coding being tested!

• Test-driven development model
  1. **Design**: specify Foo’s public interface
  2. **Test**: write unit tests against that interface
     o Tests will fail initially
  3. **Implement**: fill in methods and verify that tests now pass
  • #2 and #3 are often per-method: choose a method, write tests for it, implement it, repeat
Benefits of Test-Driven Development

- Emphasizes testing
  - Not squeezed against the deadline
  - Likely to produce better test coverage
- Clarifies understanding of how code should work
  - Get to “try out the interface before you commit to it”
  - Identify things you might have overlooked, e.g. boundary cases
  - This way, you only have to write the code once
- You’ll practice test-driven development on HW3
JUnit Semantics

How to write a technically correct JUnit test
A JUnit test class

```java
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
- All `@Test` methods run when JUnit runs your test class.
Verifying Behavior with Assertions

- Assertions: special JUnit methods
- Verifies that a value matches expectations
  
  ```java
  assertEquals(42, meaningOfLife());  \leftarrow fails if meaningOfLife() != 42
  assertTrue(list.isEmpty());        \leftarrow fails if list.isEmpty() is false
  ```

- If the value isn’t what it should be, the test fails
  - Test immediately terminates
  - Other tests in the test class are still run as normal
  - Results show details of failed tests
Using Assertions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertTrue(test)</td>
<td>fails if the boolean test is false</td>
</tr>
<tr>
<td>assertFalse(test)</td>
<td>fails if the boolean test is true</td>
</tr>
<tr>
<td>assertEquals(expected, actual)</td>
<td>fails if the values are not equal</td>
</tr>
<tr>
<td>assertSame(expected, actual)</td>
<td>fails if the values are not the same (by ==)</td>
</tr>
<tr>
<td>assertNotSame(expected, actual)</td>
<td>fails if the values are the same (by ==)</td>
</tr>
<tr>
<td>assertNull(value)</td>
<td>fails if the given value is not null</td>
</tr>
<tr>
<td>assertNotNull(value)</td>
<td>fails if the given value is null</td>
</tr>
</tbody>
</table>

- And others: [http://www.junit.org/apidocs/org/junit/Assert.html](http://www.junit.org/apidocs/org/junit/Assert.html)
- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("message", 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);
    }
}
```
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 uses 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!
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);
    }

    @Test
    public void testAddDaysWrapToNextMonth2() {
        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!

Method names describe function of each object
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);
    
    }

    Let's put it all together!
    That’s it! Test is short & sweet
Checking for Exceptions

- Verify that a method throws an exception when it should
- Place above method:
  ```java
  @Test(expected=IllegalArgumentException.class)
  ```
- Test passes if specified exception is thrown, fails otherwise
- *Only time it’s OK to write a test with no asserts!*

```java
// Try to access the first item in an empty ArrayList
@Test(expected=IndexOutOfBoundsException.class)
public void test() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}
```
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() { ... }
```
DRY (Don’t Repeat Yourself)

- JUnit tests are just regular Java code!
- Can declare fields for frequently-used values or constants

  ```java
  private static final String DEFAULT_NAME = "MickeyMouse";
  private static final User DEFAULT_USER = 
      new User("lazowska", "Ed", "Lazowska");
  ```

- Can write helper methods, etc.

  ```java
  private void eq(RatNum ratNum, String rep) {
      assertEquals(rep, ratNum.toString());
  }
  ```

  ```java
  private BinaryTree getTree(int[] items) {
      // construct BinaryTree and add each element in items
  }
  ```
Unit Test Best Practices

How to craft well-written JUnit tests
#1: Keep tests small

- Ideally, each test only tests one “thing”
  - One “thing” usually means one method under one input condition
- Low-granularity tests help you isolate bugs
  - Tell you exactly what failed and what didn’t
- Only a few (likely one) assert statements per test
  - Test halts after first failed assertion
  - Don’t know whether later assertions would have failed
- 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 `add()` using `contains()`, separately test `contains()` before any items are added
What NOT to do

- IntArrayTest
  (http://www.cs.washington.edu/education/courses/cse331/12wi/section/IntArrayTest.java)
- What’s wrong?
What NOT to do

- IntArrayTest
  
  (http://www.cs.washington.edu/education/courses/cse331/12wi/section/IntArrayTest.java)

- testIntArray tests way too many things
  - Too many methods, array states

- Solution: break down by method being tested and/or state of array
  
  (http://www.cs.washington.edu/education/courses/cse331/12wi/section/IntArrayTestBetter.java)
#2: 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
- 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
#2: Use descriptive asserts, test names

- **Test name:** describe what’s being tested
  - Good: “testAddDaysWithinMonth,” …
  - Not so good: “testAddDays1,” “testAddDays2,” …
  - Useless: “test1,” “test2,” …
  - Overkill:
    - “testAddDaysOneDayThenThenFiveDaysThenNegativeFourDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFEBruary()”
#2: 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
  assertEquals(expected, actual) instead of
  assertTrue(expected == actual)
  (why?)
  ```

  ```java
  assertTrue(b) instead of assertEquals(true, b)
  (why?)
  ```
#2: 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);
}
```
#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 + parameter values + …
- Want to avoid redundancy but still test everything
- What tests do you choose? When do you stop?
  - This is an art!
- **Equivalence classes**: inputs that you expect to cause the same behavior
  - Cause the same lines of code to execute, etc.
- If one input works correctly, expect all others in the equivalence class to also work
#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
- 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
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
    - assertNotNull(true); doesn’t count!
- Tests should be isolated – not dependent on side effects of other tests
- Use helper methods to factor out common operations
  - E.g. setting up initial state of an object
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
Example: IntStack

- What tests should we write?
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.
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.