Unit Testing: part 2

CSCI 2300

Review & Overview

- Last time:
  - Introduced the concept of “unit test”
  - Introduced JUnit – Java framework for unit testing
  - Configured accounts to compile and run JUnit tests
  - Started writing test cases for “Balloon” class
- Today:
  - What makes a good unit test
  - How many test cases do I need?
  - Continue testing “Balloon”

Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());
        assertEquals(2, d.getMonth());
        assertEquals(19, d.getDay());
    }
}
```

Expected value should be on the left

Messages in test cases

```
public class DateTest {
    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days",
              2050, d.getYear());
        assertEquals("month after +14 days",
              3, d.getMonth());
        assertEquals("day after +14 days",
              1, d.getDay());
    }
}
```

Assertions should have messages explaining what is being checked

Provides better failure output

Expected answer objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);
    }
}
```

Use "expected" object to minimize assertions

Date must have toString() and equials() methods

Expected object with message

```
@Test public void test1() {
   Date d = new Date(2050, 2, 15);
   d.addDays(14);
   Date expected = new Date(2050, 3, 1);
   assertEquals("date after +14 days",
      expected, d);
}
```

Naming test cases

Give test case methods really long descriptive names

```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days",
            expected, actual);
    }
}
```

Naming test cases: example 2

```
public class DateTest {
    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days",
            expected, actual);
    }
}
```

Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

JUnit will already show the expected and actual values in its output
Do not repeat them in assertion message

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

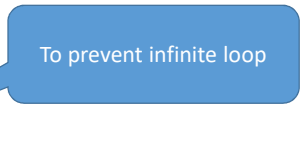
- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Pervasive timeouts




```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days",
            expected, d);
    }

   private static final int DEFAULT_TIMEOUT = 2000;
}
```

How many test cases do I need

- Multiple test cases per method
- Each test case focuses on a particular failure
- Override ‘equals’ method

Example: Balloon class

http://cs.slu.edu/~holdener/csci2300/code/oodp3code/ch03/junit/balloon/index.html

Constructor:

- What happens in we pass a negative integer
- What happens if we pass a maximum integer
- What happens if we pass some positive integer
- How can we confirm that instance variables got set correctly?
  - Can make Balloon variables 'protected' and have the test class extend 'Balloon'
  - If implementation changes, tests will need to be updated
- Test at the "interface level"

Balloon test continued

- getRadius()
  - how do we know this works correctly?
  - What test cases should we have
  - Depends on constructor working correctly
- inflate()
  - Inflate without popping
  - Inflate causing the balloon to pop
  - Inflate causing the balloon to nearly pop (radius == max radius)
  - Inflate causing the balloon to pop exceeding the max radius by only a little bit
  - How do we know if a test case is successful or not

Balloon test continued

- deflate()
  - Deflate completely
  - Deflate a little bit
  - Deflate more than current radius
- What about this:
  - Deflate by 1
  - Deflate by 2
  - Deflate by 3
  - …
  - Deflate by max radius

Bad idea!
Instead, test an average case and edge cases