Cpt S 422: Software Engineering Principles II

Testing levels – Unit Testing – part II

Dr. Venera Arnaoudova

WASHINGTON STATE UNIVERSITY

TDD for isLeap – step 1, DateTest

src/calendar/tests/DateTest.java

```
7   7   import org.junit.BeforeClass;
8   8   import org.junit.Test;
9   9
    10 + import calendar.Date;
    11 +
10  12  public class DateTest {
11  13
12  14      @BeforeClass
67  69
68  70      @Test
69  71      public void testIsLeap() {
70      -       fail("Not yet implemented");
    72  +       assertEquals(true, Date.isLeap(2012));
71  73      }
72  74
73  75      @Test
```

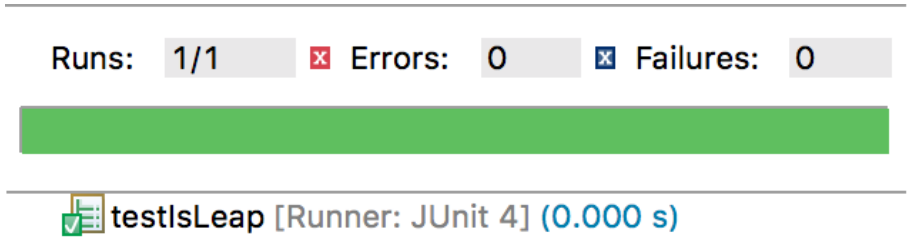
TDD for isLeap – step 1, Date

src/calendar/Date.java

Hunk 1 : Lines 162-171

```
162 162      // thus, 1992, 1996, and 2000 are leap years, while the year
163 163      public static boolean isLeap(int year)
164 164      {
165      -        //TODO
166      -        return true;
    165  +        if(year%4==0) {
    166  +            return true;
    167  +        }
    168  +        return false;
167 169      }
168 170
169 171
```

TDD for isLeap – step 1, run tests



TDD for isLeap

- Repeat the process for steps 3, 4, and 5
- Don’t forget to look for refactoring opportunities
  - Identify possible refactorings (https://www.refactoring.com/catalog/)
  - Run the tests to make sure that you did not brake anything!

isLeap – final result

```java
// A year is a leap year if it is divisible by 4,
// unless it is a century year.
// Century years are leap years only if they
// are multiples of 400 (Inglis, 1961);
// thus, 1992, 1996, and 2000 are leap years,
//while the year 1900 is not a leap year
public static boolean isLeap(int year)
{
    if( (year%4==0) && (year%100!=0) || (year%400==0)){
        return true;
    }
    return false;
}
```

```java
@Test
public void testIsLeap() {
    // divisible by 4:
    assertTrue(Date.isLeap(2012));
    // not divisible by 4
    assertFalse(Date.isLeap(2007));
    // century, not  divisible by 400
    assertFalse(Date.isLeap(1900));
    // century, divisible by 400
    assertTrue(Date.isLeap(2000));
}
```

Testing methods that are coupled

- JUnit does not allow us to test the the behavior of a method in isolation if that method is calling other methods

- To do this we will use Mockito (http://site.mockito.org/)
  - Download the .jar file from:

    https://mvnrepository.com/artifact/org.mockito/mockito-all/1.10.19

  - Include it in your test project

What can we do with Mockito? (1/5)

- Complete mocking can be used to create new objects without worrying about parameters:
  - Pattern: <CN> <on> = mock(<CN>.class);
  - Ex.: Date mockDate = mock(Date.class);

- Partial mocking can be used to wrap real objects:
  - Pattern: <CN> <on> = spy(new <CN>(<ps>));
  - Ex.: Date spyDate = spy(new Date(1,2,2000));

What can we do with Mockito? (2/5)

- Simulate the behavior of a method by returning a specific value <v> when method <m> is called with parameters <ps> on object <o>
  - Pattern: doRetrun(<v>).when(<o>).<m>(<ps>)
  - Ex.: doReturn("Tuesday").when(spyDate).dateToDayName(2,1,2000);

What can we do with Mockito? (3/5)

- Verify that a method <m> was called on object <o> with specific parameters <p1>, <p2>, etc.
  - Pattern: verify(<o>).<m>(Matchers.eq(<p1>),Matchers.eq(<p2>),..);
  - Ex.:
    verify(spyDate).dateToDayNumber(Matchers.eq(2),Matchers.eq(1),Matchers.eq(2000));

What can we do with Mockito? (4/5)

- Verify that a method <m> is
  - Never called: verify(<o>, never()).<m>("never called");
  - Called at least once: verify(<o>, atLeastOnce()).<m>(…);
  - Called at least n times: verify(<o>, atLeast(<n>)).<m>(…));

What can we do with Mockito? (5/5)

- Throw an exception <E> when method <m> is called on object <o>:
  - when(<o>.<m>()).thenThrow(new <E>());

Testing toString()

- The method toString() calls many other methods but we do not want those methods to interfere with the behavior of toString()

```
// Returns the date in the following format:
// "<dayName>, <mm>/<dd>/<yyyy>, is the <dayNumber> of
// the year and the zodiac sign is <zodiacSign>"
public String toString(){
    return this.getDayName() + ", "
            + this.getMm() + "/" + this.getDd() + "/"
            + this.getYyyy() + ", is the "
            + this.getDayNumber()
            + " of the year and the zodiac sign is "
            + this.getZodiacSign();
}
```

testToString()

```java
import static org.mockito.Mockito.*;


@Test
public void testToString() {
    // create a spy for our object Date
    Date spyDate  = spy(new Date(30,8,2017));

    // simulate the behavior for all methods that are called
    // i.e., tell those methods what to return:
    doReturn("Wednesday").when(spyDate).getDayName();
    doReturn(8).when(spyDate).getMm();
    doReturn(30).when(spyDate).getDd();
    doReturn(2017).when(spyDate).getYyyy();
    doReturn(242).when(spyDate).getDayNumber();
    doReturn("Virgo").when(spyDate).getZodiacSign();

    // test toString on the spy object
    assertEquals("Wednesday, 8/30/2017, is the 242 of "
            + "the year and the zodiac sign is Virgo",spyDate.toString());

}
```

Tasks for today

1. Refactor your code to match the partial implementation of Date-v.2-partial.java (using the refactoring feature of Eclipse!)
2. Implement testInitializeAttributes() to test method initializeAttributes(int dd, int mm, int yyyy)
3. Refactor Date.java as follows (and **adapt your tests at each step!**):
   - Check if a date is valid before creating an object
   - ~~static~~ dateToDayName(~~int mm,int dd,int yyyy~~)
   - ~~static~~ dateToDayNumber(~~int mm,int dd,int yyyy~~)
   - ~~static~~ lastDayOfMonth(~~int mm,int yyyy~~)
   - zodiacSign(~~int mm, int dd~~)
   - initializeAttributes(~~int dd, int mm, int yyyy~~)