Unit Testing Laboratory

CSCE 747 - Lecture 18 - 03/17/2016

Today's Class

- We've covered many testing techniques.
- We've covered the basics of writing executable test cases.
- Today - we put those lessons into practice.
  - We will work together to test a sample system.

Enter… The Planning System

- Everybody likes meetings.
  - Not true - but we need to book them.
- We don't want to double-book rooms or employees for meetings.
- System to manage schedules and meetings.

The Planning System

Offers the following high-level features:

1. Booking a meeting
2. Booking vacation time
3. Checking availability for a room
4. Checking availability for a person
5. Printing the agenda for a room
6. Printing the agenda for a person

Your Task

In groups, come up with a test plan for this system.

- Given the above features and the code documentation, plan out a series of test cases to ensure that these features can be performed without error.

Food for Thought

- What are the “testable units”?
  - Your tests may use any of the classes in the system, and may be at the method, class, or system level.
- Think about both normal execution and illegal inputs/actions.
  - How many things can go wrong?
  - You will probably be able to add a normal meeting, but can you add a meeting for February 35th?
  - Try it out - you have the code.

Unit Testing

Writing a Unit Test

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.
- Write a “testing class” containing a series of unit tests centered around testing that target.

```
public class Calculator {
  public int evaluate (String
                expression) {
    int sum = 0;
    for (String summand:
                expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Writing a Unit Test

```
public class Calculator {
  public int evaluate (String
                  expression) {
    int sum =
    for (Stri
              expression.split("
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

```
import static org.junit.Assert.
assertEqu
import or

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator =
        new Calculator();
    int sum =
        calculator.evaluate("1+2+3");
    assertEquals(6, sum);
     calculator = null;
  }
}
```

Convention - name the test class after the class it is testing or the functionality being tested.

Each test is denoted with keyword **@test**.

Initialization

Input

Test Steps

Oracle

Tear Down

Test Fixtures - Shared Initialization

@Before annotation defines a common test initialization method:

```
@Before
public void setUp() throws Exception
{
   this.registration = new Registration();
   this.registration.setUser(“ggay”);
}
```

Test Fixtures - Teardown Method

@After annotation defines a common test tear down method:

```
@After
public void tearDown() throws Exception
{
   this.registration.logout();
   this.registration = null;
}
```

Test Skeleton

@Test annotation defines a single test:

```
@Test
public void test<MethodName><TestingContext>() {
    //Define Inputs
    try{ //Try to get output.
    }catch(Exception error){
        fail("Why did it fail?");
    }
    //Compare expected and actual values through
assertions or through if statements/fails
}
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

- assertEquals, assertArrayEquals
- assertFalse, assertTrue
- assertNull, assertNotNull
- assertSame,assertNotSame
- assertThat

Testing Exceptions

- When testing error handling, we expect exceptions to be thrown.
- In JUnit, we can ensure that the right exception is thrown.

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
     new ArrayList<Object>().get(0);
}
```

Your Task

- Translate planned tests into executable jUnit tests.
  - If a test is supposed to cause an exception to be thrown. Make sure you check for that exception.
  - Make sure that your expected output is detailed enough to ensure that - if something is supposed to fail - that it fails for the correct reasons.

Finding Faults

Did You Find the Faults?

1: getMeeting and removeMeeting perform no error checking on dates.

```
public Meeting getMeeting(int month, int day, int index){
    return occupied.get(month).get(day).get(index);
}

public void removeMeeting(int month, int day, int index){
    occupied.get(month).get(day).remove(index);
}
```

Did You Find the Faults?

2: Calendar has a 13th month.

```
public Calendar(){
        occupied = new
ArrayList<ArrayList<ArrayList<Meeting>>>();

        for(int i=0;i<=13;i++){
            // Initialize month
            occupied.add(new ArrayList<ArrayList<Meeting>>());
            for(int j=0;j<32;j++){
                // Initialize days
                occupied.get(i).add(new ArrayList<Meeting>());
            }
        }
```

Did You Find the Faults?

3: November has 30 days.

Oh - and we just added a meeting to a day with a date that does not match that date.

```
occupied.get(11).get(30).add(new Meeting(11,31,"Day does not exist"));
```

Did You Find the Faults?

4: Used a >= in checking for illegal times.

December no longer exists.

```
if(mMonth < 1 || mMonth >= 12){
            throw new TimeConflictException("Month does not
exist.");
}
```

Did You Find the Faults?

5: We should be able to start and end a meeting in the same hour.

```
if(mStart >= mEnd){
    throw new TimeConflictException("Meeting starts before it
ends.");
}
```

What Other Faults Did You Find?

Code Coverage

- What level of coverage did our tests achieve over the system?
- How can we cover the gaps?

Next Time

- Fault-Based Testing
  - Using ideas about what could go wrong to guide testing.
  - Related reading - Chapter 16

- Homework 3 - due tonight.
- Reading assignment 4 - due next week.