CSE 143 Java

Testing and JUnit

Reading: www.junit.org; DrJava JUnit Help

Testing & Debugging

- Testing Goals
  - Verify that software behaves as expected
  - Be able to recheck this as the software evolves
- Debugging
  - A controlled experiment to discover what is wrong
  - Strategies and questions:

    What's wrong?

    What do we know is working? How far do we get before something isn't right?

    What changed?

    (Even if the changed code didn't produce the bug, it's fairly likely that some interaction between the changed code and other code did.)

Unit Tests

- Idea: create small tests that verify individual properties or operations of objects
  - Do constructors and methods do what they are supposed to?
  - Do variables and value-returning methods have the expected values?
  - Is the right output produced?
- Lots of small unit tests, each of which test something specific; not big, complicated tests
  - If something breaks, the broken test is a clue about where the problem is

Writing Tests

- When?

  Before you write the code!!!

- Say what? Why would you do that?
  - Helps you understand the problem and think about code design and implementation
  - Gives you immediate feedback once the code is written

Where to put the tests?

- DrJava's interactions window
  - Great way to prototype tests
  - Way too tedious to do any extensive testing
- Main methods
  - Either too many to do a thorough job, or
  - Methods that test too much – hard to isolate problems
- Either way, someone has to check the output
- Better: automate this by writing self-checking tests

JUnit

- Test framework for Java Unit tests
- Idea: implement classes that extend the JUnit TestCase class
- Each test in the class is named testXX (name starting with "test" is the key)
- Each test performs some computation and then checks the result
- Optional: setUp() method to initialize instance variables or otherwise prepare before each test
- Optional: tearDown() to clean up after each test
  - Less commonly used than setUp()

Example (from DrJava help)

- Tests for a simple calculator object

```
import junit.framework.TestCase;
public class CalculatorTest extends TestCase {

    public void testAddition() {
      Calculator calc = new Calculator();
      int expected = 7;
      int actual =  calc.add(3, 4);
      assertEquals("adding 3 and 4", expected, actual);
    }
    …
}
```


Another Calculator Test

```
public void testDivisionByZero() {
   Calculator calc = new Calculator();
   try {                                  // exception handling – coming attraction
     calc.divide(2, 0);
     fail("should have thrown an exception");
   } catch (ArithmeticException e) {
     // do nothing – this is what we expect
   }
}
```

What Kinds of Checks are Available

- Look in junit.framework.Assert (JavaDocs on www.junit.org)
- Examples

```
assertEquals(expected, actual);   // works on any type except double
                                  // uses .equals() for objects
assertEquals(messsage, expected, actual);  // all have variations with messages
assertEquals(expected, actual, delta);     // for doubles to test "close enough"
assertFalse(condition);
assertTrue(condition);
assertNotNull(object);
assertNull(object);
fail();
// and some others
```


setUp

- If the tests require some common initial setup, we can write this once and it is automatically executed before each test (i.e., each test starts with a fresh setUp)

```
import junit.framework.TestCase;
public class CalculatorTest extends TestCase {
    private Calculator calc;          // calculator object for tests
    /** initialize for each test */
    protected void setUp() {
       calc = new Calculator();
    }

    // tests as before, but without local declaration/initialization of calc
```


Summary

- Unit tests – a key to robust software
  - Verify correct operation of new code
  - Repeated running of tests as code changes increases confidence that changes don't introduce bugs
    (or makes it much easier to track down problems that do occur)
  - Tests become part of the project history/culture
- Write the tests before you write the code
- If you discover a bug you didn't test for, add a test
- A little up-front effort will pay off in much better quality code and much less time tracking down problems