CMSC131

Testing Overview

JUnit Testing

Software Testing

- Testing is a critical part of the design and implementation process.
- There are different types of testing with different goals…
- Ideally you want to test all possible ways your program can run and confirm that the output and side effects are all correct.
- Testing individual modules helps identify flaws in the "foundation" that should be fixed before other things are built upon it.

Regression Testing

What happens if you thoroughly test a module and then decide it needs to be modified at a later time?

- You might want to retest the entire module.
- You at least want to retest anything that depends on the things that were changed.

It would be useful to have a way of "saving" your testing scenarios and the checks used to confirm that the output and side effects are correct.

Module Testing Approaches

There are many approaches..

- Manually run through scenarios and check results.
- Write test drivers and various input and output files for comparison of actual to expected output.
- Use one of the xUnit family of tools such as…
  - JUnit in Java
  - NUnit in .NET
  - CppUnit in C++
  - etc.

There are research projects exploring new and different testing approaches (such as the GUITAR and ICE projects here at UM).

JUnit Testing (I)

We will now see a resource available in Java called JUnit testing.

Each test is a method of the form:

```
@Test
public void testTest() {

   //Code and tests here…
}
```

JUnit Testing (II)

Two basic test assertions are:

assertTrue(*boolean_expression*);

assertFalse(*boolean_expression*);

These can appear anywhere in the body of a testing method.

If the assertion test succeeds, execution of the body continues. If it fails, the test fails and execution of the body of that test stops but if there are more test methods they are run.

JUnit Testing (III)

assertSame(*var_1, var_2*);

This checks whether these two variables contain the same values. For primitives this acts in a slightly unusual way behind the scene but acts as expected. For object references, this essentially checks whether the two object references "point" to the same memory location.

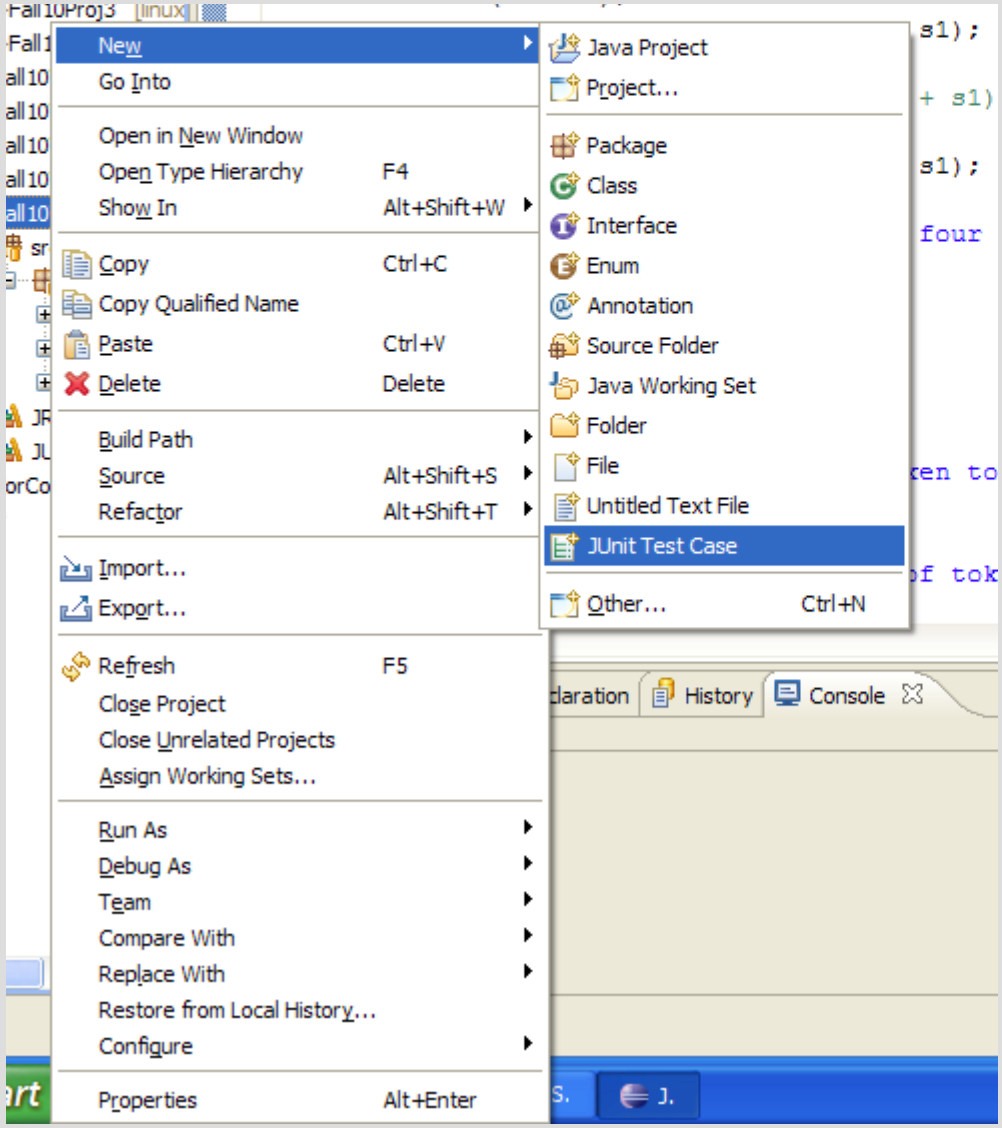
JUnit Testing (IV)

assertEquals(*var_1, var_2*);

For primitives this acts as expected. For object references this is an "interesting" one because it uses a very specific form of the **`equals`** operator to check whether these two variables are equal. If the class of the objects passed in implement the *exact* **`equals`** operator Java expects (which our self-written Student class doesn't yet) it works as expected.

NOTE: We will explore this more later…

Add a JUnit Test Case



A Test Set is a Class

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

public class MyTestCases{

//You put your test cases here.  Each test
//  method needs to have @Test above it.




}
```

Constructors, Setters and Getters

```
@Test
public void test1name() {
  String n = "Fred";
  StudentTwo s = new StudentTwo(n, 6, 1234568);
  assertTrue(s.getName().equals(n));
}
```

Other Methods

```
@Test
public void test1ident() {
  StudentTwo s =
     new StudentTwo("Fred", 6, 1234508);
  assertTrue(s.getLastFourIdentDigits()==4508);
}
```

Test Scenarios (I)

```
@Test
public void test1tokens() {
  StudentTwo s =
     new StudentTwo("Fred", 6, 1234568);

  for (int i=0; i<4; i++) {
     s.useToken();
  }

  assertTrue(s.getTokenLevel() == 2);
}
```

Test Scenarios (II)

```
@Test
public void test2tokens() {
  StudentTwo s =
     new StudentTwo("Fred", 3, 1234568);

  for (int i=0; i<4; i++) {
     s.useToken();
  }

  assertTrue(s.getTokenLevel() == 0);
}
```

Test Scenarios (III)

```
@Test
public void test3tokens() {
  StudentTwo s =
     new StudentTwo("Fred", 3, 1234568);

  for (int i=0; i<4; i++) {
     s.useToken();
  }

  assertFalse(s.useToken());
}
```

What to test? How to design tests?

“Test Everything"

- Make sure that *every* method is tested (you only have direct access to public methods, but you need to think about how to test the private ones as well).
- Think about the "corner cases“ (lowest/first, highest/last, etc.).
- Try some random combinations of scenarios.
- Test for success but also test for error cases that are meant to be handled.
- It would be good if you could make sure that every decision branch is tested.

What to test? How to design tests?

It’s a good idea to write your tests (or at least some of them) based on your specs rather than writing them after you implement a module to avoid

When working with your clients or managers it can also be useful to discuss your test cases with them and see which scenarios they might notice as missing.

What percentage of project cost is testing?

1. Less than 10%
2. 10%-25%
3. 25%-40%
4. 40%-60%
5. More than 60%