CMSC131

Testing Overview

JUnit Testing

Software Testing

Testing is a critical part of the program design and implementation process, and 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 can help 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 or NUnit in .NET or CppUnit in C++, etc.

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

How much time?

Think about your work on project 2.

- How much time was flowcharting and planning?
- How much time was turning that into Java code?
- How much time was debugging the resulting Java code?

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
11/13 © 1995 United Feature Syndicate, Inc.(NYC)

http://dilbert.com/strip/1995-11-13

What do you think the percentage of project time/cost testing takes?

20%	1. Less than 10%
20%	2. 10%-25%
20%	3. 25%-40%
20%	4. 40%-60%
20%	5. More than 60%

Response Counter

JUnit Testing (I)

We will now see a resource available in Java called JUnit testing. Note: Automated testing is only one part of the overall process of testing and quality assurance. Each test is a method of the form:

```
@Test
public void testTest() {
   //Code and tests here…
}
```

JUnit Testing (II)

Two basic test assertions sometimes used are:

assertTrue(*boolean_expression*);

assertFalse(*boolean_expression*);

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

If an 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 still 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.

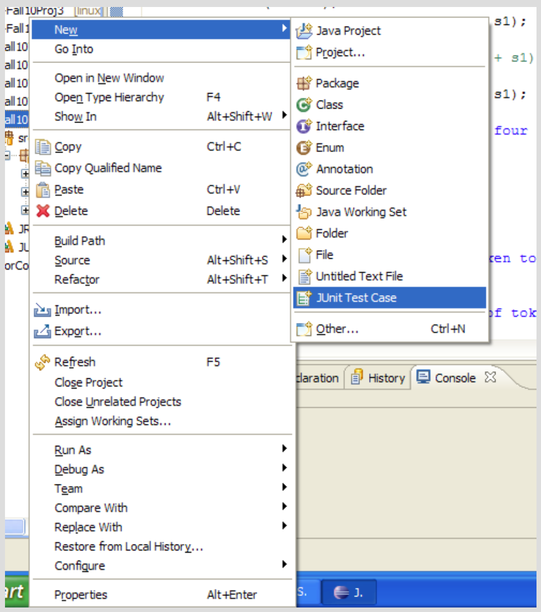
JUnit Testing (IV)

`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 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.



}
```

What to test? How to design tests?

“Test Everything“ (very hard, but a useful target)

– 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.

We don’t need source code to test a module, we just need the names of the public methods…

Example: Testing BigBoxOfInts

```
BigBoxOfInts()
  Constructor builds a BigBoxOfInts that starts empty.

void    addToBox(int newVal)
  This method adds the given int to the BigBoxOfInts.

int     howManyStored()
   This method returns the number of things stored in the BigBoxOfInts.

void    removeAllFromBox(int delVal)
   This method deletes all copies of the given int from the
   BigBoxOfInts.

void    removeOneFromBox(int delVal)
   This method deletes one copy of the given int from the BigBoxOfInts.

java.lang.String      toString()
  Returns a string representation of the BigBoxOfInts.
```

Test creating and simple adding.

```
@Test
public void testStartsEmpty() {
BigBoxOfInts myList = new BigBoxOfInts();
  assertEquals(0, myList.howManyStored());
}

@Test
public void testAddOneThing() {
  BigBoxOfInts myList = new BigBoxOfInts();
  myList.addToBox(5);
  assertEquals(1, myList.howManyStored());
  assertEquals("[5]",myList.toString());
}
```

Test adding and removing.

```
@Test
public void testAddOneThingAndRemoveIt() {
      BigBoxOfInts myList = new BigBoxOfInts();
      myList.addToBox(5);
      myList.removeOneFromBox(5);
      assertEquals(0, myList.howManyStored());
      assertEquals("[]",myList.toString());

      myList.addToBox(5);
      myList.removeAllFromBox(5);
      assertEquals(0, myList.howManyStored());
      assertEquals("[]",myList.toString());
}
```

More tests of adding and removing.

```
@Test
public void testAddManyThingAndRemoveOne() {
      BigBoxOfInts myList = new BigBoxOfInts();
      for (int val=0; val<5; val++) {
            myList.addToBox(val);
      }
      myList.removeOneFromBox(2);
      assertEquals(4, myList.howManyStored());
      assertEquals("[0, 1, 3, 4]",myList.toString());
}
```

And test some more...

```
@Test
public void testAddManyThingAndRemoveSeveral() {
      BigBoxOfInts myList = new BigBoxOfInts();
      for (int val=0; val<5; val++) {
            myList.addToBox(val);
      }
      myList.removeOneFromBox(2);
      myList.removeOneFromBox(0);
      myList.removeOneFromBox(4);
      assertEquals(2, myList.howManyStored());
      assertEquals("[1, 3]",myList.toString());
}
```

And test even more...

```
@Test
public void testAddManyWithDupsAndRemove() {
          BigBoxOfInts myList = new BigBoxOfInts();
          int copies=8;
          for (int multiple=0; multiple<copies; multiple++) {
                    for (int val=0; val<5; val++) {
                              myList.addToBox(val);
                    }
          }
          myList.removeAllFromBox(2);
          myList.removeAllFromBox(0);
          myList.removeAllFromBox(4);
          assertEquals(2*copies, myList.howManyStored());

          String shouldBe = "[";
          for (int multiple=0; multiple<copies; multiple++){
                    shouldBe += "1, ";
          }
          for (int multiple=0; multiple<copies-1; multiple++){
                    shouldBe += "3, ";
          }
          shouldBe += "3]";
          assertEquals(shouldBe,myList.toString());
}
```

Some challenges with testing…

In 1990, software engineer / author Boris Beizer wrote about what he called "*The Pesticide Paradox*"

– The idea is that writing a good set of tests should help detect bugs, but it will likely miss some, and running the tests over and over won't be of any use in finding those particular bugs.

NOTE: Your tests themselves might contain bugs.

Also note that testing can't really show that bugs ***don't*** exist, just that the scenarios you are testing it the way you are testing them don't fail, which is still important and shows why test scenario selection is important…

Copyright © 2010-2018 : Evan Golub