5.2. JUnit

http://www.cis.upenn.edu/~matuszek/cit597-2003/Lectures/19-junit.ppt

19-Mar-11

Test suites

- Obviously you have to test your code to get it working in the first place
  - You can do *ad hoc* testing (running whatever tests occur to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

2

Test suites (2)

- Disadvantages of a test suite
  - It’s a lot of extra programming
    - This is true, but use of a good test framework can help quite a bit
  - You don’t have time to do all that extra work
    - *False*--Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
  - Reduces total number of bugs in delivered code
  - Makes code much more maintainable and refactorable
    - This is a *huge* win for programs that get actual use!

XP approach to testing

- In the Extreme Programming approach,
  - Tests are written before the code itself
  - If code has no automated test case, it is *assumed not to work*
  - A test framework is used so that automated testing can be done after every small change to the code
    - This may be as often as every 5 or 10 minutes
  - If a bug is found after development, a test is created to keep the bug from coming back
- Consequences
  - Fewer bugs
  - More maintainable code
  - Continuous integration--During development, the program *always works*--it may not do everything required, but what it does, it does right

Rhythm

- "There is a rhythm to developing software unit tests first.

  You create one test to define some small aspect of the problem at hand.

  Then you create the simplest code that will make that test pass.

  Then you create a second test.

  Now you add to the code you just created to make this new test pass, but no more!

  Not until you have yet a third test.

  You continue until there is nothing left to test."
  - http://www.extremeprogramming.org/rules/testfirst.html

JUnit

- JUnit is a framework for writing unit tests
  - JUnit was written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
  - JUnit uses Java's reflection capabilities (Java programs can examine their own code)
  - JUnit helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version
  - JUnit is not yet (as far as I know) included in Sun's SDK, but an increasing number of IDEs include it
  - BlueJ, JBuilder, and Eclipse now provide JUnit tools

Terminology

- A test fixture sets up the data (both objects and primitives) that are needed to run tests
  - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a *single* class
- A test case tests the response of a single method to a particular set of inputs
- A test suite is a collection of test cases
- A test runner is software that runs tests and reports results

- An integration test is a test of how well classes work together
  - JUnit provides some limited support for integration tests

What JUnit does

- JUnit runs a suite of tests and reports results
- For *each* test in the test suite:
  - JUnit calls `setUp()`
    - This method should create any objects you may need for testing
  - JUnit calls *one* test method
    - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
    - In fact, since it's your code, the test method can do anything you want
    - The `setUp()` method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you
  - JUnit calls `tearDown()`
    - This method should remove any objects you created

Structure of a JUnit test class

- Suppose you want to test a class named `Fraction`
- `public class FractionTest`
  `extends junit.framework.TestCase {`
  - This is the unit test for the `Fraction` class; it declares (and possibly defines) values used by one or more tests
- `public FractionTest() { }`
  - This is the default constructor
- `protected void setUp()`
  - Creates a test fixture by creating and initializing objects and values
- `protected void tearDown()`
  - Releases any system resources used by the test fixture
- `public void testAdd()`, `public void testToString()`, etc.
  - These methods contain tests for the Fraction methods `add()`, `toString()`, etc. (note how capitalization changes)

9

Assert methods I

- Within a test,
  - Call the method being tested and get the actual result
  - assert what the correct result should be with one of the provided assert methods
  - These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an `AssertionFailedError` if the test fails
  - JUnit catches these Errors and shows you the result
- `static void assertTrue(boolean` ***test***`)`
  `static void assertTrue(String` ***message***`, boolean` ***test***`)`
  - Throws an `AssertionFailedError` if the test fails
  - The optional *message* is included in the Error
- `static void assertFalse(boolean` ***test***`)`
  `static void assertFalse(String` ***message***`, boolean` ***test***`)`
  - Throws an `AssertionFailedError` if the test fails

10

Example: Counter class

- For the sake of example, we will create and test a trivial "counter" class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value
- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;

    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    protected void tearDown() { } // no resources to release

    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

The Counter class itself

```
public class Counter {
    int count = 0;

    public int increment() {
        return ++count;
    }

    public int decrement() {
        return --count;
    }

    public int getCount() {
        return count;
    }
}
```

- Is JUnit testing overkill for this little class?
- The Extreme Programming view is: If it isn't tested, assume it doesn't work
- You are not likely to have many classes this trivial in a real program, so writing JUnit tests for those few trivial classes is no big deal
- Often even XP programmers don't bother writing tests for *simple* getter methods such as `getCount()`
- We only used assertTrue in this example, but there are additional assert methods

Assert methods II

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - This method is heavily overloaded: *arg1* and *arg2* must be both objects *or* both of the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as `public boolean equals(Object o)`--otherwise it uses ==

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects refer to the same object (using ==)

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects do not refer to the same object

Assert methods III

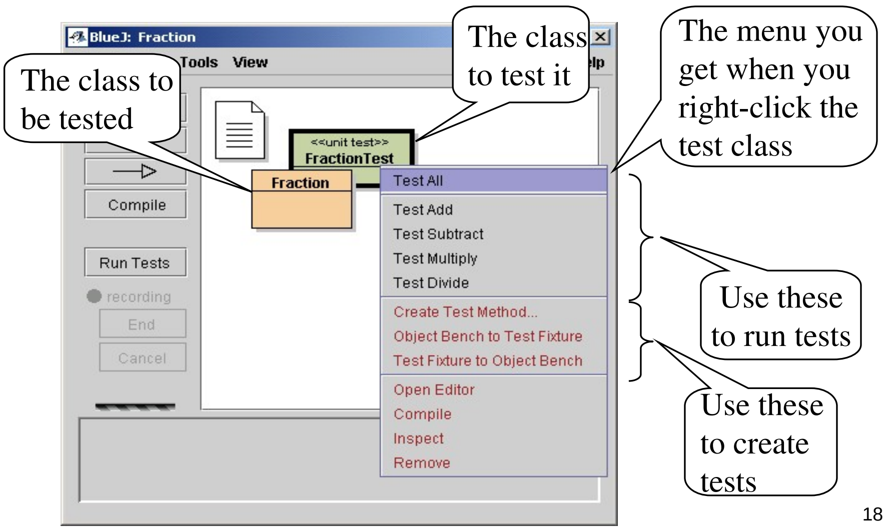
- `assertNull(Object object)`
  `assertNull(String message, Object object)`
  - Asserts that the object is null
- `assertNotNull(Object object)`
  `assertNotNull(String message, Object object)`
  - Asserts that the object is null
- `fail()`
  `fail(String message)`
  - Causes the test to fail and throw an `AssertionFailedError`
  - Useful as a result of a complex test, when the other assert methods aren’t quite what you want

BlueJ

- BlueJ 1.3.0 provides support with commands such as `Create Test Class` and `Create Test Method`
- You can create objects on the test bench and move them to the fixture (and back again)
- BlueJ also has a “recording” mode where you create and manipulate objects, and BlueJ turns your actions into test code
  - This is a first implementation and is still quite buggy
  - It’s worth experimenting with, but you have to check the code produced to see if it makes sense
- BlueJ also makes it easy to run a single test, a suite of tests, or all tests
- BlueJ’s display of JUnit results is virtually identical to the way results are displayed by commercial IDEs, such as JBuilder

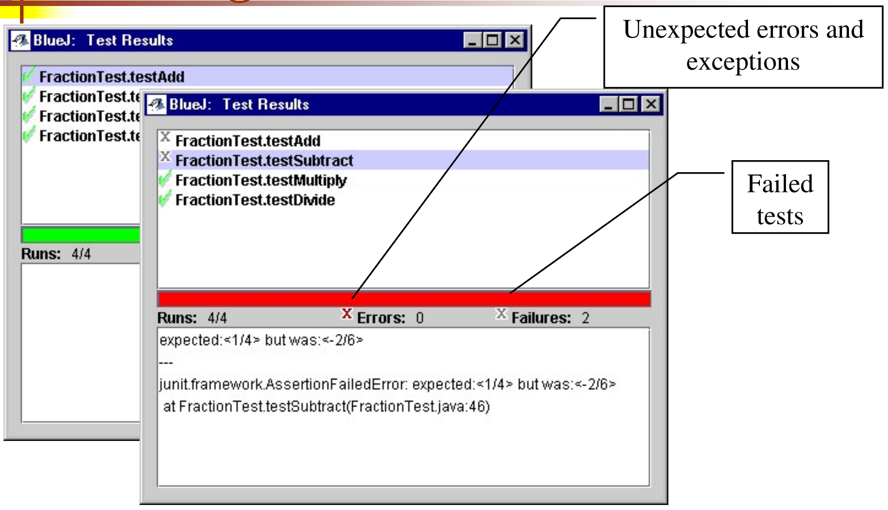
Test classes in BlueJ

- To use JUnit in BlueJ, go to the Preferences... and check Show Unit Testing Tools



18

Viewing test results



```
expected:<1/4> but was:<-2/6>
---
junit.framework.AssertionFailedError: expected:<1/4> but was:<-2/6>
 at FractionTest.testSubtract(FractionTest.java:46)
```

If you run a single test, and it is successful, you just get a message in the status line

20

Creating a test class in BlueJ

- If you have an existing class, right-click on it and choose Create Test Class
  - If your class is named MyClass, the new test class will be named MyClassTest
- To create the test class first, just choose New Class… and give the test class the name of a future class, with 'Test' appended
  - Later, after you create the class to be tested,you can right-click it and choose Create Test Class
    - BlueJ will complain that the class already exists, but it will also correctly associate the test class with the class to be tested

21

Creating the setUp() method

- BlueJ has an Object Bench (at the bottom of the main window)
  - You can create objects on the Object Bench by right-clicking on a class and choosing one of its new constructors
- You can right-click on a test class and choose:
  - Object Bench To Test Fixture or
  - Test Fixture To Object Bench
- Since setUp() is your code, you can modify it any way you like (such as creating new objects in it)

22

Implementing the tearDown() method

- In most cases, the tearDown() method doesn't need to do anything
  - The next time you run setUp(), your objects will be replaced, and the old objects will be available for garbage collection
  - It doesn't hurt to set to null the objects you created in setUp()
  - Like the finally clause in a try-catch-finally statement, tearDown() is where you would release system resources (such as streams) that might not otherwise be released

23

Recording test cases

- An easy way to create a test method is to right-click a *compiled* test class and choose Create Test Method…
  - Enter the name of the method you want to test; you don't have to say "test"
    - BlueJ will capitalize your method name and prefix it with test
  - If you wish, you can copy Test Fixture To Object Bench
  - Use BlueJ to make calls to the method
    - After each call, BlueJ will tell you its result, and ask you to type in the result you expected
      - The result can be equal to, the same as, not the same as, null, not null, or equal to (double or float)
    - You can even create a new object to use as a result
  - When you are done click the End button (under Recording)
- *Review the results!*
  - This is a new feature in BlueJ, and sometimes it produces bad syntax
  - A comma in your expected result *will* confuse BlueJ 1.3.0

24

The structure of a test method

- A test method doesn't return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an `AssertionFailedError`
  - Hence, a test method just calls some assertion method, any of which may throw an `AssertionFailedError`
- The JUnit framework catches the error and deals with it; you don't have to do anything

25

Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This ignores:
    - Programs that do work in response to GUI commands
    - Methods that are used primary to produce output
- I think heavy use of JUnit encourages a "functional" style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

26

The GUI problem

- The whole point of JUnit is to make testing *easy*
  - You click a button, and all your tests “just happen”
- To test a GUI, you have to sit there and interact with the GUI--*not good!*
- You can “automate” GUI use by “faking” events
  - Here’s a starter method for creating your own events:

    -
      ```
      public void fakeAction(Component c) {
          getToolkit().getSystemEventQueue().postEvent(
              new ActionEvent(c, ActionEvent.ACTION_PERFORMED,
      ""));   }
      ```
  - You can explore the Java API to discover how to create other kinds of events

The printing problem

- If a method does output to the screen or to a file, you want to make sure the output is correct
  - Again, you want to set this up once, and forever after have it all done *automatically*
- How can you capture output?
  - Rather than always printing on `System.out`, you can do your printing on an arbitrary `PrintStream`
  - The `PrintStream` can be passed into methods as a parameter
  - Alternatively, you can redefine `System.out` to use a different PrintStream with `System.setOut(`***PrintStream***`)`
  - A Java `PipedOutputStream` can be connected directly to a `PipedInputStream`; this might be helpful
- Whatever you do, you would like to minimize the effect on the program you are trying to test

The End