CS108, Stanford
Fall, 2007-08

Handout #6
Nick Parlante

Unit Testing

This handout introduces the ideas of unit testing and looks at using JUnit in Eclipse.

Unit Testing Theory

Software Engineering Challenges -- Complex Systems

- With Java, we're doing much better at code re-use and avoiding simple memory bugs.
- Large software systems remain hard to build and bugs are the main problem.
- Complexity -- it can be hard to make a change in a large system because it's hard to know if that change introduces a bug off in some other part of the system. Modularity helps address this problem, but it's still a problem.

Unit Testing

- For each piece of "work" code -- a class or a method, pair the work code with some "unit test" code
- The unit test code calls the work code through its public API, calling it a few different ways and checking the results.
- Test code does not need to be **exhaustive** -- test code adds a lot of value even just hitting a few fairly obvious cases.
- Unit tests are a standard, maintained way to keep tests in parallel with the work code
- vs. the more one-off, informal way of doing some testing here and there manually
- The unit tests are more of an investment -- effort to build, but then they endure, improving development for the lifetime of the code.

Fast Feedback -- A Boundary

- Unit test code is some work to install, but then it can be run very easily
- That the tests can be run easily guides your coding -- you get fast feedback as you try out ideas.
- The Unit tests are a like a boundary set up right at the edge of the space of correct operation of the code. Once the boundary is there and is run automatically, it provides quick valuable feedback when bugs are introduced.
- Why is a supertanker hard to steer? Because you turn the wheel, but don't get real feedback for many seconds. Unit tests are about tightening the feedback loop. It's easier to work in a system when you get instant feedback when you go over some boundary.

High Quality Code

- We think about building lots of different types of code -- throw away code to production code (not everything needs to be super high quality of course)
- Historically, code was built to an intuitive "it appears to work when I run it" quality level
- With unit tests, we have the possibility to building code to a much higher quality level -- because we have the tests, and the infrastructure can run the tests constantly.
- The advantages multiply if we put together many components, each of which has its own unit tests
- Unit tests have raised the standard of what it means to build up a system of high-quality code

Test Driven Development (TDD) -- Workflow

- The "Test Driven Development" (TDD) style is a code development workflow that really emphasizes testing. Write the test code first, then write the work code and debug it until the tests pass.

- Every feature has corresponding unit test code. So a for a foo() method, we have test code that calls foo() a few different ways and checks the results (details below).

Unit Testing Advantages

- Can yield code of enormously higher quality compared to code that is just built with programmer ad-hoc testing.
- With the test-first style, you get a chance to frame the problem and think about the cases before getting to the work code. This can provide a nice ramp-up to writing the code compared to staring at a blank screen.
- When writing the work code, you had to think about various cases anyway. Later on, your forget them, and certainly other people using the code do not get the benefit of them. With unit tests, you record those ideas once and get their benefit for the lifetime of the code.

Unit Test Change Freedom

- With unit tests in place, it is easier to make changes/experiments and see what works. It's hard to do that with a large system without good tests -- you are afraid to make a change.
- In this sense unit tests create freedom for a project -- decisions do not need to be set in stone early in the project, because we have the capacity to make changes safely late in the project.

Unit Test Tax -- Costly?

- Writing tests certainly adds some fraction more work 20%? 50%? -- it's certainly more code to write and think about.
- However it certainly increases quality, reduce time that goes into debugging.
- Debugging can be such a time sink ... the 20%-50% tax can look very cheap.

Unit Test Types

- I think of unit tests as falling in roughly three categories
- **Basic** -- cases with small to medium sized inputs that are so simple they should obviously work. These should not be hard to think of, so I do them first and they should not take long.
- **Advanced** -- harder, more complex cases. Some of these, you only think of later on as you get deeper into the algorithm. This is the category that tends to grow over time as you get more insight about the problem and observe more weird cases.
- **Edge** -- there are also cases that are simple but represent edge conditions -- the empty string, the empty list, etc.

Unit Tests -- Basic

- Suppose you have a foo() method to test.
- The test code for foo() will be in methods with names like testFooBasic(), testFooAdvanced(), testFooEdge() (see examples below)
- As you first think about the problem, think of some obvious, basic cases where it should work. Do not block trying to think of exotic cases -- just get some basic ones down for your basic unit tests.
- Writing basic unit tests…
  - Unit tests should use the regular, public interface of the class used by regular client code -- call things, test results
  - Start with easy, obvious cases just to get started
  - Writing out the cases helps firm up the issues in your mind -- a nice, easy way to get started before writing the work code.
- At first the tests fail, since the work code is not written -- unit tests can fail by not even compiling -- that's fine at this stage
- Now start writing the work code, eventually the tests pass, yay!
- Don't get caught up trying to write a unit test for every possible angle -- just hit a few good ones. Tests do not need to try every possible input -- just a few.

Call Every Method A Few Times Differently

- If a class has foo() and bar() methods, the test code should call each of those a few different ways -- say 5 calls.
- Don't call foo() 5 times, but where the calls are very similar.
- When testing a equals(x, y) method -- don't only give x,y where equals() returns true -- call it once or twice where it should return false too!
- If someone has changed the method body to something like "return false;" .. the unit tests should at least be able to notice that.

Evolve Trick

- This is just one technique to get started. I start with the code for my first test.
- Then the later tests, I copy/paste the first test, and then adjust the inputs/expected-output to try to push on the code in a slightly different way.
- In this way, the series of tests "evolve", each being a variation on the one above.
- The disadvantage here can be that the tests are too similar, but it's a convenient way to get started.

Strangely Compelling

- The cycle of working at the code until tests pass is strangely enjoyable. You hit the run button, and it's a little charge when the tests finally run green. I think this is because you get a concrete sense of progress vs. the old style of working on code but not really knowing if you are getting anywhere.

Unit Tests -- Advanced

- At some point, it's time to add harder test cases -- that hit more complex, subtle problem cases.
- You may be able to think of these outside of the implementation.
- Or often, at some point during the implementation, you notice some issue, like "oh, I need to handle the case where A and B are different lengths". You can write a unit test that captures that particular issue, and then go back to the implementation until the test passes.
- As you come across cases you had not thought about, go add a unit test first, then add the code for that case
- If you need some tricky if-logic in the work code to distinguish two cases, write a unit test that pushes on exactly that boundary to see that its right.
  - e.g. you figure out how some case in the code needs to be < vs. <= -- you can encode that case in a unit test to see that you have it right.

Unit Tests vs. API Design

- API design -- that a class presents a nice interface for use by others -- is vital part of OOP design.
- API design is hard, since it's difficult for the designer to understand the class and its API the way they will appear to clients.
- Unit tests have the virtue of making the designer literally act like a client, using the class in a realistic way using only its public API. In this way, the unit tests help the designer to see if the public API is awkward for expressing common cases. By writing tests first, this insight about the API appears very early in the life of the class when it's easy to change or tune.

Unit Test Cases vs. Debugging Cases

- When debugging something hard, you typically "set up" a particular case, and then step through your code on that case
- After you are done, all that work disappears!
- With this situation and unit tests, instead of setting up a case to debug, set up a unit test that hits that case and debug on that.
- Now, after you fix the bug, you get the benefit of that unit test forever!
- To be fair, setting up a unit test is a bit more work, but the payoff can be nice.

Unit Test Boundary Fun

- Change a key < in the work code to a <= to observe the unit test fail -- it really is bearing down on that case, then change it back to <. In this way you see that the unit test boundary really is where you think it is.
- Change a comment or something else not scary in the code. If you're bored, run the tests again, just to see the green.

Writing JUnit Tests in Eclipse

Eclipse JUnit Tests

- JUnit is a very popular system for unit tests, and it is integrated very well into Eclipse
- Right-click on the class to be tested
- Select New...JUnit Test Case
- For a class named Binky, name the test class BinkyTest
- Select Finish -- creates your BinkyTest class, possibly with some boilerplate in it

junit.jar

- JUnit depends on some classes that are in the file junit.jar
- Following the New JUnit Test Case steps above, Eclipse will ask if you want to add junit.jar to your project if it is missing, so that's the easiest way to get junit.jar in your project.

Writing JUnit Tests

- Write a method, beginning with the word "test". For a "foo" method, you might name it "testFoo()". (JUnit simply looks for methods beginning with lowercase "test", and figures they must be test methods.)
- For example, if the class being tested has a foo() method, then we might write a testFoo() method that calls foo() 5 different ways to see that it returns the right thing.
- Your first basic tests can just hit the obvious cases.
- More advanced tests should push on hard, weird, and edge cases -- things that are the most likely to fail. You **want** the test to fail if possible -- it is giving you very valuable information about your code.
- You can write obnoxious, hard tests that hit at edge cases. If the code can get past these, it can do anything!
- Or put another way, you want take a sort of aggressive posture towards the code in the tests -- pushing on the code in ways that will make its author nervous.
- The tests do not need to be exhaustive -- the space of inputs is so big, exhaustiveness is a false goal. Try to hit a meaningful variety of test cases, and that's good enough and don't worry about it. Unit tests have diminishing returns. Once you have a few good ones, you've got most of the benefit.

Common Unit Testing Mistake

- The most common mistake in writing unit tests is not being hard enough.
- It's easy to write, say, 7 unit tests that all work at about the same "moderate" level. In reality, that's not a good use of time. If the first 3 moderate tests pass, probably the other 4 will too. The last 4 aren't actually adding anything.
- Instead, the best approach is:
  - Write 2 or 3 "moderate" tests, and then you are done with moderate tests -- 3 is enough.
  - Then write 3 hard/mean/obnoxious tests, and ideally each of those should be hard in a different way.

- It's difficult to get in the mindset of writing truly hard tests, but of course only the hard tests really drive the quality up towards being perfect.

JUnit assertXXX Methods

- JUnit has assertXXX() methods that check for the "correct" results
- assertEquals(*desired-result*, *method-call*);
  - Works for primitives, and uses .equals() for objects
- assertTrue(*expr-that-should-be-true*);
- The convention in the checks is that the desired values is first, the computed value is second.
- Use assertTrue(Arrays.equals(a,b)) for arrays, since regular .equals() does not work (Arrays.deepEquals(a,b) for multi-dimensional arrays).

Unit Test Info String

- It can be handy to print out the expected and actual data during a unit-test run, however the convention is that when unit tests go into production, they should print anything.
- As an alternative, the assertXXX() methods take an extra first String argument which will print if the test fails.
- Suppose we have some foo(String a, String b) method that is supposed to return true. We could put extra info to print in the first String argument, and that will print out if the assert fails.
  - assertTrue("call foo:" + a + " : " + b, foo(a,b));
- Also, instead of printing, you can just put a breakpoint in the failing test, and look at the literal data in the debugger.

Running JUnit

- At the left, select the test class, right click, select Run JUnit Test
- Click on the JUnit tab in the left pane to see the results
- Alternately, click on the package on the left. Then running JUnit Tests will run all of the tests.

Working With JUnit Failures

- On failure, double click a line in the "failures" list to go to the test method that failed.
- The stack trace at the lower left shows the call sequence. If there was an assert failure, it shows the expected and actual values. (very handy!) Double click it to get details of the expected and actual values.
- Double clicking in the stack trace goes to that line.
- The data and variables are not live -- the JUnit report is of what happened in the past
- To see the data live, put a breakpoint in the work code, and select Debug... to run the unit test in the debugger where you can look at the values as it runs.
- Key idea: In the list of tests in the JUnit pane in Eclipse, you can right-click one test to Run or Debug **just that test**. In this way, you can break in the work code for just the test that is breaking.

Unit Test Copy/Paste

- Very often, unit tests have a natural setup/call/test structure.
- It can be handy to copy one testFoo() method and paste it in to make the next testFoo2() method -- that's an ok practice. Unit tests really are the same idea, repeated 5 times, so this technique makes more sense than it would for production code.
- If you have complex data setup, you may want to factor out some private data-building utilities called from the test methods.

Unit Test Object/setUp()

- Sometimes, unit tests require a lot of data to set up. Rather than repeating that for each test method, the unit test class can have regular instance variables that store data used by all the test methods. The special method "void setUp()" can set up the data structures for use by the test methods. JUnit makes a new instance of your test object and calls setUp() before each test method, so each test starts with a fresh context. The unit test system tries to make the test runs independent and repeatable.. the results should not depend on which tests run earlier or later.

Emails Example

```
import java.util.*;
/*
 * Emails Class -- unit testing example.
 * Encapsulates some text with email addresses in it.
 * getUsers() returns a list of the usernames from the text.
 */
public class Emails {
    private String text;

    // Sets up a new Emails with the given text
    public Emails(String text) {
        this.text = text;
    }

    // Returns a list of the usernames found in the text.
    // We'll say that a username is one or more letters, digits,
    // or dots to the left of a @.
    public List<String> getUsers() {
        int pos = 0;
        List<String> users = new ArrayList<String>();

        while (true) {
            int at = text.indexOf('@', pos);
            if (at == -1) break;

            // Look backwards from at
            int back = at - 1;
            while (back >= 0 &&
                    (Character.isLetterOrDigit(text.charAt(back)) ||
                     text.charAt(back)=='.')) {
                back--;
            }

            // Now back is before start of username
            String user = text.substring(back + 1, at);

            if (user.length() > 0) users.add(user);

            // Advance pos for next time
            pos = at + 1;
        }

        return users;
    }
}
```

EmailsTest

```
import junit.framework.TestCase;
import java.util.*;
/*
  EmailsTest -- unit tests for the Emails class.
 */
public class EmailsTest extends TestCase {

    // Basic use
    public void testUsersBasic() {
        Emails emails = new Emails("foo bart@cs.edu xyz marge@ms.com baz");
        assertEquals(Arrays.asList("bart", "marge"), emails.getUsers());
        // Note: Arrays.asList(...) is a handy way to make list literal.
        // Also note that .equals() works for collections, so the above works.
    }

    // Weird chars
    public void testUsersChars() {
        Emails emails = new Emails("fo f.ast@cs.edu bar a.2.c@ms.com ");
        assertEquals(Arrays.asList("f.ast", "a.2.c"), emails.getUsers());
    }

    // Hard cases -- push on unusual, edge cases
    public void testUsersHard() {
        Emails emails = new Emails("x y@cs 3@ @z@");
        assertEquals(Arrays.asList("y", "3", "z"), emails.getUsers());

        // No emails
        emails = new Emails("no emails here!");
        assertEquals(Collections.emptyList(), emails.getUsers());

        // All @, all the time!
        emails = new Emails("@@@");
        assertEquals(Collections.emptyList(), emails.getUsers());

        // Empty string
        emails = new Emails("");
        assertEquals(Collections.emptyList(), emails.getUsers());
    }
}
```