Eclipse and Unit Testing

Eclipse

- Open source project out of IBM, www.eclipse.org
- Many, many features -- somewhat bewildering
- See the compiler notes and links on the course page for full Eclipse information. This handout is just a quick introduction to get things started.

Import Existing Project (HW)

- The homework starter files will be distributed as set-up Eclipse projects, ready to import.
- An Eclipse project is equal to a directory -- it contains the source files and also hidden .project files that contain the Eclipse project info. Therefore, you want to copy/move your whole project directory when moving it from one place to another.
- Suppose you have a project directory in the filesystem somewhere
  - Contains all the .java files, and Eclipse will put the .class files there too
  - Also contains .project and .classpath files that contain the Eclipse configuration
- Select "File > Import"
- Select "Existing project"
- Browse to project directory and select. Eclipse will write its files, changes etc. to that directory (i.e. that's the directory to take with you, not lose, etc.)

Running

- At the left, select the class containing main()
- Right-click (on the Mac ctrl-click), and select Run.. Java Application
- Or, select Run... to manage the "configurations" to run
- The green triangle "run again" button is very handy -- run whatever you ran before again. It also has a menu of your most recent runs (different mains(), unit tests...).

Debugging

- Double click in the editor at the left of a line to set a breakpoint on that line
- Use "Debug" instead of "Run" to run in the debugger (Note: "run" is different from "debug" -- run ignores the breakpoints).
- This should prompt to switch to the "debugger" perspective
  - There are little buttons at the far upper right to switch perspectives
  - Also the "Open Perspective" menu
- In the debugger, look at the stack and variables
- Use the red-square to kill the program.
- Click the "J-Java" in the upper right to return to the editing perspective (after killing the program typically)

Closing / Quitting

- Right-click the project and select "Delete". This should prompt with a "do not delete contents", which is what we want ("contents" in this case refers to the .java and other files out in the directory). This removes the project from Eclipse, but leaves its directory, files, etc. intact out in the filesystem.
- This detaches the project from Eclipse, and you can quit.

Other Great Eclipse Features

- Eclipse has **many** features, but here are a few of my favorites
- Type "foo." -- it pops up completions. Hit return to select, escape to get out. Also, typing ctrl-space brings this up on command.
- Completion also works with generic syntax in spots -- a spot where you can see from context that a <String> or whatever is called for -- ctrl-space and Eclipse just puts it in.
- Hover over a word in the source to pull up its javadoc.
- Select some code, use Source > Correct Indentation to fix indentation
- Use Source > Shift Left/Right (tab, shift-tab also work) to bulk change the indent of something
- Select a method/variable/class name and use the Refactor > Rename command -- this is a superficial feature, but is sure is handy to fix up a name choice quickly.
- Select a block of code inside a method, and use Refactor > Extract Method. It's smart about what parameters are required.
- Use the Source > Toggle Comment command to block comment/uncomment code.

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.
- 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.
- Solution: codify tests for every component as it is built. Make it easy to run all the tests. Allow us to keep quality high, and makes the change process less scary.

High Quality Code

- We think about building lots of different types of code
- Normally, code gets built to a "it seems to work" quality level
- With unit tests, we have the possibility to building code to a much higher quality level
- 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 -- Workflow

- With the "test first" style, we 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).
- Test code does not need to be **exhaustive** -- test code adds a lot of value even just hitting fairly obvious cases.

Unit Testing Advantages

- Can yield code of enormously higher quality compared to code that is just built with programmer ad-hoc testing.
- 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.
- Go the extra step of recording the test cases -- they will pay dividends for the whole lifetime of the code, future features, etc.
- 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.

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.

Unit Test Tax

- Writing tests certainly adds 20% more work -- it's just more code to do.
- However it certainly increases quality, reduce time that goes into debugging.
- Debugging can be such a time sink ... the 20% 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. It's not about being comprehensive.

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.

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 and write some code, 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 is 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 great.

Unit Test Fun

- Change a comment -- run the tests again, just to see the green
- 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 <.

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 just 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 can take a sort of aggressive posture towards the code in the tests -- really pushing on the code in hard ways.
- 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.

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 that's done
  - 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 mode 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*);
- Use assertTrue(Arrays.equals(a,b)) for arrays, since regular .equals() does not work.

Printing

- It can be nice 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 don't print anything (you should comment your printlns out).
- 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 is not live -- the JUnit report is of what happened in the past

- To see the data live, put a breakpoint on the line in question, and select Debug... to run the unit test in the debugger where you can look at the values as it runs.

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 similar, 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 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.

Emails Example

```
import java.util.*;
/*
 * Emails Class -- unit testing example.
 * Encapsulates some text with email addresses in it.
 * getUsers() extracts 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.
    // A username is made of zero or more letters, digits, dots, underscores
    // to the left of the @.
    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 until finding a non-username char
            // or hitting the start of the string
            int back = at - 1;
            while (back >= 0 &&
                    (Character.isLetterOrDigit(text.charAt(back)) ||
                     text.charAt(back)=='.' || text.charAt(back)=='_' )) {
                back--;
            }

            // now back is before start of username
            users.add(text.substring(back + 1, at));

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

        return users;
    }
```

```
    // Little test main() that just prints from the input
    // on the command line.
    public static void main(String[] args) {
        System.out.println("Running Emails...");
        for (int i=0; i<args.length; i++) {
            Emails emails = new Emails(args[i]);
            System.out.println(emails.getUsers());
        }
    }

}
```

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_b.c@ms.com ");
        assertEquals(Arrays.asList("f.ast", "a_b.c"), emails.getUsers());
    }

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

        // no emails
        emails = new Emails("no emails here!");
        assertEquals(Arrays.asList(), emails.getUsers());

        // lone @
        emails = new Emails("@");
        assertEquals(Arrays.asList(""), emails.getUsers());

        // empty string
        emails = new Emails("");
        assertEquals(Arrays.asList(), emails.getUsers());
    }
}
```

Deltas Example

```
/*
 Deltas Class
 Deltas encapsulates a moderately complex computation as fodder for unit-testing.
 Does not have any data -- just a static deltas(int[]) method
 with this definition:
 -sums up the deltas between adjacent number pairs, so {1, 3, -1} -> 6
 -delta pairs where the first number is in the first 20% of the array
 count double, so {1, 2, 3, 4, 5} -> 5
 -deltas pairs where the first number is 0 do not count, so {1, 0, 10} -> 1
 */
```

```
public class Deltas {
    // Returns the deltas value of the given array.
    public static int deltas(int[] values) {
        int sum = 0;
        int doubleLength = values.length / 5; // part of array that counts double

        for (int i=0; i<values.length-1; i++) {
            int here = values[i];
            int next = values[i+1];

            // skip if 0, double if within doubleLength
            if (here != 0) {
                int delta = Math.abs(here-next);
                if (i<doubleLength) delta *= 2;
                sum += delta;
            }
        }
        return sum;
    }
}
```

Deltas Test

```
import junit.framework.TestCase;
import java.util.*;

/*
  DeltasTest -- unit tests for the Deltas class.

  There are three aspects:
  -basic summing of deltas
  -doubling the first 20% of the numbers
  -not counting deltas where the fist num is 0

  The basic tests work on a single problem aspect -- then harder tests
  try them in combination.

  In this case, Deltas includes only one method. If there were more methods,
  we would test them here too.
 */
public class DeltasTest extends TestCase {




    // basic -- no doubling, no zeros
    public void testDetalsBasic() {
        // len 4 -- no doubles
        assertEquals(4, Deltas.deltas(new int[] {1, 3, 2, 1} ));
        assertEquals(11, Deltas.deltas(new int[] {1, 5, -2, -2} ));
    }

    public void testDeltasZeros() {
        // try some zeros (also shows using first info String arg)
        int[] a = {1, 0, 0, 10} ;
        assertEquals("zeroes should not count:" + Arrays.toString(a),
                1, Deltas.deltas(a));

        int[] b = {0, 10, 1, 0} ;
        assertEquals("zeroes should not count:" + Arrays.toString(b),
                10, Deltas.deltas(b));
    }

    // bump the len up to 5,9,10 to check the doubling -- note the choices
    // of lengths 5,9,10 deliberatly push on the "boundary" case of the doubling.
    public void testDeltasDoubling() {
         // len 5 -- 1 double
```

```
        assertEquals(5, Deltas.deltas(new int[] {1, 2, 3, 4, 5} ));

         // len 9 -- 1 double
        assertEquals(6, Deltas.deltas(new int[] {1, 3, 1, 1, 1, 1, 1, 1, 1} ));

        // len 10 -- 2 doubles
        assertEquals(8, Deltas.deltas(new int[] {1, 3, 1, 1, 1, 1, 1, 1, 1, 1} ));
    }

    // hard cases -- combine doubling with zeros, and weirder looking cases
    public void testDeltasHard() {
         // len 5 1 double + 1 zero
        assertEquals(7, Deltas.deltas(new int[] {1, 3, 0, 1, 1, } ));

         // len 7 1 double + some zeros
        assertEquals(14, Deltas.deltas(new int[] {1, 3, 0, 1, 2, 0, 5, 1 } ));

         // 1 double + 1 zero cancels it
        assertEquals(2, Deltas.deltas(new int[] {0, 3, 1, 1, 1} ));

         // simple but weird edge cases
        assertEquals(0, Deltas.deltas(new int[] {} ));
        assertEquals(0, Deltas.deltas(new int[] {0} ));
        assertEquals(0, Deltas.deltas(new int[] {0,0,0} ));
        assertEquals(0, Deltas.deltas(new int[] {-1,-1,-1,-1,-1} ));
    }
}
```