CS 401 – Spring 2014

JUnit Laboratory

Objectives

The objective of this lab is to learn how to test code with the unit testing framework JUnit.

Background

The project contains two classes – Student and DegreeAudit.

The Student class represents a student for the purposes of doing a degree audit.

The DegreeAudit class is a driver class that is used to test some aspects of the Student class. It does this by creating some student objects, setting their attributes, and printing the results of calling some of their methods.

We have been using this method of testing our code throughout the semester. It relies on a human to read the output and to compare it to the expected output to determine if the code is working correctly. The problem with this testing method is that the human has to carefully check the output, and needs to do so every time the code is changed, compiled and run. What we would like to have is a more automated way of testing our code.

Unit testing provides a way of automatically testing individual classes and methods in an automated and repeatable way. By writing tests that test the behavior of classes and methods and comparing them automatically to expected values, we can let the computer run through our suite of tests every time we make a change to the code. This way we can be sure that any changes we made did not break some code that was already working. In many large software projects the test suite is run automatically every time a developer pushes new code to repository.

We will be using JUnit, a unit testing framework written specifically for Java. JUnit can be run on its own, or as a plug-in to integrated development environments like Eclipse. JUnit is already built in to BlueJ.

Procedure

1. Fork the JUnit-Lab Repository

1. Go to the junit-lab repository at https://github.com/kwurst/junit-lab
2. Fork the JUnit-Lab repository to your account.
3. Add your instructor and partner(s) as collaborators.

2. Download (clone) the JUnit-Lab Repository

1. In Eclipse, choose File:Import
2. Choose Git:Projects from Git
3. Choose Clone URI
4. Paste the SSH clone URL from GitHub
5. Set the protocol to SSH

6. Select the master branch
7. Choose the directory you want to clone into.
8. Select Use the New Project Wizard
9. Choose Java:Java Project
10. Enter a project name

3. Read and Understand the JUnit-Lab Code

1. Read both the code, and the documentation for both classes, and be sure you understand what the code is supposed to do, and how it works.
2. Run the main method to be sure you understand what it is doing, and what the code is testing.

If you do not understand any part of the code, ask the instructor. In the next step you will be writing tests to be sure that the code is correct, and that continues to work correctly as we modify it.

4. Create a StudentTest Class

1. Right-click on the Project, and choose *New:Source Folder*.
2. Name the new folder *test*.
3. Right-click on Student class, and choose *New:JUnit Test Case*.
4. Set the source folder to be the test folder you just created.
5. Check the box to create a setUp() method.
6. Check the box to create a test method for getCurrentEarnedCr().
7. Allow Eclipse to add JUnit 4 to the build path.
8. The StudentTest class will open in the editor.

5. Write a Test Method

Our first test will be a simple one to determine if our code for storing the Student's Current Earned Credits works correctly. Modify the testGetCurrentEarnedCr() method to look like:

```
@Test
public void testCurrentEarnedCr() {
    Student student1 = new Student("Jane", "Smith");
    int credits = 45;
    student1.setCurrentEarnedCr(credits);
    assertEquals(credits, student1.getCurrentEarnedCr());
}
```

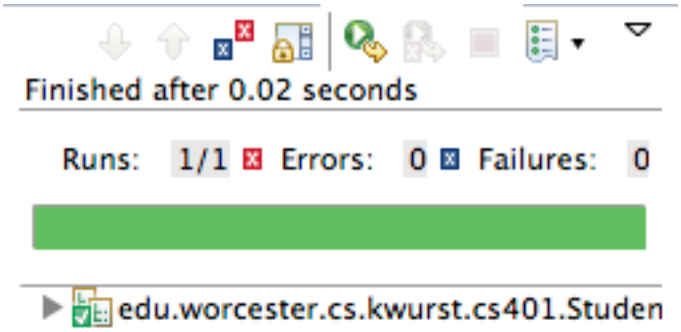
- The first line (`@Test` annotation) tells JUnit that this will be a test method.
- The second line is simply the header for the test method. The name of the method does not have to have the word "test" in it, but it is standard to do so.[1]
- The third line creates the Student object needed for the test.

[1] In JUnit 3, including "test" in the name of the method was required to make a method a test method. JUnit 4 added the @Test annotation to do this, so it is no longer necessary to include the word "test" in the method name.

- The fourth and fifth lines set the Student's Current Earned Credits to some value. This is being done on two lines, and with a variable for the credits value, so that we can use the value of the variable on the next line.
- The sixth line compares the value of the credits variable that we used with the value returned by the getCurrentEarnedCr() method to see if they are equal. The assertEquals method is part of the JUnit framework.

6. Run the Test Method

1. Save your code.
2. Right-click on StudentTest.java and choose Run as:JUnit Test.
3. A new window should appear that looks like:



4. The green bar indicates that the test was successful.

7. Modify the Test Method to Make It Fail

Just to see what a failing test looks like, lets modify our test class so that the it does not pass:

1. Change the last line of the testCurrentEarnedCr() method, so that it looks like this:

```
assertEquals(40, student1.getCurrentEarnedCr());
```

2. Compile your code and run the test again. You should get a red bar this time, indicating that the test has failed.

3. The failing test will have an **X** on its icon t in the upper part of the window. This is how you will know which test has failed when you have multiple tests in your test suite.
4. Select the test line with the **X** on it, and an explanation of why the test failed will be displayed in the lower portion of the window. You can even double-click on the line below that includes the name of the test method to take you directly to the method's code.
5. Go back and return your test method to it's working state, and run your test again.

8. Write and Run a Test Method for Anticipated Additional Credits

Follow the steps in parts 6 and 7 to create and run test method for Anticipated Additional Credits. *You do not need to modify the test to make it fail.*

Notice that you all of your tests are run every time, and notice how easy to see that all of the tests passed – you do not need to scan through all of the output and compare values yourself.

9. Use the setUp() Method

Notice that for each of your tests, you needed to create a student object as the first step. Since this needs to be done for every test method, it would be nice to have this done automatically before each test. This is what the setUp() method is for. The `@Before` annotation tells JUnit to execute this method before every `@Test` method.[2]

1. Declare an instance variable for a Student object at the top of the class:

```
private Student student1;
```

2. Move the line that creates the Student object to the setUp() method:

```
@Before
public void setUp()
{
    student1 = new Student("Jane", "Smith");
}
```

3. Delete the line that creates the Student from each of your test methods.
4. Run your tests again.

[2] In JUnit 3, the method was required to be "setUp" to make a this method be called before every test method. JUnit 4 added the @Before annotation to do this, so it is no longer necessary to call the method "setUp". You can even have multiple @Before methods, but there is no guarantee that they will run in any particular order.

10. Test Method for readyToGraduate()

Let's write at test for a more difficult to test method – readyToGraduate(). For a student to be ready to graduate they must meet 4 criteria[3]:

1. Completed 120 credits.
2. GPA of 2.0 or higher.
3. Completed LASC requirements.
4. Complete major requirements.

So, to test the readyToGraduate() method, we need to set 4 different values first – and we are testing a Boolean return value.

1. Create at new test method called testReadyToGraduate1(). We will have more than one test for this method, so we are adding a number at the end of the method name:

```
@Test
  public void testReadyToGraduate1()
  {
      student1.setCurrentEarnedCr(120);
      student1.setGpa(2.0);
      student1.setMajorComplete(true);
      student1.setLascComplete(true);
      assertTrue(student1.readyToGraduate());
  }
```

2. For this method we are testing for a student who is ready to graduate, so we are setting all 4 criteria to make that true.
3. Notice that on the last line, we are using a different assert method – assertTrue. We could have used assertEquals and just set the first parameter to true, but since this is a common thing to test, JUnit provides an assertTrue and an assertFalse method.
4. Run your tests again.

11. Tests for ***Not*** readyToGraduate()

Write 4 more testReadyToGraduate() methods that will to test that the student is ***not*** ready to graduate. Have the Student in each of these tests *not* meet *one* of the 4 criteria.

1. Don't forget to give each test methods a different name.
2. Don't forget to change the last line in each to use the assertFalse method.[4]
3. Run your tests again.

[3] Four criteria for our purposes for this lab. In reality, there are other criteria to be met.
[4] These are still passing tests. We are just testing for a negative condition – that readyToGraduate() returns false.

12. One More Test for readyToGraduate()

Let's make one more test for readyToGraduate() that should pass. Let's test the case where the student has met all 4 criteria, but has met the Current Earned Credits criteria by completing *more* than 120 credits:

1. Copy the testReadyToGraduate1() method.
2. Edit the first line of the body so that the student has completed more than 120 credits.
3. Run your tests again.
4. The test failed! What happened? Go find the error.

13. Fix and Test getCurrentRemainingCr()

Looks like we should have written some tests for getCurrentRemainingCr() – if we had, we would have noticed the bug.

1. Write a test for getCurrentRemainingCr() where the student has taken less than 120 credits.
2. Write at test for getCurrentRemainingCr() where the student has taken exactly 120 credits.
3. Write a test for getCurrentRemainingCR() where the student has taken more than 120 credits.
4. Run your tests.
5. The first two getCurrentRemainingCr() tests should pass, and the third should fail.
6. Add an if statement to getCurrentRemainingCr() so that it passes the test.
7. Run your tests again. All of your tests, including the readyToGraduate() tests, should pass.

14. Using Your Test Suite to Validate Code Changes

Now we will see the value of having a suite of tests written that you can easily run. You know that all of the code that you have tested correctly. You can feel comfortable making changes to the code in those methods, and you won't lose the functionality that is already there, because you can just run your tests again to be sure they still work as they did before.[5]

We've come up with a better[6] way to implement getCurrentRemainingCr(). There is a statement in Java called the ternary operator. It is a type of a replacement for an if-else. If you had code that looks like this:

```
if (test)
    return value1;
else
    return value2;
```

[5] You need to realize that you have not written tests for all of your methods yet, nor have you tested every possible set of inputs for those methods. There is still the possibility that other bugs will crop up in that untested code, or for untested cases. But you can be sure that you have not broken any of the previously tested code.

[6] Better is arguable for this case. This new implementation will be shorter, but it may be less clear. Your mileage may vary…

You could replace it with this:

```
return test?value1:value2;
```

So, let's update our code:

1. Replace the if-else statement in getCurrentRemainingCr() with:

```
return (currentEarnedCr >= 120)?(REQUIRED_CR – currentEarnedCr):0;
```

2. Run your tests.
3. What?! We had some failing tests?! I guess I forgot how to use the ternary operator.
4. Swap the two values in the ternary operator – the zero should be before the question mark, and the subtraction should be after the question mark.
5. Run your tests.
6. Now everything works correctly still. I am assured that the change I made did not break any previously working code.

15. Test getStudentCount()

You can probably write tests at this point for all of the rest of the functions of the Student class except for the automatically increasing student id number and the getStudentCount() method. These require a second Student object to be created.

1. A good first attempt at a test for the Student ID might look like this:

```
@Test
public void testStudentCount() {
    new Student("Joe", "Jones");
    assertEquals(2,Student.getStudentCount());
}
```

This should work because Joe is the second student we created, after Jane – right?

Run your tests….

Wrong!

We have been creating a new Student object for each test, so our student count is higher than 1 when we start this test.

So, we need to get the student count before we create Joe, save that in a variable, and then check that the count has gone up by one after we create Joe:

```
@Test
public void testStudentCount() {
    int count = Student.getStudentCount();
    new Student("Joe", "Jones");
    assertEquals(count+1,Student.getStudentCount());
}
```

16. Test Student ID

You are going to have the same type of problem, for the same reason, when you try to test that the Student ID increases every time you create a new student.

Fortunately, the same solution applies:

```
@Test
public void testStudentID() {
    int count = Student.getStudentCount();
    Student s2 = new Student("Joe", "Jones");
    assertEquals(String.format("%07d", count+1),s2.getId());
}
```

17. Commit Your Working Program

18. Push Your Program to GitHub and Clone Onto Your Partner's Computer