Lab: Getting Started With JUnit

Objective

In this lab you will learn how to create and run a JUnit test fixture in Eclipse. You will also practice how to interpret the results of a JUnit test fixture run and how to use the results to track down and fix issues in the code under test and in the test code itself.

Setup

Follow these steps to set up a project for this lab.

1. Create a new Eclipse project by copying `ProjectTemplate`. Name the new project `JUnitGettingStarted`.
2. Open the `src` folder of this project and then open `(default package)`. As a starting point you can use any of the Java files. Rename it `FactoringUtility` and delete the other files from the project.
3. Follow the link to `FactoringUtility.java`, select all the code on that page and copy it to the clipboard; then open the `FactoringUtility.java` file in Eclipse and paste the code to replace the file contents. Save the file.

Method

Take a look at the `FactoringUtility` class. There are 4 methods in `FactoringUtility`: one implementation of `aFactor`

```
/**
 * Reports some factor of a number.
 *
 * @param n
 *            the given number
 * @return a factor of the given number
 * @requires n > 0
 * @ensures aFactor > 0 and n mod aFactor = 0
 */
public static int aFactor(int n) {...}
```

and 3 different versions of `aNonTrivialFactor` with the same contract but different implementations

```
/**
 * Reports some non-trivial factor of a composite number.
 *
 * @param n
 *            the given number
 * @return a non-trivial factor of the given number
 * @requires n > 2 and [n is not a prime number]
 * @ensures 1 < aNonTrivialFactor < n and n mod aNonTrivialFactor = 0
 */
public static int aNonTrivialFactor(int n) {...}
```

Review these contracts and make sure you understand how the two contracts are different and what the requires and ensures say about the preconditions and postconditions for these methods.

Creating a JUnit Test Fixture

To create a new JUnit test fixture, which is basically a normal Java class with test methods identified by the `@Test` annotation, follow these steps.

1. Right-click on the `test` folder in the **Package Explorer** and select **New > JUnit Test Case**. In the **New JUnit Test Case** window do the following:
    1. make sure that **New JUnit 4 test** is selected
    2. make sure that the **Source folder** input box contains `JUnitGettingStarted/test` (if that is not the case, you should update the contents to make it so); this specifies the location in your project where the JUnit test fixture will be located
    3. enter `FactoringUtilityTest` in the **Name** input box
    4. enter `FactoringUtility` in the **Class under test** input box
    5. click **Finish**.

Eclipse opens the newly created test fixture. Take a look at it and observe the import statements (needed by JUnit) and the one sample test case (which is set up to always fail).

Now replace the only test case with the following:

```java
@Test
public void testAFactor1() {
    /*
     * Set up variables and call method under test
     */
    int n = 1;
    int factor = FactoringUtility.aFactor(n);
    /*
     * Assert that values of variables match expectations
     */
    assertEquals(1, factor);
}
```

As you can see, the `testAFactor1` test case simply invokes the `FactoringUtility.aFactor` static method under test with an argument equal to 1. After the call, it checks that the value returned is indeed 1, i.e., the only positive factor of 1 and the only possible correct return value for the method. (Note that because the `aFactor` *static* method is being invoked in a different class from the one where it is declared, we need to use the name of the method *qualified* with the name of the class where it is declared.)

Running a JUnit Test Fixture

Let's run the test fixture to execute our one test case. Running a JUnit test fixture in Eclipse is very similar to running a Java program and there are several ways to do it. A simple way is to right-click on the test fixture (`FactoringUtilityTest.java`) in the **Package Explorer** view and select **Run As > JUnit Test** in the contextual pop-up menu.

When you run a JUnit test fixture for the first time, something interesting happens. A new view opens in Eclipse, the **JUnit** view. This view shows the results of running the test fixture. In this case, assuming everything went as expected, you should see a green bar (indicating that all test cases passed) and above the bar, three entries: **Runs: 1/1** (indicating that one test case out of one available was run), **Errors: 0** (indicating that there were no errors), and **Failures: 0** (indicating that there were no failures). Below the green bar, you can expand a list of the test cases run (only one in this case) showing whether each test case passed (check-mark in green box) and how long it took to run the test case.

Interpreting the Results

To see what the possible outcomes of running a JUnit test fixture are, let's copy some more test cases.

1. Follow the link to `FactoringUtilityTest.java`, select all the code on that page and copy it to the clipboard; then go back to `FactoringUtilityTest.java` in the Eclipse editor and paste the code to replace the entire file contents. Save the file.
2. Note that the test cases you just copied are not meant to be a systematically developed test plan for the methods under test. Rather they are designed to show the different behaviors you might observe when running JUnit test fixtures.
3. Run the fixture. Be patient, as some test cases may take a little longer than expected (so you should try to explain why that happens while analyzing the results later in the lab).

Observe the results. You should see a red bar (instead of the green one you encountered before) and above it the following: **Runs: 12/12**, **Errors: 2**, and **Failures: 4**.

To interpret the result we first need to consider what could happen when running a test case. These are the possible behaviors:

1. The test case completes without run-time errors and all assertions in the test case are satisfied. The test case passes, i.e., it does not show a defect (bug) in the code under test.
2. The test case terminates with a run-time error (e.g., dividing by 0 or accessing an element of an array outside the range of valid indices) but *not with a failed assertion*. JUnit labels this an *error*.
3. The test case terminates with a failed assertion (either one of JUnit's assertions or one checking a method's precondition). JUnit labels this a *failure*.
4. The test case does not terminate. This is most likely due to the presence of an infinite loop in the test case. You will have to stop JUnit by clicking the red square button in the **JUnit** view toolbar.

It is important to observe the following:

- The outcome of a JUnit test case is the result of the execution of the *entire* test case and often it may not be clear how the observed behavior is affected by the code under test (i.e., the implementation of the method being tested) as opposed to the code of the test case itself. In particular, in the case of error, failure, or non termination, the problem can be either in the method being tested or in the test case or in both.
- It is possible that even when the test case completes successfully, i.e., without revealing a bug, a defect in the test code (e.g., a wrong assertion) may be hiding an actual problem in the method being tested.
- If the test case makes a call to the method under test with a violated precondition, JUnit does not warn the tester that the test case is flawed and useless (because no matter what the method under test does, the test case cannot show the presence of a bug). The behavior here is affected by whether the violated precondition is checked by the method under test or not. If the method under test checks the precondition (and does so correctly), JUnit displays it as a failure and the error message in Eclipse's **JUnit** view shows the violated assertion error message. If the method under test does not check the violated precondition, any of the above four possible outcomes can occur. **Remember: test cases that violate the requires clause of the method under test are worthless and you should never include them in your test fixtures.** (See slides 46-47 in Testing.)

Your Turn

First note the distinction between testing and debugging. The goal of *testing* is to show that some code has a defect, while the goal of *debugging* is to find the defect and repair it. (See slide 29 in Testing.) For the rest of this lab, you will be debugging the code under test and sometimes the code of the test cases.

It is now your turn to analyze, classify, and fix each test case. For each of the test cases provided, fill out the corresponding row in the table below with the following information (if you want to be able to take a record of your results, you may want to fill out a similar table on paper instead of on this page):

1. **Outcome**: the outcome of running the test case, one of *pass*, *error*, or *failure*;
2. **Problem Source**: if there is any defect exposed (or hidden) by the test case, specify where the problem is, e.g., *method under test* or *test case code* and what kind of problem you discovered;
3. **Resolution**: fix the problem in the method under test or in the test fixture and briefly describe here how you resolved it.

After you apply each fix (either to the code of the method under test or to the code of a test case), rerun the test fixture to make sure the issue was indeed fixed and that you have not introduced new bugs. If you have, you need to fix those as well.

Test Case	Outcome	Problem Source	Resolution
`aFactorTest1`	`failure`	`test case code line 22 and 29`	`22) int n=1; 29) assertEquals(1, n);`
`aFactorTest2`	`pass`		
`aFactorTest3`	`pass`		
`aFactorTest4`	`pass`		
`aFactorTest5`	`pass`		
`aNonTrivialFactorV1Test1`	`pass`		
`aNonTrivialFactorV1Test2`	`failure`	`test case code line 132`	`132) assertTrue(2 < n);`
`aNonTrivialFactorV1Test3`	`failure`	`method under test line 47`	`47) factor = factor + 1;`
`aNonTrivialFactorV2Test1`	`pass`		
`aNonTrivialFactorV2Test2`	`failure`	`method under test line 70` `test case code line 205`	`70 ) made into an else statement` `205) assertEquals(0, n % factor);`
`aNonTrivialFactorV2Test3`	`error`	`method under test line 70` `test case code line 205`	`70 ) made into an else statement` `205) assertEquals(0, n % factor);`
`aNonTrivialFactorV3Test1`	`error`	`method under test line 87`	`int factor = 4;`

Note a couple of very useful features of the **JUnit** view in Eclipse.

- If you select (by clicking on it) one of the test cases in the **JUnit** view, in the **Failure Trace** panel below the list of test cases, Eclipse displays useful information about what went wrong. Note the difference between the information provided when an `assertEquals` fails (e.g., `aNonTrivialFactorV2Test2`) vs. when an `assertTrue` fails (e.g., `aNonTrivialFactorV1Test2`).
- If you double-click on one of the test cases in the **JUnit** view, Eclipse takes you in the editor to the statement in the test case where the issue occurred. This is invaluable in determining where the problem originated.

Once you are done with this part of the lab, you should have fixed any bugs both in the methods in `FactoringUtility` and in the test cases in `FactoringUtilityTest` and all the JUnit test cases should pass. (Note that this goal may be achieved by all three implementations of `aNonTrivialFactor` being the same correct implementation and some test cases having to be removed, e.g., because they are worthless...)

Additional Activities

1. Come up with at least 3 significantly different new test cases for one of your corrected implementations of `aNonTrivialFactor`.
2. Add them to the `FactoringUtilityTest` test fixture.
3. Run the updated test fixture and fix any new problems you discover.