Unit Testing in Jazz Using JUnit

Laurie Williams, Dright Ho, Ben Smith and Sarah Heckman [Contact Authors]
CSC 326 - Software Engineering
Department of Computer Science
North Carolina State University

Back to Software Engineering Tutorials

0.0 Contents

1.0 Introduction to JUnit

JUnit is an open source unit testing framework for automatically unit testing Java programs. JUnit provides functionality for writing and running unit test cases for a project under development, and is helpful when doing Test-Driven Development (TDD). This tutorial describes how to use JUnit within the Jazz environment, but most of the information except for running the JUnit test cases is generic to any JUnit set up. See http://www.junit.org/ for directions on how to run JUnit test cases outside of the Jazz environment.

Jazz is built on the Eclipse platform and comes with JUnit built into the Workbench. You can quickly create test case and test suite classes to write your test code in. With Jazz, Test-Driven Development (TDD), becomes very easy to organize and implement.

Typically, you create the application class that you want to test first. Then a tester can create the associated test case which provides a one-to-one correspondence between application classes and test classes. However, a tester can create test classes individually and later tie them to application classes. By using the built in functionality to create JUnit classes, the test cases are created with the needed imports and extensions for JUnit to run. Once the test case class is built the actual test cases are then coded in by the tester.

JUnit test classes can be rolled up to run in a specific order by creating a Test Suite. When a tester creates a test suite, Jazz will name it for you and will specify all of the test cases in the scope of the project that it can find. The code to run the test suite and to add all of the specified test cases you've created, is added to the test suite for you.

There are two versions of JUnit currently available. JUnit 3.8 uses specific naming conventions for identifying test classes, methods, and suites. These naming conventions are described below:

- **Test Case Class**: Named [`classname`]`Test.java`, where `classname` is the name of the class that is being tested.
- **Test Case Method**: Named `test`[`methodname`], where `methodname` is the name of the method that is tested.
- **Test Suite**: Default name for Jazz/Eclipse is `AllTests.java`

JUnit 4.0 uses annotations (which are keywords that start with @) to identify test classes, methods, and suites. The key annotations are described below.

- **Test Case Class**: Named [`classname`]`Test.java`, where `classname` is the name of the class that is being tested.
- **Test Case Method**: Annotate the method with `@Test`. The method name does not have to start with test.
- **Test Suite**: Only write a `suite()` method if you want to run your JUnit 4 tests with the JUnit 3.8 test runner.
- **Test Set Up**: Annotate the method with `@Before`
- **Test Tear Down**: Annotate the method with `@After`

1.1 Import CoffeeMaker into Jazz

We will be using the CoffeeMaker example through out this tutorial to highlight the main features of JUnit. CoffeeMaker is a small application that simulates the functionality of a coffee maker. The requirements for CoffeeMaker may be found here.

Download the CoffeeMaker example from here. See Eclipse Import/Export for instructions on how to import CoffeeMaker into your Jazz workspace.

1.2 CoffeeMaker File Structure

It is considered a best practice in testing, to separate the test case code from the application code. Typically the application code is in a source folder called *src/* while the test code is in a source folder called *unittests/* or *tests/*. Below is the file structure for the CoffeeMaker example that is used for as the exercise in this tutorial. You can emulate this file structure in your other Eclipse projects.

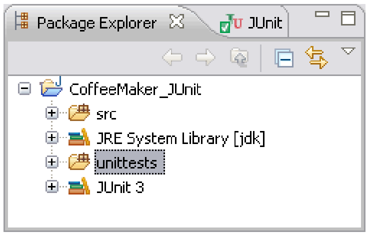


Figure 1: Best practice file structure

Most projects that you create will have a *src/* folder that contain the main application code, a *bin/* (which is not shown in the Package Explorer view) folder that contains the compiled *.class* files, and a *lib/* folder that contains jar files that need to be on the project's build path. To run JUnit test cases the JUnit libraries must be on the classpath. JUnit 3.8 and JUnit 4 are integrated with Jazz; therefore, we can add the Jazz provided libraries to the project.

1.3 Putting JUnit libraries on the Project Classpath

The JUnit library is required to be on the project classpath to compile and run the JUnit test cases. The classpath may be specific to the computer that you are on, so check to make sure that the JUnit library is set up correctly. If your JUnit library isn't set up correctly, there will be an warning message at the top of the Java Build Path window.

1.1.1 Right click on the project and select **Properties**. In the left column select **Java Build Path** and select the **Libraries** tab.

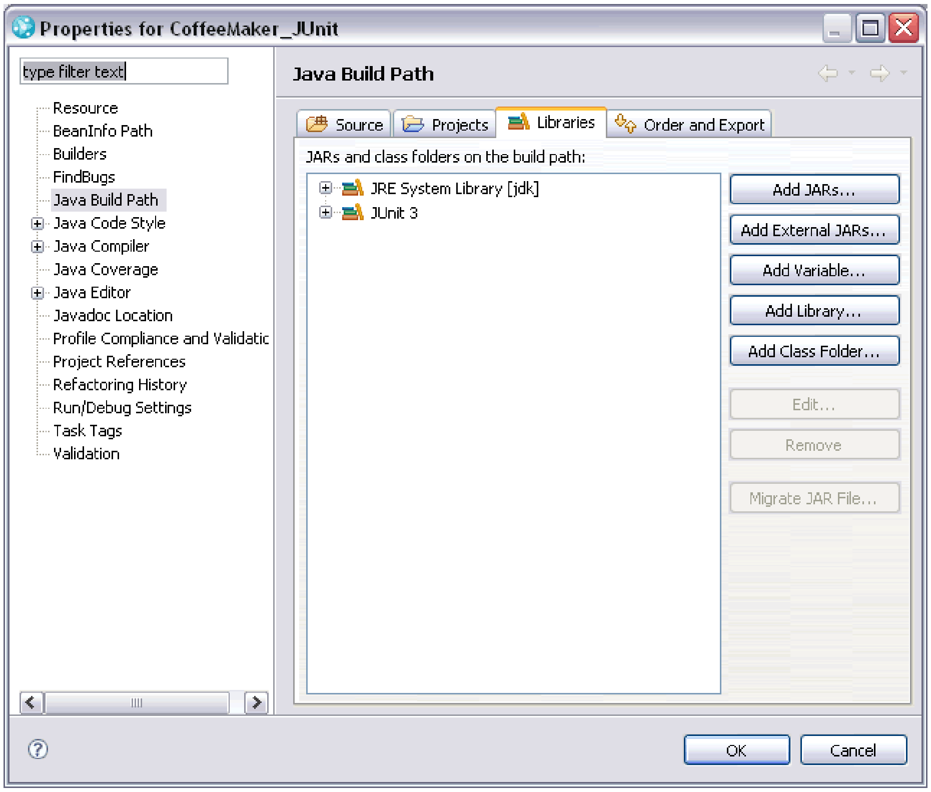
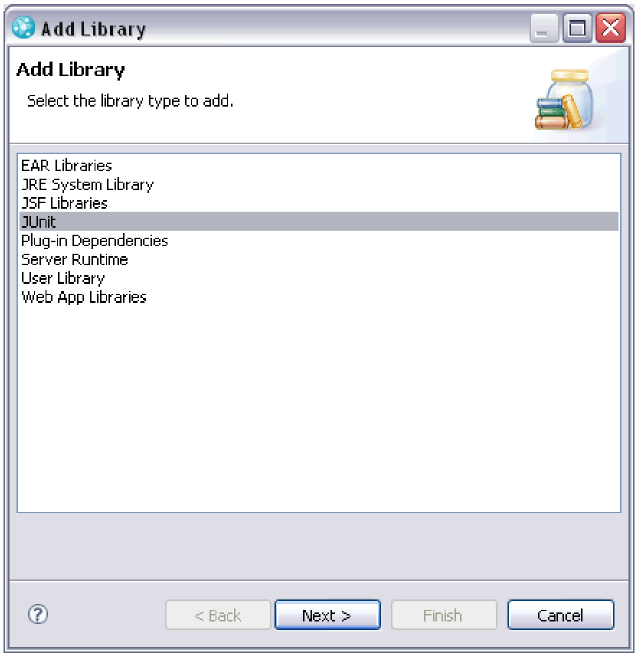


Figure 2: Java Build Path for a Java Project

1.1.2. Select the **Add Library...** button to the right of the window. Select **JUnit** from the library list, and press the **Next >** button.



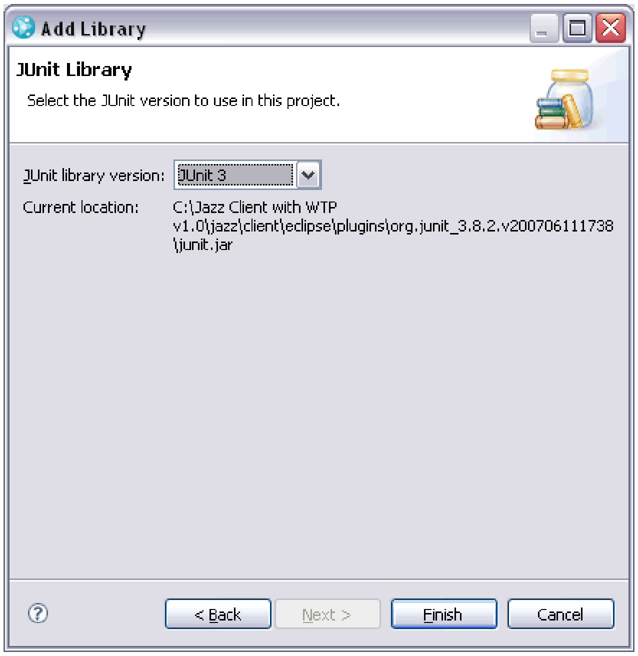
Add Library

Select the library type to add.

EAR Libraries
JRE System Library
JSF Libraries
JUnit
Plug-in Dependencies
Server Runtime
User Library
Web App Libraries

Figure 3: Add Library window.

1.1.3 Use the drop down box to select the version of JUnit that you want to use for your project. We are using **JUnit 3** for the CoffeeMaker example in this tutorial. The location of your JUnit 3 library may vary from the one pictured below.



JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 3

Current location: C:\Jazz Client with WTP v1.0\jazz\client\eclipse\plugins\org.junit_3.8.2.v200706111738 \junit.jar

Figure 4: Select version for JUnit library

Top | Contents

2.0 Creating a Test Class

JUnit convention is that there is a test class created for every application class that does not contain GUI code. Usually all paths through each method in a class are tested in a unit test; however, you do not need to test trivial getter and setter methods.

2.1 There are many ways to create a JUnit Test Case Class.

2.1.1 Select **File > New > JUnit Test Case**

2.1.2 Select the arrow of the button in the upper left of the toolbar. Select **JUnit Test Case** ,

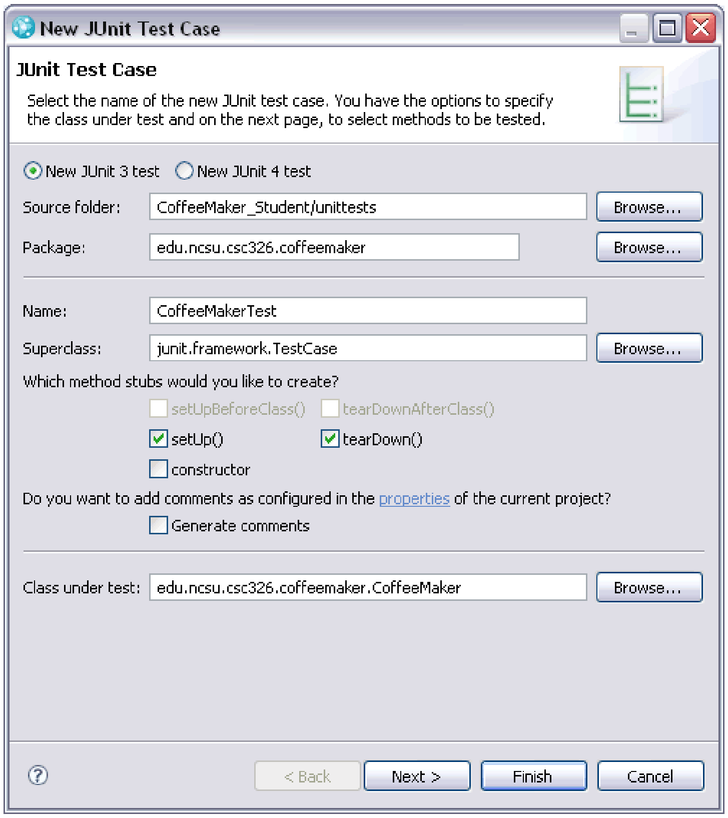
2.1.3 Right click on a package in the Package Explorer view in the Java Perspective, and select **JUnit Test Case**, or

2.1.4 Click on the arrow of the icon in the toolbar. Select **JUnit Test Case** .

2.1.5 You can create a normal Java class as shown in the Eclipse tutorial, but include *junit.framework.TestCase* as the super class of the test class you are creating.

2.1.6 Right click on an application class that you want to create a test class for, and select **New > JUnit Test Case**.

2.2 The wizard for creating a new JUnit test case classes is displayed in Figure 5. If you use the method of creating a JUnit test case presented in step 2.1.6, most of the information below will be filled in for you, but may require minor modifications.



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

● New JUnit 3 test ○ New JUnit 4 test

Source folder: CoffeeMaker_Student/unittests

Package: edu.ncsu.csc326.coffeemaker

Name: CoffeeMakerTest

Superclass: junit.framework.TestCase

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()

☑ setUp() ☑ tearDown()

□ constructor

Do you want to add comments as configured in the properties of the current project?

□ Generate comments

Class under test: edu.ncsu.csc326.coffeemaker.CoffeeMaker

Figure 5: JUnit test case class creation wizard

2.2.1 The source folder specifies where you want to create the test case. Based on the project hierarchy presented in Figure 1, we want to create the new test case in the *unittests/* source folder.

2.2.2 Next, specify the package you want your test case to belong to. If you are associating your test case with an application class, then you will want both of these classes to share the same package so you can take advantage of package level visibility when testing your application class.

2.2.3 Specify the test case name. If you are associating your test case with an application class the convention is that the test case name is the application class name followed by Test.

2.2.4 JUnit has several standard methods that are useful for testing. setUp() and tearDown() methods are run before and after each test method.

2.2.5 If you are associating your test case with an application class, then **Browse** for the application class in the **Class under test** field.

2.3 If you selected a **Class under test** you can click the **Next** button to select which methods you want to write test cases for. The method signatures will be created for you. Click **Finish** and the new test case class will open in the editor.

2.4 Below is a test case template from the **JUnit 3 FAQ**. This test class demonstrates the basic functionality of the `setUp()` and `tearDown()` methods, and gives example test cases. The `testForException()` method demonstrates how to test that an exception is properly thrown.

Note: All source methods in the class under test must be public or protected, not private, in order to be tested by JUnit. If the method in the class under test is protected, the test class must be in the same package.

```
import junit.framework.TestCase;


public class SampleTest extends TestCase {


     private java.util.List emptyList;


     /**
      * Sets up the test fixture.
      * (Called before every test case method.)
      */
     protected void setUp() {
          emptyList = new java.util.ArrayList();
     }


     /**
      * Tears down the test fixture.
      * (Called after every test case method.)
      */
     protected void tearDown() {
          emptyList = null;
     }
     public void testSomeBehavior() {
          assertEquals("Empty list should have 0 elements", 0, emptyList.size());
     }

     public void testForException() {
          try {
               Object o = emptyList.get(0);
               fail("Should raise an IndexOutOfBoundsException");
          }
          catch (IndexOutOfBoundsException success) {
          }
     }

}
```

2.5 Below is a test case template based on the examples presented in the **JUnit 4 FAQ**. This test class demonstrates the basic functionality of the `setUp()` and `tearDown()` methods, and gives example test cases. The `testForException()` method demonstrates how to test that an exception is properly thrown.

Note: All source methods in the class under test must be public or protected, not private, in order to be tested by JUnit. If the method in the class under test is protected, the test class must be in the same package.

```
import org.junit.*;
import static org.junit.Assert.*;


public class SampleTest {


private java.util.List emptyList;


    /**
     * Sets up the test fixture.
     * (Called before every test case method.)
     */
    @Before
    public void setUp() {
        emptyList = new java.util.ArrayList();
    }


    /**
     * Tears down the test fixture.
     * (Called after every test case method.)
     */
    @After
    public void tearDown() {
        emptyList = null;
    }


    @Test
    public void testSomeBehavior() {
        assertEquals("Empty list should have 0 elements", 0, emptyList.size());
    }

    @Test(expected=IndexOutOfBoundsException.class)
    public void testForException() {
        Object o = emptyList.get(0);
    }
}
```

Top | Contents

3.0 Creating a Test Suite

A TestSuite is a simple way of running one class that, in turn, runs all test cases at one time.

3.1 There are four ways to create a JUnit Test Suite Class. First, select the directory (usually *unittests/*) that you wish to create the test suite class in.

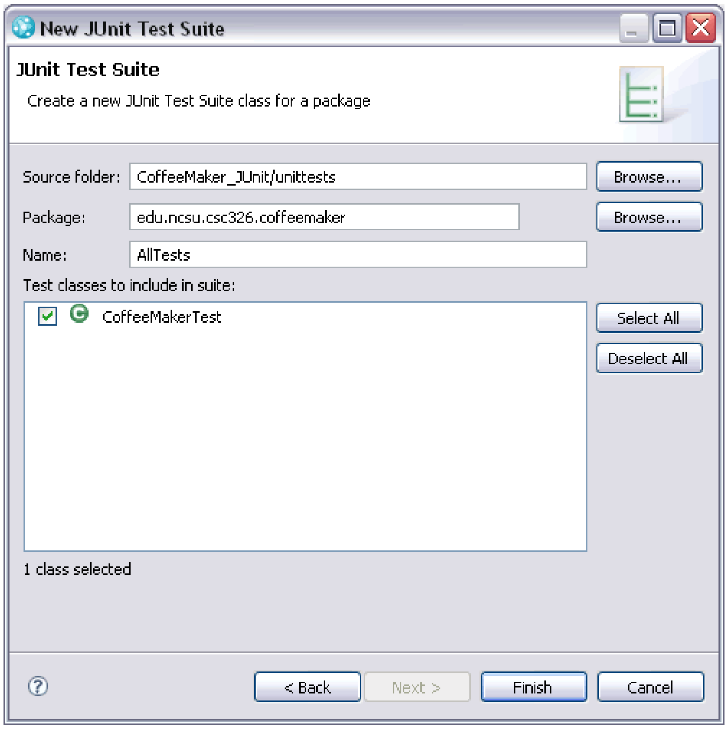
3.1.1 Select **File > New > Other... > Java > JUnit > JUnit Test Suite.**

3.1.2 Select the arrow of the button in the upper left of the toolbar. Select **Other... > Java > JUnit > JUnit Test Suite**,

3.1.3 Right click on a package in the Package Explorer view in the Java Perspective, and select **New > Other... > Java > JUnit > JUnit Test Suite**, or

3.1.4 You can create a normal Java class as shown in the Eclipse tutorial, but include *junit.framework.TestSuite* as the super class of the test class you are creating.

3.2 Check to make sure that you are creating the TestSuite in the proper source folder and the proper package. Give the test suite a name. The default name is AllTests.java



New JUnit Test Suite

JUnit Test Suite

Create a new JUnit Test Suite class for a package

Source folder: CoffeeMaker_JUnit/unittests

Package: edu.ncsu.csc326.coffeemaker

Name: AllTests

Test classes to include in suite:

☑ CoffeeMakerTest

1 class selected

Figure 6: Create JUnit test suite wizard

3.2.1 Use the **Browse** buttons to search for a source folder, and the package.

3.2.2 Select which test classes you would like to include in the test suite.

3.3 Click **Finish**. The new test suite class will be open in the editor.

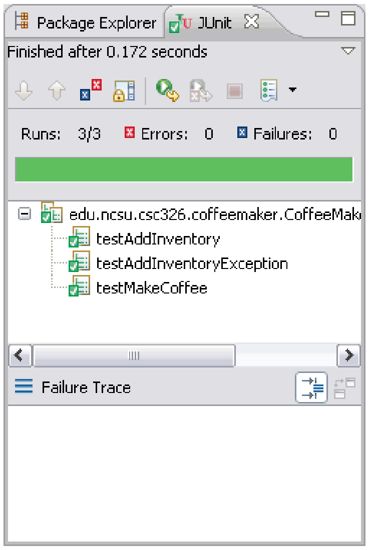
3.4 Below is a test suite template from the **JUnit 3** FAQ. This test suite demonstrates the basic functionality of the *suite()* method, which is what you add each of the test cases to the suite in. This should all be generated for you by Eclipse if you use the first 3 methods in step 3.1 to create the Test Suite.

```
import junit.framework.Test;
import junit.framework.TestSuite;


public class SampleTestSuite {


     public static Test suite() {
          TestSuite suite = new TestSuite("Sample Tests");

          // Add one entry for each test class
          // or test suite.
          suite.addTestSuite(SampleTest.class);

          // For a master test suite, use this pattern.
          // (Note that here, it's recursive!)
          suite.addTest(AnotherTestSuite.suite());

          return suite;
     }

}
```

Top | Contents

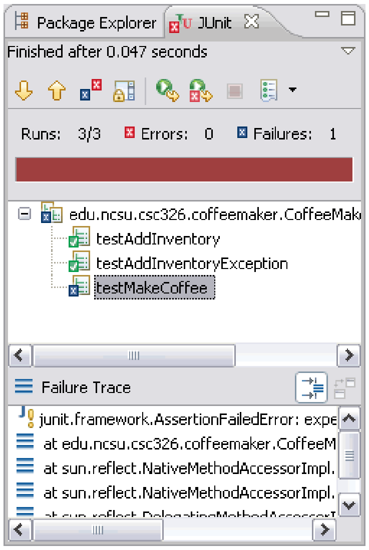
4.0 Running JUnit Test Cases

The goal when running JUnit test cases is a green bar in the JUnit view. The green bar shows that all of the test cases passed. A red bar shows that one or more of the test cases failed. The stack trace of a test failure or error are displayed below the listing of the test cases run. Figures 7 and 8 show the green and red bars in the JUnit view.



```
Finished after 0.172 seconds
Runs: 3/3   Errors: 0   Failures: 0
edu.ncsu.csc326.coffeemaker.CoffeeMak
    testAddInventory
    testAddInventoryException
    testMakeCoffee
```

Figure 7: JUnit green bar



```
Finished after 0.047 seconds
Runs: 3/3   Errors: 0   Failures: 1
edu.ncsu.csc326.coffeemaker.CoffeeMak
    testAddInventory
    testAddInventoryException
    testMakeCoffee
junit.framework.AssertionFailedError: expe
at edu.ncsu.csc326.coffeemaker.CoffeeM
at sun.reflect.NativeMethodAccessorImpl.
at sun.reflect.NativeMethodAccessorImpl.
at sun.reflect.DelegatingMethodAccessorI
```

Figure 8: JUnit red bar.

4.1 There are three ways to run JUnit Test Cases or Test Suites.

4.1.1 You can right click on the test case class or test suite class and select **Run As > JUnit Test.**

4.1.2 You can select a test case or suite and click the arrow on the icon or select Run from the toolbar, and select **Run As > JUnit Test.**

4.1.3 You can select a test case or suite and click the arrow on the icon or select Run from the toolbar, and select **Open Run Dialog...** From here you will create a new JUnit test configuration, and name it. You can choose to run a single test case, a test suite, or run all test cases in a project or folder.

Note: You should be careful here if you have FIT tests in the project. Running all of the test cases in the project will run those FIT tests and give you errors. The best idea is to just run the tests on the *unittest/* source folder or whichever source folder contains your tests.

Top | Contents

5.0 Assertion Statement Reference

In JUnit, a test occurs when ever you assert that something should be a specific value. Each JUnit test case (method) should have at least one of the assert statements listed below, otherwise the test passes, which can be misleading if you never actually check the value of any data. The most common assert statement is `assertEquals()` which takes at least two arguments. The first argument is the expected value of some piece of data at that point in the test case. The second argument is the actual value of that data, usually obtained by a method call, at that point in the test case. The method will check for equality between the two pieces of data. For a custom object this means that the `equals()` and `hashCode()` methods should be overridden. Otherwise, the `assertEquals()` method will check for equality of the objects location in memory. These will be different because the expected object is created within the test case while the actual object is created within the application.

If any of your assert statement fail, the test progress bar in the JUnit view will be red. Additionally, a failing assert statement will cease execution for that test case. Therefore, assert statements after that point will not be executed.

This is a list of the different types of assertion statements that are used to test your code. Any Java data type or object can be used in the statement. These assertions are taken from the JUnit API.

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
- assertEquals(message, expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
- assertFalse(condition)
- assertFalse(message, condition)
- assertNotNull(object)
- assertNotNull(message, object)
- assertNotSame(expected, actual)
- assertNotSame(message, expected, actual)
- assertNull(object)
- assertNull(message, object)
- assertSame(expected, actual)
- assertSame(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)
- fail()
- fail(message)
- failNotEquals(message, expected, actual)
- failNotSame(message, expected, actual)
- failSame(message)

Top | Contents

6.0 Exercise

For this exercise we will be using the CoffeeMaker project. Unzip the CoffeeMaker project to your home directory and import the project into Jazz. Please see the Eclipse Import/Export Guide for instructions on how to import a project into Jazz (which is built on Eclipse).

We all know that most computer scientists love caffeine, so the Computer Science department is looking to put a coffee kiosk in EBII building. The coffee kiosk must be able to make coffee for students to purchase. Take a look at the CoffeeMaker User Stories/Requirements to look for boundaries and other things to test.

The CoffeeMaker code is complete; however, we need you to create and run unit tests on the following user stories: **1) Add a Recipe, 2) Delete a Recipe, 3) Edit a Recipe, 4) Add Inventory, 5) Check Inventory, and 6) Purchase Coffee**. One test class has been created for you: `CoffeeMakerTest` under the *unittests/* directory. You can create `RecipeTest` and `InventoryTest` classes as well. There are currently 5 (very obvious) bugs in the system. We need you to generate enough unit tests to find 2 of the 5 bugs. Once you find the bugs, create a fix (These should be very simple fixes. If the fix takes longer than 5 minutes, you found a bigger bug than the one we wanted you to find! - and you should let your TA know). Create a list of the bugs that you find, and submit the CoffeeMaker project to your TA with a green bar!

Deliverables to the TA

- List of bugs found in the Add a Recipe, Delete a Recipe, Edit a Recipe, Add Inventory, Check Inventory, and Purchase Beverage user stories in CoffeeMaker
- Fixed CoffeeMakert project where all JUnit test cases pass.

Top | Contents

7.0 Resources

- JUnit
- JUnit FAQ
- JUnit API
- Eclipse
- CoffeeMaker Example

Top | Contents

Back to Software Engineering Tutorials

Email the authors with any questions or comments about this tutorial.
Last Updated: Thursday, August 28, 2008 4:28 PM