Lab 2- Automated Requirements-Based API Unit Testing using JUnit

Developed and maintained by: Dr. Vahid Garousi
vgarousi@gmail.com
www.vgarousi.com

Part of the Open Software Testing Laboratory Courseware:
sites.google.com/view/software-testing-labs

TABLE OF CONTENTS

REVISION HISTORY OF THIS DOCUMENT:

Summer 2008	First version was developed by Dr. Vahid Garousi and his team at U of Calgary
September 2010-2017	Various improvements were made, using experience from using these lab in various course offerings and feedbacks from students
July 2020	The lab document was updated based on the latest version of the Zoho issue (bug) tracking system
Fall 2021	Made various improvements using student comments, provided between 2017-2021

1 INTRODUCTION

1.1 OBJECTIVES

There are several key objectives of this lab. The first is to introduce students to the fundamentals of automated unit testing, specifically unit testing based on requirements for each unit. The most widely used unit testing tool for Java is the JUnit framework, which is a part of the XUnit framework family.

After completing the lab, students will be able: to develop automated test code in JUnit and other XUnit testing frameworks such as NUnit, CSUnit, PHPUnit, etc.

1.2 THIS LAB IS A GROUP WORK

All the tasks of this lab should be completed in groups of two (as you have formed before). One single report has to be completed and submitted for each group. In the report, there should be a section which should discuss "How the team work/effort was divided and managed". As we discuss in Section 4, a Word template file is provided by the professor which already has a heading for this purpose.

1.3 TOOLS TO BE USED IN THIS LAB

1.3.1 JUnit testing framework

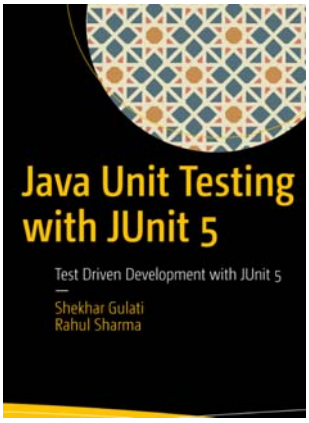
The main testing tool for this lab is JUnit (www.junit.org). JUnit is a widely-used unit testing framework for Java. JUnit has been important in the development of test-driven development (TDD), and is one of a family of unit testing frameworks which is collectively known as *xUnit* (www.wikipedia.org/wiki/XUnit), e.g., NUnit for .Net, PHPUnit for PHP, CppUnit for C++, etc.

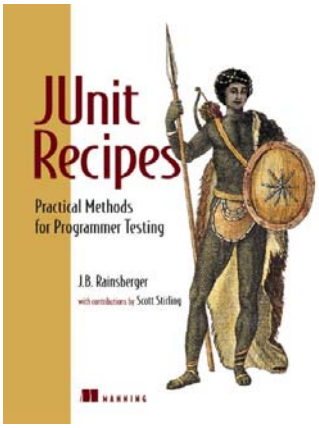
JUnit was first developed in early 2000's by a team of practitioners and soon "*JUnit took off like a rocket - and was essential to supporting the growing movement of Extreme Programming and Test Driven Development*"[1]. JUnit has evolved through the years, and each new version has brought new features and improvements in it. As of 2021, its latest version is 5[2]. But for this lab, we prefer to use the version 4, since it is better for learning purposes.

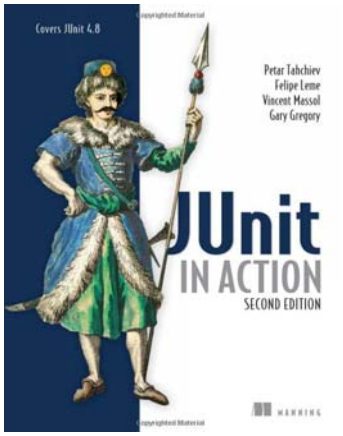
JUnit has seamless integration with many Integrated Development Environments (IDE's), e.g., the Eclipse development environment [1]. The JUnit framework allows developers to quickly and easily develop unit tests and test suites, and execute them.

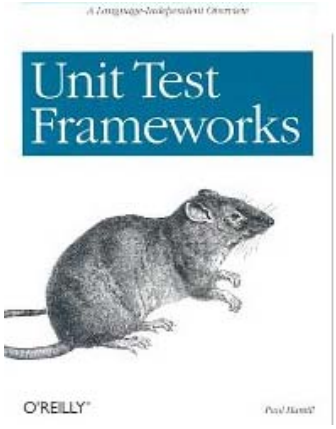
There are many online resources and books on JUnit. Some of the many books on JUnit are shown below:

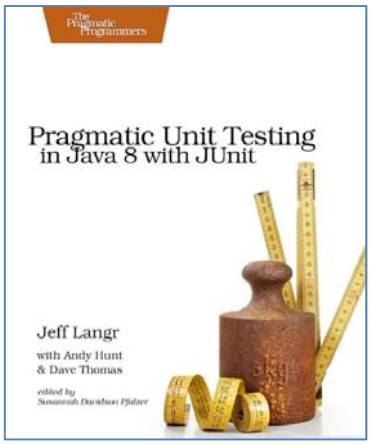
[1] martinfowler.com/bliki/Xunit.html
[2] mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api











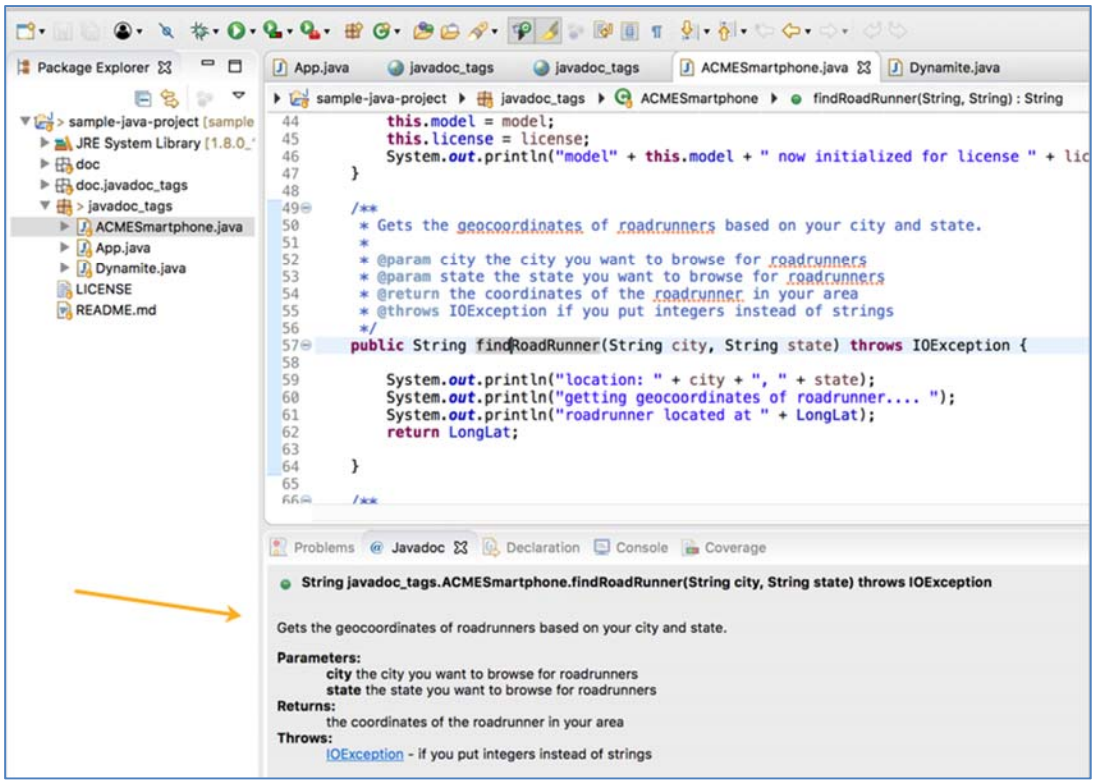
1.3.2 Javadoc

Another tool which is often used for the purpose of testing is Javadoc (www.wikipedia.org/wiki/ Javadoc). Javadoc is a software documentation generator created by Sun Microsystems for Java for generating API documentation in HTML format from Java source code.

While Javadoc is not a testing tool, it will be used in the context of this lab as the format in which the requirements specification of the System Under Test (SUT) has been specified in. We will use requirements specification in the Javadoc to derive black-box test cases.

Developing the documentation and the code in the same location does not only improve communication between developers, maintainers and testers, but it also makes it simpler to keep the documentation up to date, and prevent potential redundancies and/or mistakes. For more information on Javadoc, see java.sun.com/j2se/javadoc.

A screenshot of how documentation using Javadoc is developed and derived is shown in the following.



1.4 SOFTWARE SYSTEM UNDER TEST (SUT)

The System Under Test (SUT) which we will test in this lab is JFreeChart (www.jfree.org/jfreechart). JFreeChart is an open source Java framework for programmatically creating and displaying charts in Java. This framework supports many different (graphical) chart types, including pie charts, bar charts, line charts, histograms, and several other chart types. A snapshot of four different types of charts drawn using JFreeChart is shown in Figure 1.

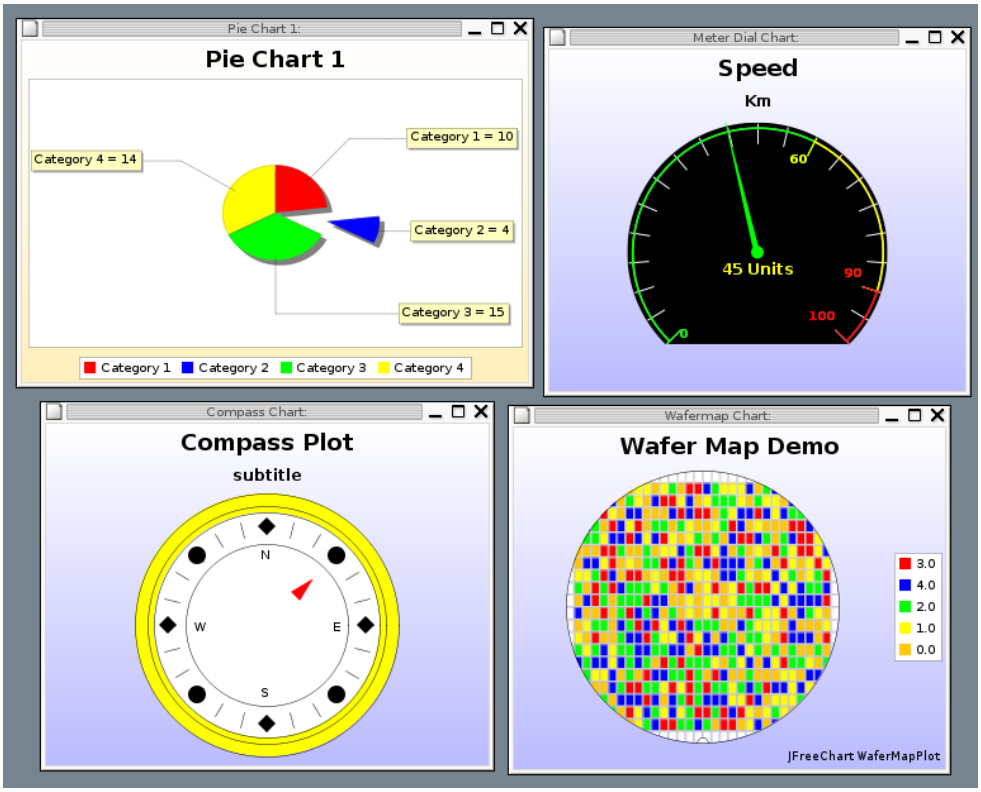


Figure 1 - A snapshot of four different types of charts drawn using JFreeChart.

To get started with the JFreeChart system, download the lab artifacts and extract the entire archive to a known location. Then, find "JFreeChart v1.0.zip" file inside the artifacts. More information on how to get started with these files will be provided in the familiarization stage (Section 2.1). Note that the versions of JFreeChart distributed for this lab do not correspond with latest releases of JFreeChart. The versions have been slightly modified for the purposes of this lab.

The JFreeChart framework is intended to be integrated into other systems as a quick and simple way to add charting functionality to Java applications. With this in mind, the API for JFreeChart is required to be relatively simple to understand, as it is intended to be used by many developers as an open source off-the-shelf framework.

While the JFreeChart system is not technically a stand-alone application, the developers of JFreeChart have created several demo classes which can be executed to show some of the capabilities of the system. These demo classes have *Demo* appended to the class name. For the purpose of this lab, full knowledge of the usage of the JFreeChart API is not particularly necessary.

1.5 Overview of the Lab Work

The lab-work is divided into three main sections. Similar to lab 1, the first section is familiarization (Section 2.1). During the familiarization stage, students will be shown how to set up a JUnit test project in Eclipse using JUnit, develop a simple unit test, and how to navigate through software requirements of the SUT (JFreeChart) in the Javadoc documentation.

After familiarization, the second section of the lab is designing the test cases using the software requirements specified in Javadoc, and development of unit tests. In this stage, students will develop unit test suites for several classes of the SUT. These test suites will be comprised of unit tests which have been generated based on the requirements.

Finally, upon completion of the test suites, during the last stage, the tests will be executed on several versions of the SUT and test results will be collected and recorded.

Deliverables of the lab will include: lab report (50% of the lab mark), and JUnit test suite (50% of the lab mark). Details about deliverables and grading are discussed in Section 4.

2 Instructions

This section details the instructions for executing the lab. All sections of this lab should be completed as a group as students have already formed.

2.1 FAMILIARIZATION

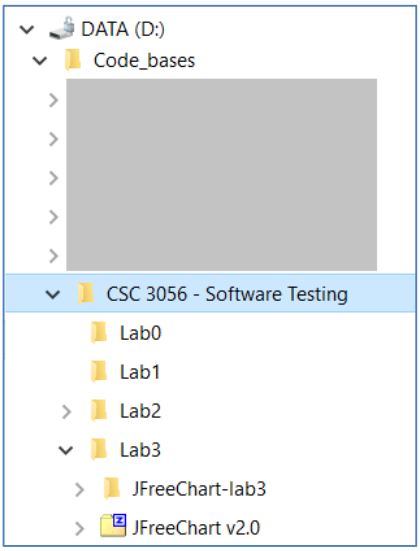
Both students of each group should perform this section of the lab together on a single computer. Ensure that both of you understand the concepts in this section before moving on to the rest of the lab.

1. Download the "Lab 2- artifacts" ZIP (RAR) file from the course website.
2. Extract the contents of the ZIP RAR file into a known location. The contents should include: the *JFreeChart v1.0.zip*

2.1.1 Creating a new Eclipse project

3. Open Eclipse Java IDE
4. Open the *New Project* dialog by selecting the *File -> New -> Project…*
5. Dr. Garousi has recorded two videos to help students with on setting up JUnit test projects. To continue your project setup, make sure to watch them and ensure doing everything in the exact same way:
   - Part 1: https://youtu.be/_bGb0cO1pyg
   - Part 2: https://youtu.be/6kRcq2QTUgQ
   - Creating a new Eclipse project and loading the SUT (JFreeChart): youtube.com/watch?v=XbjnJLxpquc

Strong recommendation: We recommend that you structure your folder structure like the following: having a separate folder for each lab in this course:



2.1.2 Adding the necessary Java libraries to your project

6. The *Java Settings* dialog should now be displayed. This dialog has four tabs along the top: *Source*, *Projects*, *Libraries*, and *Order and Export*.
7. Move to the *Libraries* tab, and click the *Add External JARs (or Libraries)…* button. Select the *jfreechart.jar* file from the known location and click *Open*. JAR stands for *Java ARchive* (more info: en.wikipedia.org/wiki/JAR_(file_format)).
8. As the next step, click *Add External JAR…* again. This time, add all the *.jar* files from the *lib* directory where you have unzipped the *JFreeChart v1.0.zip* file. The Java Settings dialog should now look like Figure 2, below.

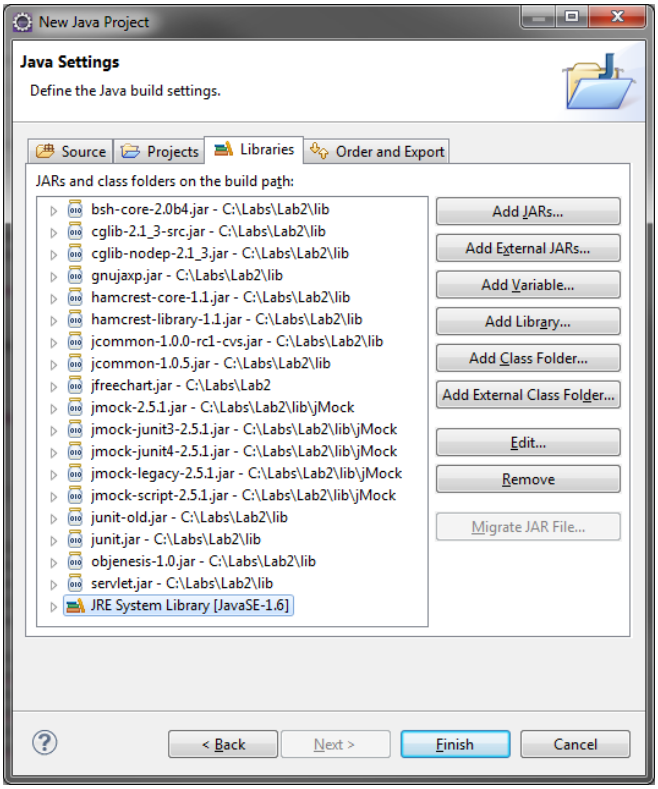


Figure 2 - The Java Settings dialog after adding required archives

9. Click *Finish*. The project (SUT) is now set up and ready for testing.

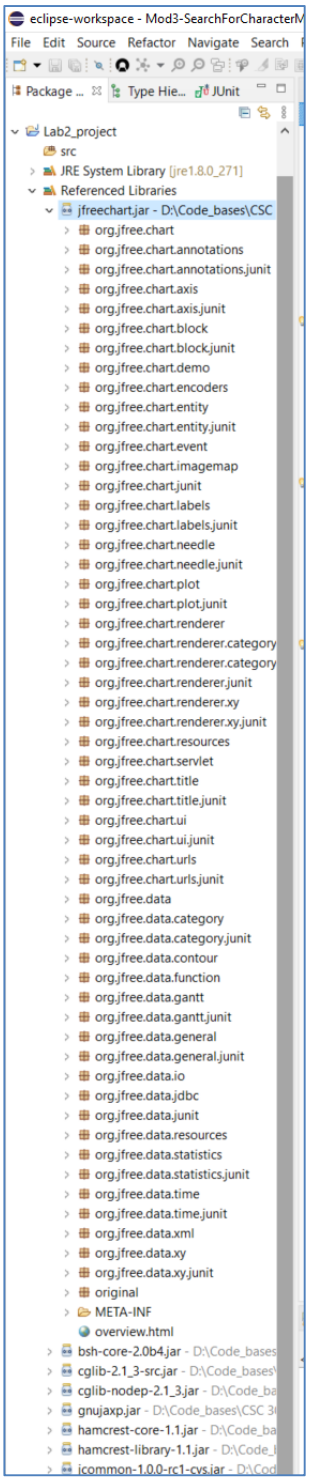
2.1.3 Reviewing the SUT (JFreeChart) framework and running its demos

10. Let's take a few minutes to review the SUT (JFreeChart framework). Of course, in this lab, we do not have its source code and only have its executable file (JAR: *Java ARchive* file format).

11. We can see the contents of the JFreeChart library by going to *Reference Libraries* node in your newly-created project, see the large screenshot on the right side →

12. As you can see in the tree structure of JFreeChart JAR file, the framework is grouped into two main packages, (1) `org.jfree.chart` and (2) `org.jfree.data`. Each of these two packages is also divided into several other smaller packages. For the purpose of testing in this lab, we will be focusing on the `org.jfree.data` package.

13. The JFreeChart JAR file comes with several demo classes, each showing a type of chart which can be generated by JFreeChart. In the package explorer, expand the *Referenced Libraries* item in the newly-created JFreeChart project, exposing the .jar files just added. Right click on the *jfreechart.jar*, and select *Run As -> Java Application* (Figure 3).



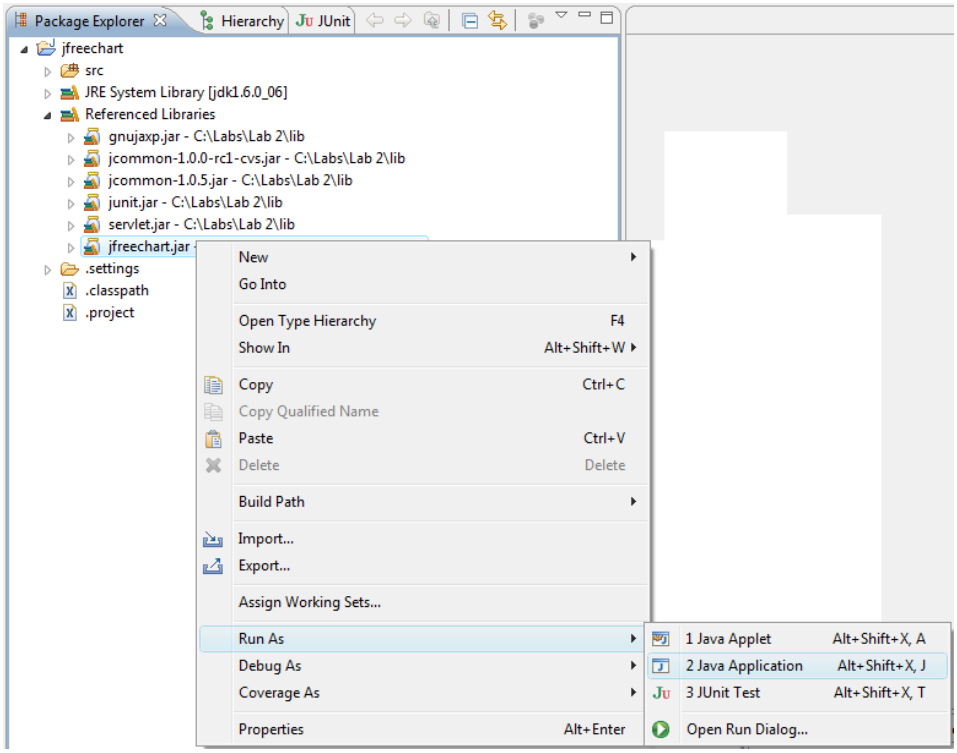


Figure 3 - Running demo classes of JFreeChart

14. In the *Select Java Application* dialog, select any of the demo applications (e.g., *CompositeDemo*), and click *OK* as shown in Figure 4. You should see some demo charts as shown in Figure 4.

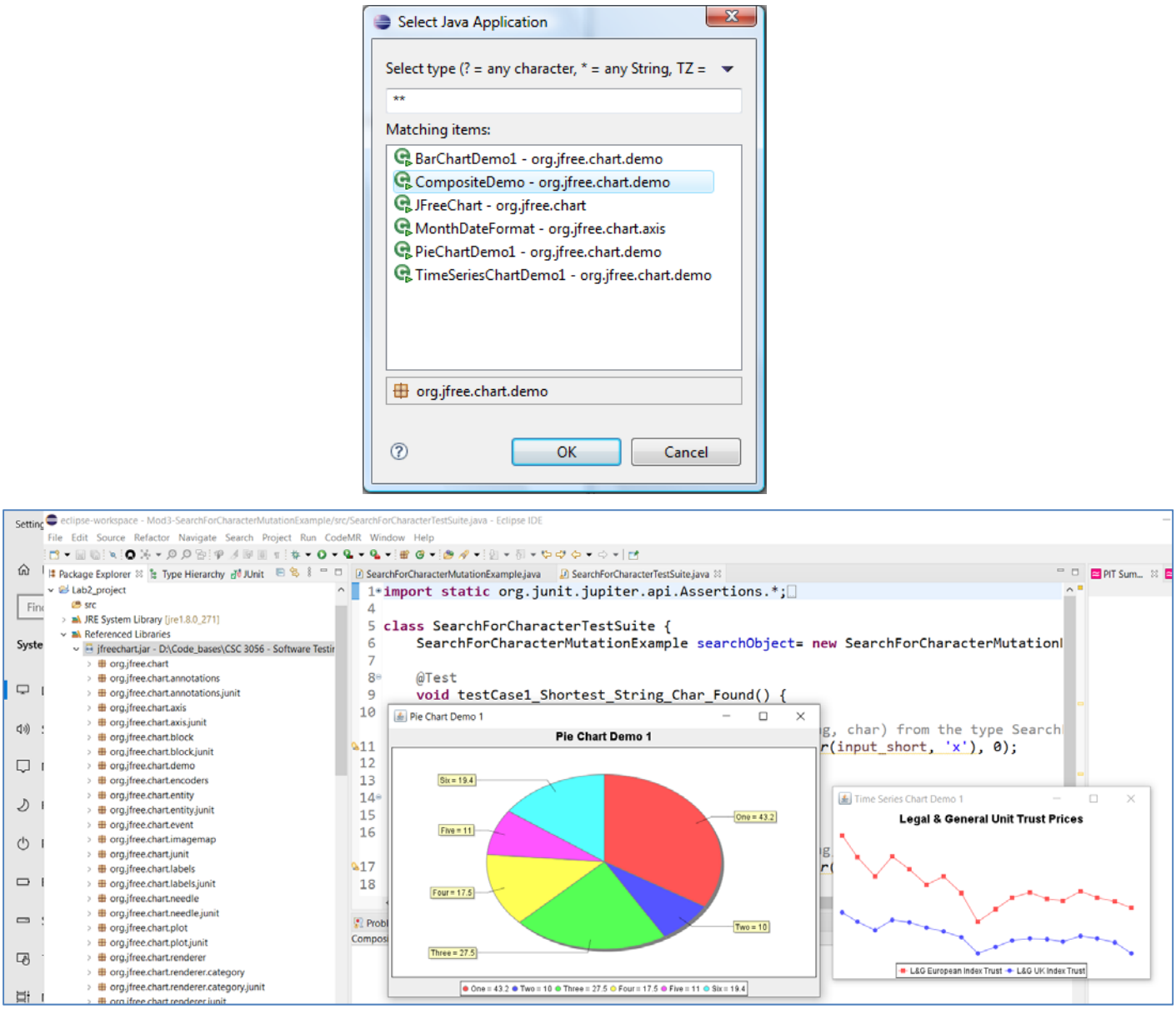


Figure 4 - Running demo classes of JFreeChart

Note: Feel free to ask the instructor or the Teaching Assistant (TA) – if we have in a given term - if you have any issues so far.

2.1.4 Navigating Javadoc API specifications

In Section 2.1.6 of this lab document, the document will ask you to design and develop unit tests for a number of classes based on specifications (requirements) contained in the Javadocs for those classes. As you have learned in the lectures, this is called Black-box testing.

15. Unzip the *JFreeChart–ModifiedJavadoc.zip* file and open the file *index.html*. Note the location of different elements in the Javadoc as shown in Appendix B. Note that Javadocs can be browsed with all classes shown, or with classes filtered by package. Each of these two approaches has its usefulness. Viewing all classes is useful if you know what the class you are looking for is called, as they are ordered alphabetically. Viewing classes in a single package only is useful for when you're not sure exactly what class you're looking for, but know what area of the code it might be found in.

16. In the list of packages (top-left corner), scroll down to find the `org.jfree.data` package and click on it. This should show a list of classes in the class list (bottom left corner) now.

17. In the class list, click on the `Range` class. The main content pane now shows the API specifications of the `Range`. Scroll down and notice the layout of the specification. At the very top is a description of the class itself (including inheritance information), followed by nested classes, attributes, methods (starting with any constructors), inherited methods, and finally the detailed specification for each method; as shown below.

Figure 5 – Reviewing classes and their properties in Javadoc API specifications

18. Take note of the information available in the *Method Summary* and *Method Detail* sections of the main content pane, as this is what you will be testing methods against (as test oracle) in the following section. For especially effective tests, however, the specifications need to be specific, precise, clear and complete.

2.1.5 Developing a simple JUnit test class for the `org.jfree.data.Range` class

As an example class under test (CUT), let us choose the `org.jfree.data.Range`. You can find its API specification, in the Javadoc as shown in Figure 5. For developing tests for any class, you need to review the API specification of the class: its methods and their inputs and outputs, as shown in Figure 5 for this class under test (CUT).

To create a test suite containing a single unit test in JUnit, follow these steps.

19. In the package explorer, expand the *Referenced Libraries* list item to show all the archives that the project uses.
20. Expand the *jfreechart.jar* archive to expose all the packages that are contained in that archive.
21. Expand the `org.jfree.data` package within the archive to show all the .class files contained in that package.
22. Finally, expand the `Range.class` item to expose the class contained in that file, along with all the class' methods and fields contained in that class; as shown in Figure 6.
23. Right click on the `Range` class (has a green 'C' icon, denoting it is a class), and select *New -> JUnit Test Case*. Type *RangeTest* in the *Name* field and type '`.test`' in the *Package* field at the end of the default package to create a new package for the test cases. Ensuring that test classes are in a separate package makes it easier to keep the two apart.

Also, ensure that the superclass is as specified in Figure 6. Also, make sure that you have checked the boxes for the method stubs specified in Figure 6, e.g., `setUp()`

24. Click *Finish*.

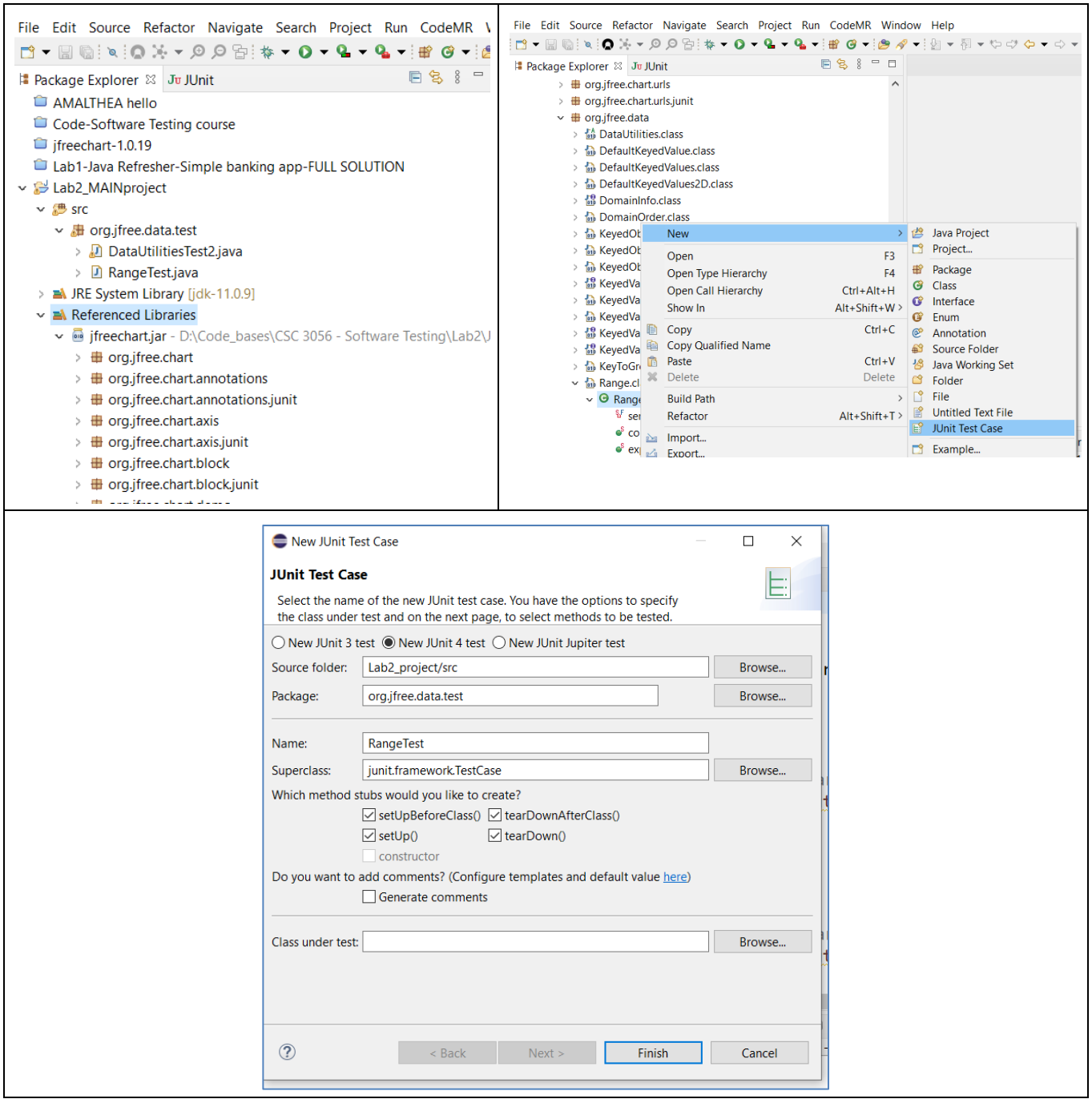


Figure 6 - Creating a new JUnit test case for the `org.jfree.data.Range` class

25. The newly-created test class (`RangeTest`) extends `TestCase`, the super class of all test cases as defined in JUnit. The `TestCase` class is a JUnit class which provides a unified interface for test cases and suites. A set of test cases which test similar functions of a class may all want to initialize an instance of that class and prepare for the test cases in the same way. The way that JUnit facilitates this is the `setUp()` and `tearDown()` methods. Write the `setUp()` and `tearDown()` methods for the `RangeTest` class as shown in Figure 7, including the `testRange` attribute which is required for the test cases to have access to the object initialized in the `setUp()` method.

```
package org.jfree.data.test;

import junit.framework.TestCase;
import org.jfree.data.Range;
import org.junit.*;

public class RangeTest extends TestCase {

   private Range rangeObjectUnderTest;

   @Before
   public void setUp() throws Exception {
      rangeObjectUnderTest = new Range(-1, 1);
   }

   @After
   public void tearDown() throws Exception {
   }

}
```

Figure 7 - Simple `setUp()` and `tearDown()`

26. It is possible that the IDE will tell you that the `Range` class cannot be resolved. It is best to use the IDE's QuickFix feature to fix minor issues like this. Choose to import the class from `org.jfree.data.Range` in this case, as shown in Figure 8.

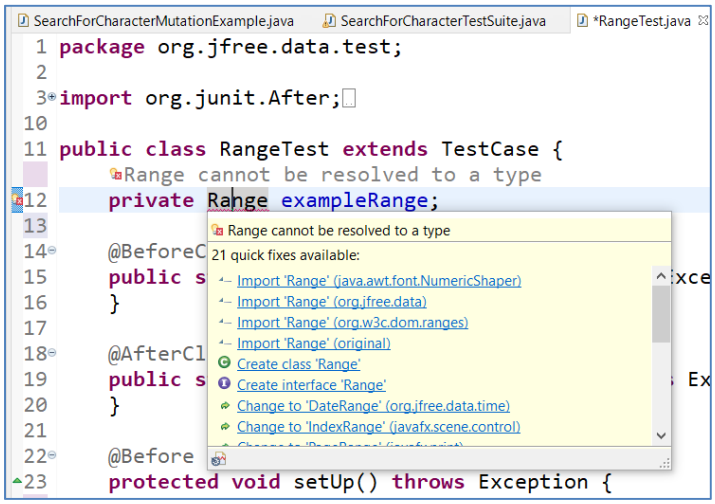


Figure 8–Resolving package/module imports using the IDE's QuickFix feature for the Range class

27. Test cases in JUnit are individual methods which are usually prefixed with the word "test", for example: `testCentralValue()`. The `@Test` annotations are also the identifiers. These test cases can perform any number of steps, and should follow four phase testing (setup, exercise, verify, and teardown). The test oracle for each test case in JUnit can be implemented in several ways, using assertions being the recommended one. An example assertion is:

```
assertTrue(("example string").length() == 14);
```

… which will always pass (assuming that the `length()` method works as expected). The syntax of the above assertion as defined in JUnit is:

```
public static void assertTrue([java.lang.String message,]
                              boolean condition)
```

… which asserts that a condition is true. If it is not, the assert throws an `AssertionFailedError` with the given message (if any). Assertions are tested when a test method is executed, and will cause a test to fail if the assert conditions are not met. For a complete list of all the assertions available using JUnit, refer to Appendix A.

28. As a practice, write a simple test case for the `getCentralValue()` method as shown in Figure 9 below. The syntax for `assertEquals` is as follows:

```
public static void assertEquals(java.lang.String message,
                                double expected,
                                double actual,
                                double delta)
```

`assertEquals` asserts that two values are equal. This particular version of the method also has a delta value, to allow for some variance when comparing floats (since it can be difficult to tell if a value is going to be 1.500000001 or exactly 1.5 in the present of inevitable automatic rounding of floating point numbers in Java). If the values are not equal, an `AssertionFailedError` is thrown with the given message. The `expected` argument is the test oracle which we should derive from the requirements.

```
package org.jfree.data.test;

import junit.framework.TestCase;
import org.jfree.data.Range;
import org.junit.*;

public class RangeTest extends TestCase {

   private Range rangeObjectUnderTest;

   @Before
   public void setUp() throws Exception {
      rangeObjectUnderTest = new Range(-1, 1);
   }

   @After
   public void tearDown() throws Exception {
   }

   @Test
   public void testCentralValueShouldBeZero() {
      assertEquals("The central value of -1 and 1 should be 0",
                   0, rangeObjectUnderTest.getCentralValue(), 0.000000001d);
   }
}
```

Figure 9 – The `RangeTest` test class after addition of a simple test case (test method)

29. Now that you have a completed test case, run the test class. To do this, right click on the `RangeTest` class in the Package Explorer and select *Run As -> JUnit Test*.

30. This will change the perspective to the JUnit perspective, and run all the tests within the `RangeTest` class. The test just written should pass, indicated by the JUnit view as shown in Figure 4 below. In JUnit a *Failure* and *Error* are similar, but differ slightly. If you get an error it means that your test method did not execute as expected (i.e., an uncaught exception), a failure, however, means that the execution was as expected, but an assertion failed.

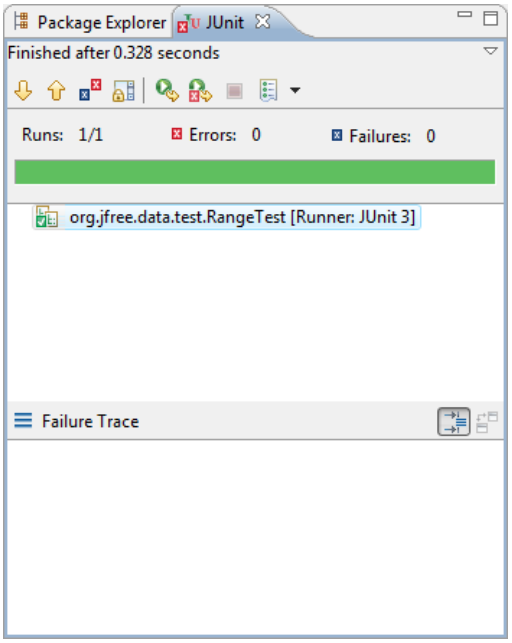


Figure 10 - JUnit view in Eclipse showing a passed test method

2.1.6 Developing a simple JUnit test class for the `org.jfree.data.DataUtilities` class

Let us show you how to develop a JUnit test class for another class under test (CUT) in the JFreeChart code-base. The class under test is `org.jfree.data.DataUtilities` and you can find its API specification, in the Javadoc as shown in Figure 11. For developing tests for any class, you need to review the API specification of the class: its methods and their inputs and outputs, as shown below.

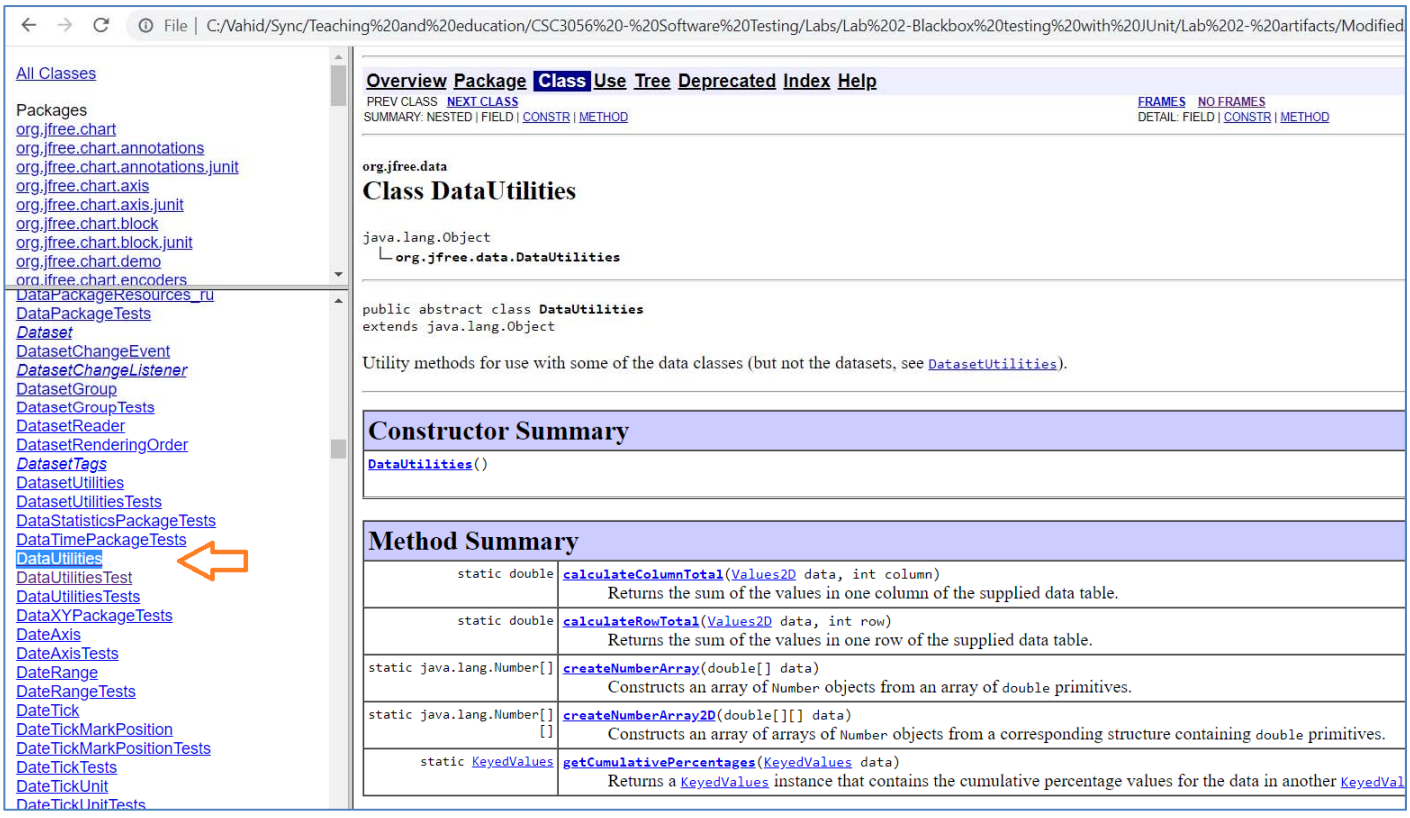


Figure 11 - API specification of the `org.jfree.data.DataUtilities` class

The steps below are almost identical as what we did for the Range class in the previous section.

31. In the package explorer, expand the *Referenced Libraries* list item to show all the archives that the project uses.
32. Expand the *jfreechart.jar* archive to expose all the packages that are contained in that archive.
33. Expand the `org.jfree.data` package within the archive to show all the .class files contained in that package.
34. Finally, expand the `DataUtilities.class` item to expose the class contained in that file, along with all the class' methods and fields contained in that class; as shown in Figure 12.
35. Right click on the `Range` class (has a green 'C' icon, denoting it is a class), and select *New -> JUnit Test Case*. Type *RangeTest* in the *Name* field and type '`.test`' in the *Package* field at the end of the default package to create a new package for the test cases. Ensuring that test classes are in a separate package makes it easier to keep the two apart. Also, ensure that the superclass is as specified in Figure 6. Also, make sure that you have checked the boxes for the method stubs specified in Figure 6, e.g., `setUp()`
36. Click *Finish*.

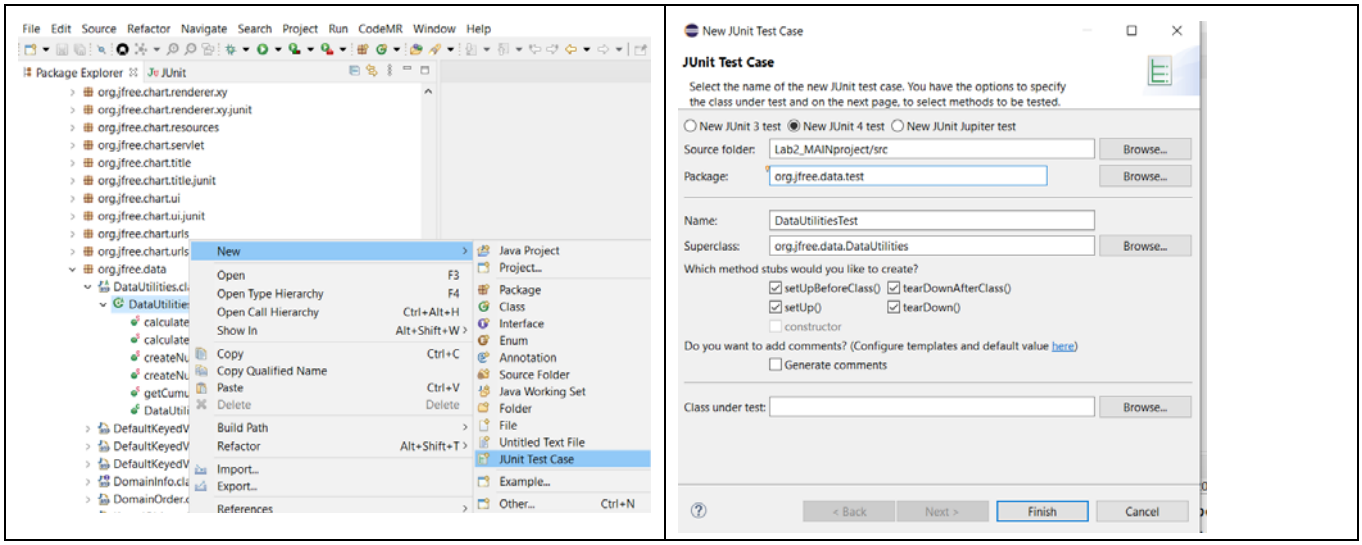


Figure 12 - Creating a new JUnit test case for the `org.jfree.data.DataUtilities` class

37. Write the test-code of `DataUtilitiesTest` as follows. WE will explain next some new coding concepts used in this test-code snippet.

```
package org.jfree.data.test;

import org.jfree.data.DataUtilities;
import org.jfree.data.DefaultKeyedValues2D;
import org.jfree.data.Values2D;
import junit.framework.TestCase;

public class DataUtilitiesTest extends TestCase
{
        private Values2D values2D;

        public void setUp()
        {
                DefaultKeyedValues2D testValues = new DefaultKeyedValues2D();
                values2D = testValues;
                testValues.addValue(1, 0, 0);
                testValues.addValue(4, 1, 0);
        }

        public void tearDown()
        {
                values2D = null;
        }

        public void testValidDataAndColumnColumnTotal()
        {
```

```
                assertEquals("Wrong sum returned. It should be 5.0",
                                5.0, DataUtilities.calculateColumnTotal(values2D, 0), 0.0000001d);
        }
}
```

38. By the way, it will be quite likely that you will need to resolve package/module imports using the IDE's QuickFix feature, as shown in Figure 13.

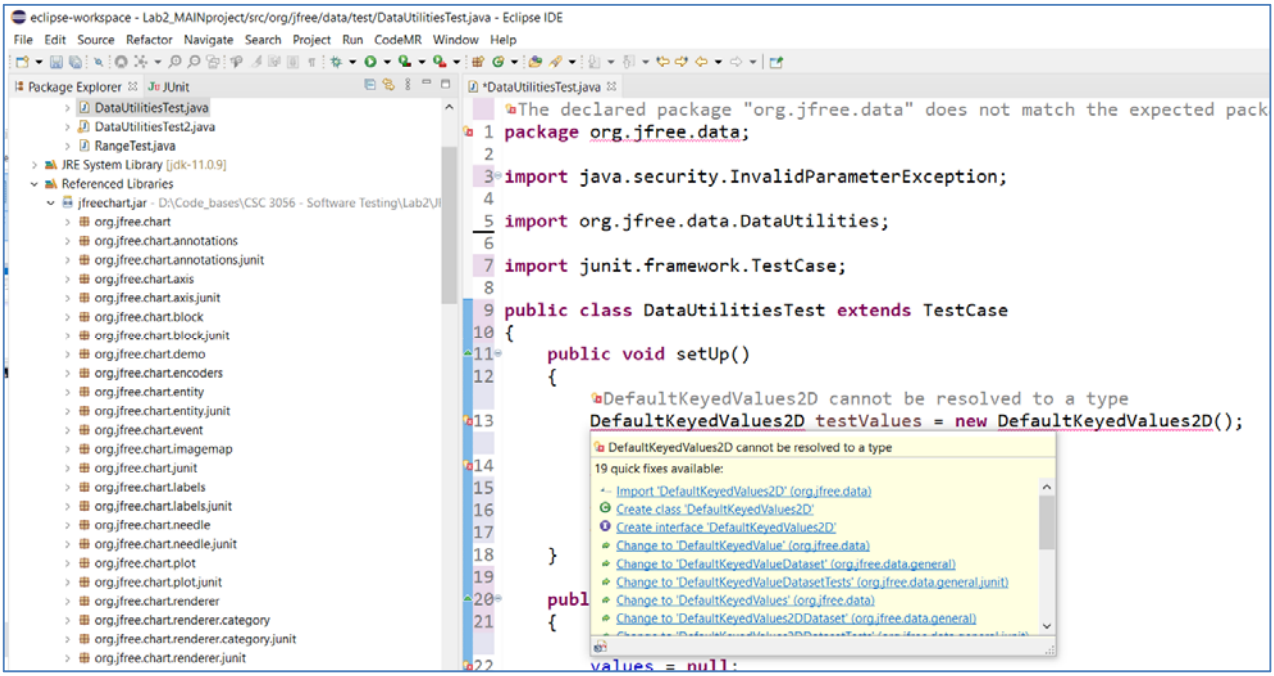


Figure 13- Resolving package/module imports using the IDE's QuickFix feature for the `DataUtilities` class

2.1.7 Using "implementing" classes for "interfaces" in Java

39. When developing JUnit test scripts for certain classes under test, we need to deal with the so-called "interfaces" in Java. First, read and learn the concept of Java "interfaces" in this URL: en.wikipedia.org/wiki/Interface_(Java)

40. Some methods in the class under test (CUT) `DataUtilities` use the interfaces `Values2D` and `KeyedValues`. In such cases, we need to find out and use the proper "implementing" classes of the interfaces, and that information is often available in Javadoc specification of the API (see Figure 14). You can use the following "implementing" classes for them: `DefaultKeyedValues2D` and `DefaultKeyedValues`, as shown in the Javadoc, shown in Figure 14.

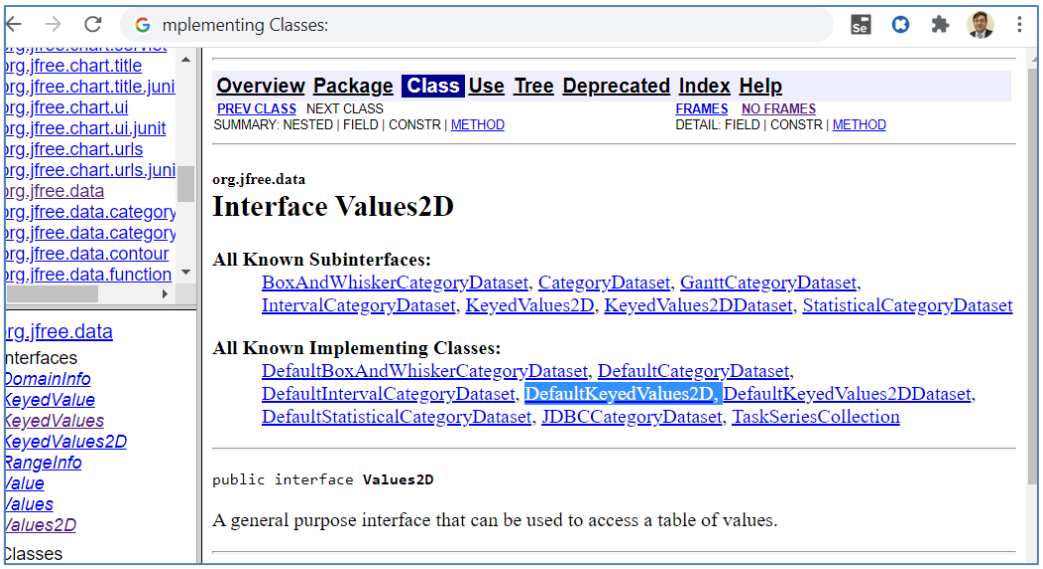


Figure 14 – "Implementing" classes of interfaces in Javadoc API specifications

41. Now look at the test-code of DataUtilitiesTest that we have provided for you above. In there, look at how the interface Values2D and class DefaultKeyedValues2D have been used. In fact, if you move the mouse over those two data types, you will see that Eclipse will tell you that they are an interface and a class, respectively; see the (i) and (c) icons in Figure 15.

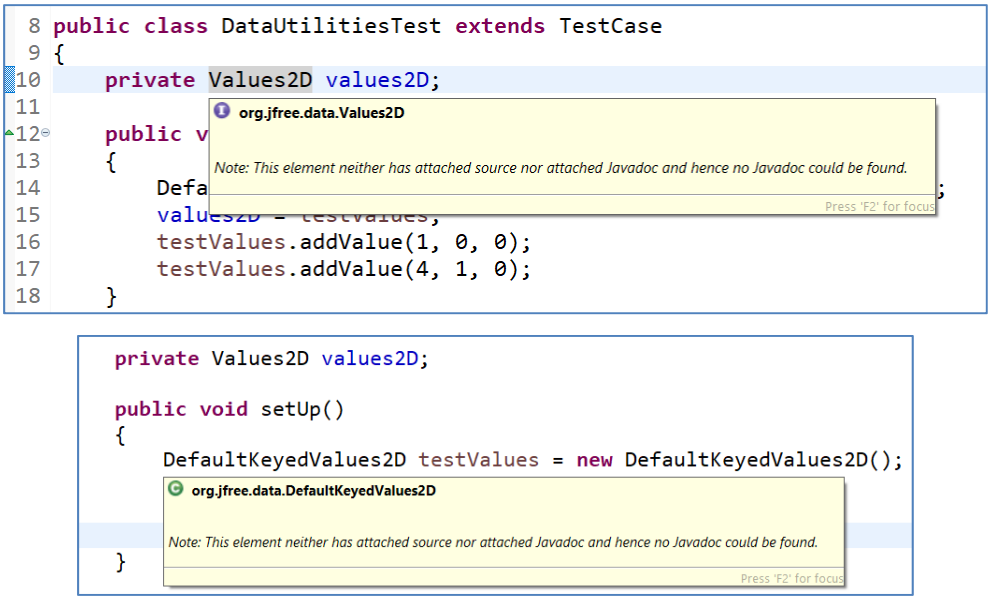


Figure 15 – Eclipse telling us the types of data types in our code

42. Since *Values2D* is an interface, its behavior "must be implemented by other classes", as per definition of interfaces: en.wikipedia.org/wiki/Interface_(Java). *DefaultKeyedValues2D* is one such class as per the API specification. Therefore, we declare an object of that class type and make that equal to the object of the interface type as you can see in the test-code: `values2D = testValues`. We then manipulate the object of class *DefaultKeyedValues2D* (named `testValues` in our code) afterwards, e.g.. call its methods. The "scope" of the `testValues` object is only in the setup method level ("scope" means the boundary in which a variable is defined, known and can be used). However, the "scope" of the object of the interface *Values2D* (named `values2D` in our code) is in the class level, as you can see in the test-code. In other test methods, we thus need to use the `values2D` object, e.g., as we are using it in the *testValidDataAndColumnColumnTotal* test method.

2.1.8 Asserting for expected "exceptions" in JUnit

An "exception" is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. Depending on what previous courses you have taken, you may have or have not learned about exceptions. See the links below to learn about exceptions:

- A short definition: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html
- A comprehensive page: https://en.wikipedia.org/wiki/Exception_handling

Exceptions are classified using a hierarchy in every programming language. In Java, a partial snapshot from the hierarchy of exceptions is as shown in Figure 16. Note: This is a UML class-diagram[1], also called a *meta-model*[2]. You do not have to memorize or learn of all of this diagram in our course, but it is good to see it in case you will later hear about exceptions in your future software engineering career.

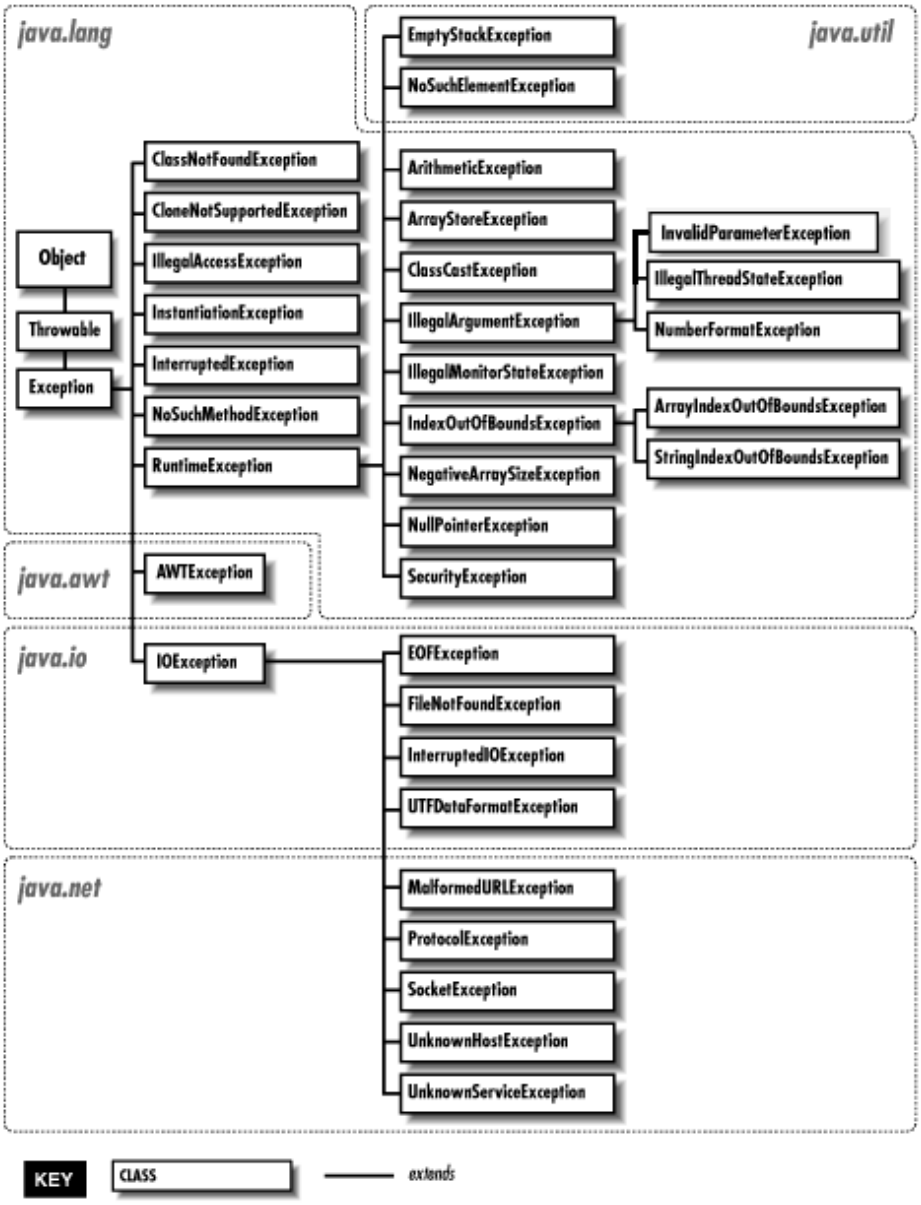


Figure 16 – Hierarchy and types of exceptions in Java

[1] https://en.wikipedia.org/wiki/Class_diagram
[2] https://en.wikipedia.org/wiki/Metamodeling

43. When we design test cases "on paper" and then write the corresponding JUnit test cases, in some instances the expected outcomes are exceptions. For example, let's look at the API Javadoc of method *calculateColumnTotal* inside class `DataUtilities`, shown in Figure 17.

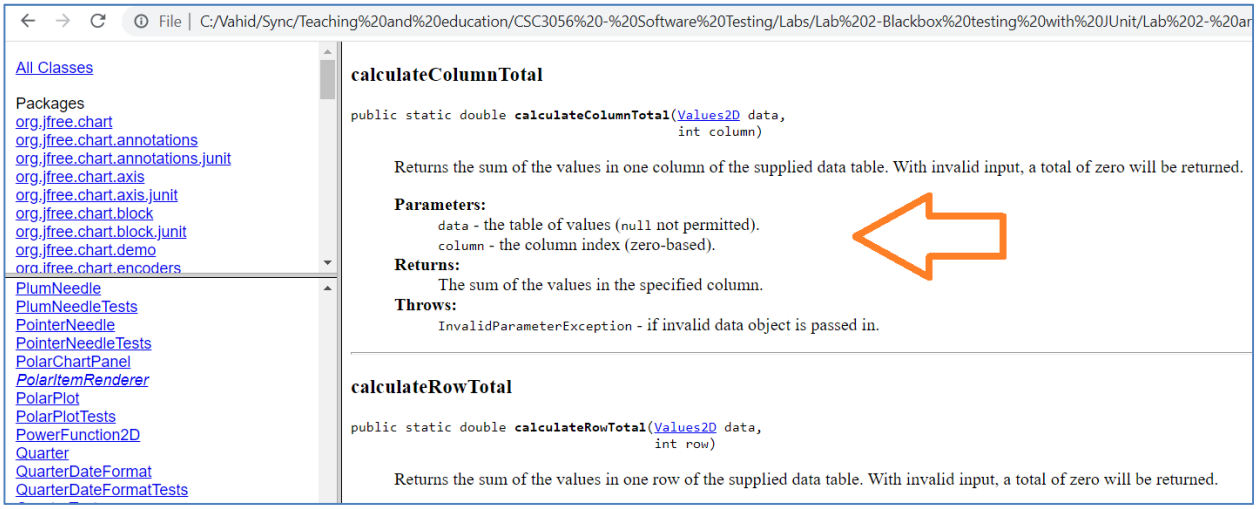


Figure 17 – API Javadoc of the method *calculateColumnTotal* inside class `DataUtilities`

44. As the API Javadoc mentions, if we pass in an invalid data object to the *calculateColumnTotal* method, it "should" return ("throw"[1]) an *InvalidParameterException*. Therefore, when we develop our JUnit test method, and let's say, pass a NULL object to the first parameter (data) of this method, our test case shall expect to see an exception, and that is considered a "pass" for the test case. If the *calculateColumnTotal* method does not return an exception, given a null data object, the test cases will be considered a fail. With all these explanations, we now provide the test case method for the above case, below. You need to carefully review the test code below to ensure you fully understand it. If you have issues understanding each line of this test code, talk to you friends and/or the instructor.

```
public void testNullDataColumnTotal()
{
        try
        {
                DataUtilities.calculateColumnTotal(null, 0);
                fail("No exception thrown-Expected outcome was: a thrown exception
                        of type: InvalidParameterException");
        }
        catch (Exception e)
        {
                assertTrue("Incorrect exception type thrown",
                        e.getClass().equals(InvalidParameterException.class));
        }
}
```

Figure 18 – Another test case for class `DataUtilities`

45. You may not have heard about try/catch blocks either. In that case, learn about it by doing a Google search.
46. Add the above test-code into to your test-code of `DataUtilitiesTest` that we helped you develop in the previous section. Note: If the compiler would show an error message on the *InvalidParameterException* class (saying that it is unknown), use the Eclipse' QuickFix feature to fix the import statement needed for that (we have discussed it in the previous sections of this lab document).
47. Run the test class `DataUtilitiesTest` and observe its outcome. You shall see the test outcome seen in Figure 19. *testValidDataAndColumnColumnTotal* shall pass. *testNullDataColumnTotal* shall fail. The latter test case mean that the SUT does not return what we had expected from it, as per its API Javadoc.

Note: The instructor has intentionally injected an undisclosed number defects in the SUT. Thus, a number of your test cases in this lab-work shall fail and of course, you should leave them as they are, and report the failures in your lab report. But you need to ensure that your test cases are designed properly (on "paper", before coding them) and then coded properly in Java/JUnit.

[1] https://rollbar.com/guides/java/how-to-throw-exceptions-in-java/

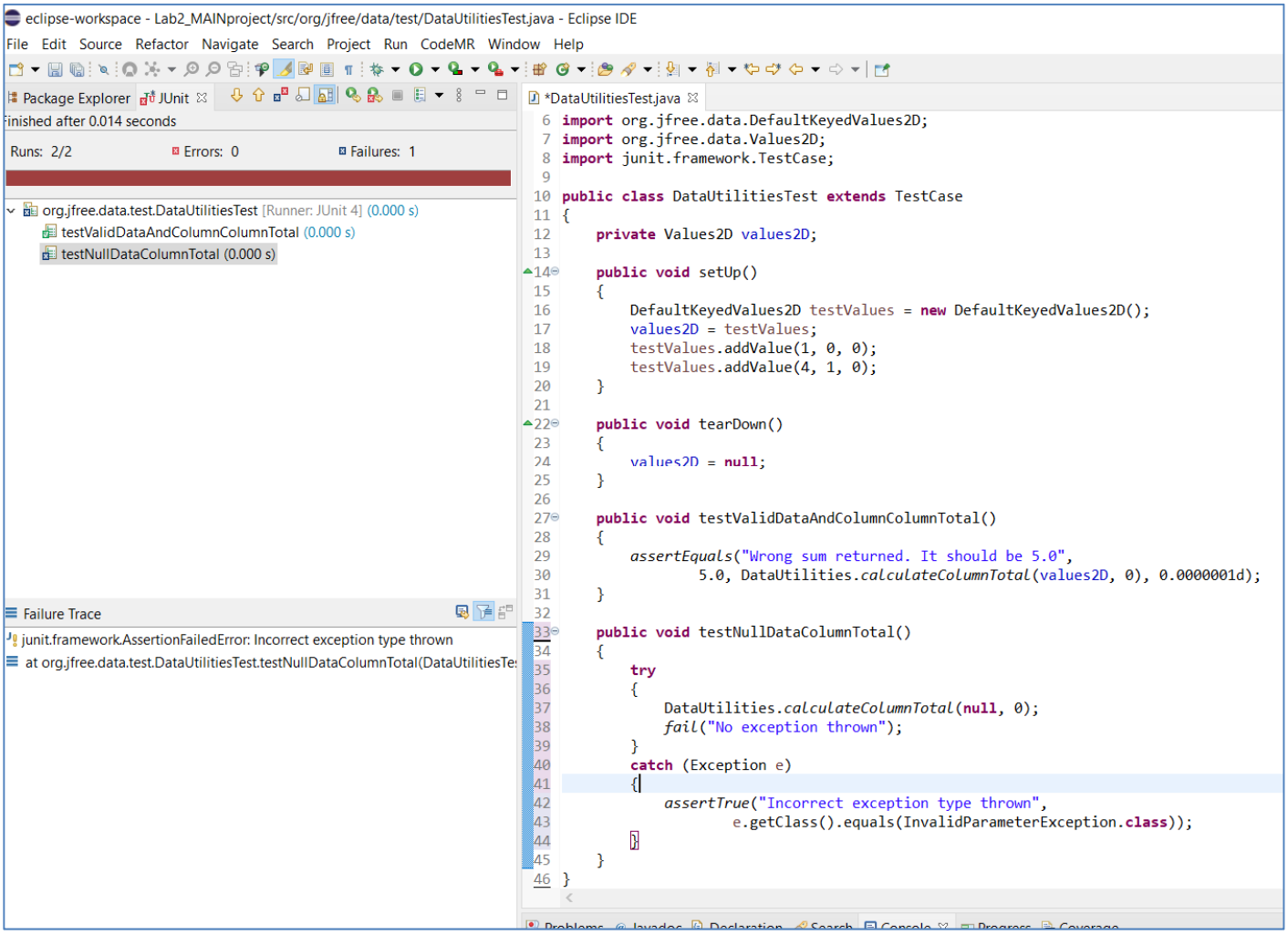


Figure 19 – Outcome of JUnit test execution of two test cases for class `DataUtilities`

More details and examples for asserting for expected exceptions in JUnit are provided in Appendix C.

2.1.9 Errors versus failures in JUnit test-case executions

48. When you execute a set of JUnit test-cases, you will see thee possible outcomes in the JUnit execution view of Eclipse:

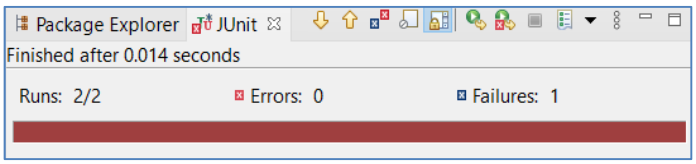


Figure 20 – Thee possible outcomes for test cases in the JUnit execution view

49. Failures are counted and shown in the JUnit execution view when any of your test cases fail – i.e. when the condition of any is evaluated to False.
50. On the other hand, Errors in the JUnit execution view correspond to issues (often defects) in your test-code itself. They are often unexpected errors that occur while trying to actually run the test suites (the set of all test cases in your JUnit code-base), such as improper exception handling, etc. If your test method or the unit under test throws an exception which does not get properly handled through your test-code and the Assertion framework in JUnit, it gets reported as an Error in the JUnit execution view. For example, a *NullPointer*, or a *ClassNotFound* exception will report an error, if they are not properly handled using the mechanism presented in the above example code snippet.

As a general rule, when your test-code is executed, you should NOT see any errors in the JUnit execution view. If you see any, you shall carefully inspect your test code, fix the issue rerun it and ensure number of errors=0, in the JUnit execution view.

51. **In summary, you need to ensure that your test cases are designed properly (on "paper", before coding them) and then coded properly in Java/JUnit.**

2.1.10 Designing test cases for methods with no parameters and developing test code for them

52. We have seen in the lectures and sections above many examples of how to design test cases for methods with one or more parameters and developing test code for them. But what about designing test cases for methods with no parameters? That is an interesting and important question!
53. Let's take an example from JFreeChart, the method *getLength()* inside class *Range*. Its API spec is as follows:

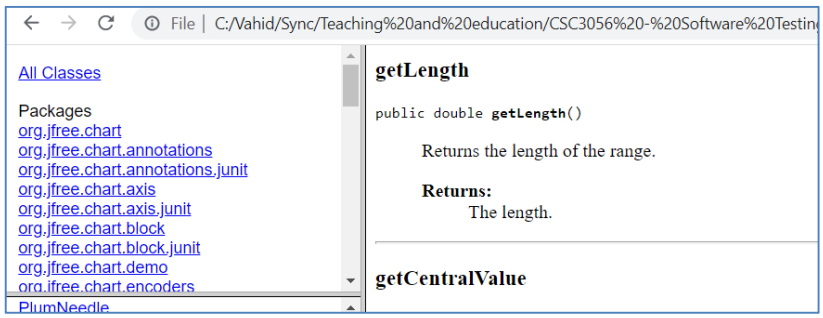


Figure 21 – Thee possible outcomes for test cases in the JUnit execution view

54. To apply equivalence-class partitioning to this method, we need to be creative and smart, since it does not have any input. However, we should note that it is a method of a class, and look at its business logic: it should returns the length of the range. Furthermore, it is easy to find out, from the class constructor, *Range(double lower, double upper)* in the API spec, that, the range is determined by two numbers, which would be the member variables (attributes) of the class: a lower double variable, and a upper double variable. These contexts can help us realize that we need to treat those member variables when testing the method *getLength()*. No, can you apply equivalence-class partitioning to this method? Think and if you cannot find out, read on…
55. From the definition of the equivalence-class partitioning technique, we need to partition the input space in which input data have the same effect on the method under test (e.g., the result in the same output). So we can find out that:
    - If both *lower* and *upper* variable of the object have the same value, *getLength()* should return the value of zero (0)
    - Else, *getLength()* should return the value of *upper-lower*
    - To be on the safe side, we can also test when both are *lower* and *upper* positive values, when both are negative values and also when one is +ve and one is -ve. Thus, in a minimalist test design manner, that will give us at least five test cases as follows:

TC	Inputs		Expected output
	lower	upper	
1	2	2	0
2	4	9	5
3	-99	-99	0
4	-11	-4	7
5	-5	3	8

    - The JUnit test code of the above test suite will be:

```
public void testGetLength() {
    Range r1 = new Range(2, 2);
    assertEquals("getLength: Did not return the expected output", 0.0, r1.getLength());

    Range r2 = new Range(4, 9);
    assertEquals("getLength: Did not return the expected output", 5.0, r2.getLength());
```

```
    Range r3 = new Range(-99, -99);
    assertEquals("getLength: Did not return the expected output", 0.0, r3.getLength());

    Range r4 = new Range(-11, -4);
    assertEquals("getLength: Did not return the expected output", 7.0, r4.getLength());

    Range r5 = new Range(-5, 3);
    assertEquals("getLength: Did not return the expected output", 8.0, r5.getLength());

}
```

2.2 DESIGN AND DEVELOPMENT OF UNIT TEST CASES

Now that you have finished the Familiarization section above, we now provide in this section the instructions for what should actually be done for your lab-work, and to be reported in you report.

This section is recommended to be performed as a group, however the work may be divided and completed individually, or you may wish to employ "peer programming". There is a huge number of online resources about peer programming, and you can learn from them using Google search[1] or YouTube video search.

2.2.1 Test requirements (classes to be tested)

56. In this section, you will be required to design and develop unit tests for the following classes, using their specifications (as mentioned in Javadocs). The two classes to be tested are the followings, which we have seen them to some extent in the Familiarization section above :

    - `org.jfree.data.Range` (in the package `org.jfree.data`): Has 15 methods in total
    - `org.jfree.data.DataUtilities` (in the package `org.jfree.data`): Has 5 methods in total

    Take a few minutes to browse and review the API specifications of each of these classes in Javadoc, and their list of methods.

The lab workload expectation: To keep your workload manageable, we would like you to design test cases for all 5 methods of `DataUtilities` class and only 5 out of 15 methods for `Range` (feel free to choose any random five of those methods). So in total you have 10 Java class methods to develop Unit tests for. Thus, the effort level that you will spend to design and develop the test cases will be reasonable (not too high!), but still you will near test automation in a very realistic setting (similar to industrial settings).

2.2.2 The first step: Write your unit testing plan

57. As with any testing to be done, to begin with, a plan must be first documented. Document this test plan, , as it should be included inside your lab report. For this test plan, you need to include the following components:

    - The black-box test technique(s) that you are using
    - How you will derive your test cases
    - How you will organize your JUnit test suites (based on Appendix C)

    Note: We do not require from you a very extensive test plan for this lab, but when you will work in industry as a test engineer, a typical test plan could include the following sections. You are welcome to Google "test plan examples" and learn from the many example provided by test engineers online.

[1] www.google.com/search?q=peer+programming

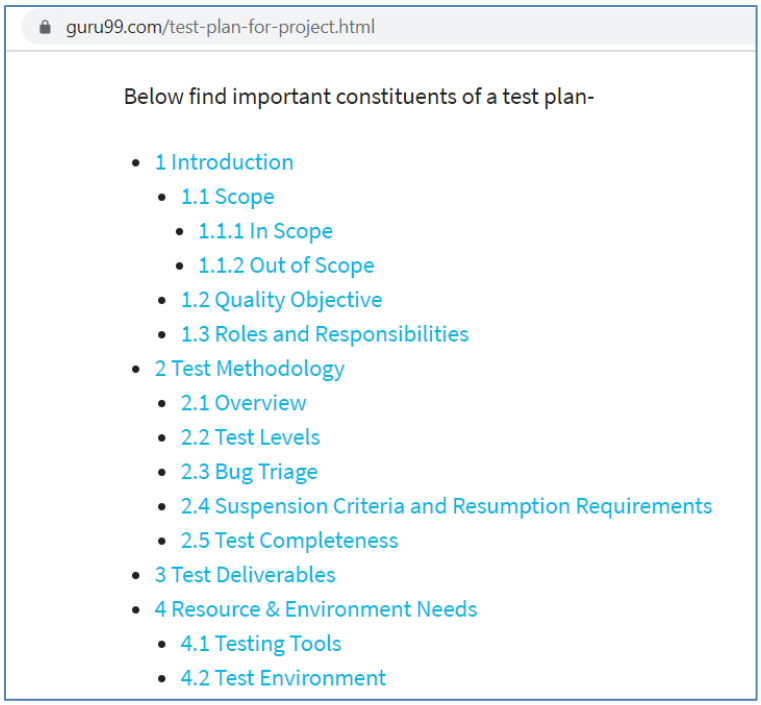


Figure 22 –Typical contents of a test plan document in software industry

2.2.3 Designing the test cases, BEFORE developing them in JUnit

58. After you documented your test plan, you should discuss and document in your report how you designed the test cases (recall the "test-case design" lectures from the class). Since you are given the requirements only, you should apply black-box test-case design techniques: equivalence classes partitioning, and boundary value analysis (BVA).
59. When using the equivalence-classing method, you need to generate the "strong" equivalence-classes test suites, and clearly they will automatically contain the "weak" equivalence-classes test suites. When applying these techniques, make sure to explicitly follow and document in your report all the steps discussed in the class, e.g., first derive the domain for each input variable, then the equivalence classes of each input variable, etc. just like how we learned and practiced with them in the lectures. You should ensure that the requirements are adequately tested. Reminder: You need to include the details of how you design test cases in your report.
60. Carry out your test plan and design your test-cases on paper (your lab report) first.

- **Important: Only after you have designed your test cases on paper, you should developing them in JUnit.**

2.2.4 Developing your test code based on your test-case design

61. The next step is to develop your test code in the JUnit framework based on the list of test cases you have designed on paper, in the previous section. Note: Each test method should include one test-case only. For example: `testPositiveValuesForMethodX()` and `testNegativeValuesForMethodX()`, instead of a single `testMethodX()`. This will help to keep test cases consistent, and make analysis of test case impact simpler later on.
62. Before writing your test methods, you should first read the test-code architecture and also naming conventions provided in Appendix C, and follow them in your test code development.
63. If, in your group, you have divided the test cases between team members, then upon completion of the tests, review each other's test methods, looking for any inconsistencies or defects in the tests themselves.
64. While you develop your test cases, in intervals, execute your partial test suite on *JFreeChart v1.0.zip*. Note that we have intentionally injected various (random) defects in the SUT! And thus a number of your tests should fail. Therefore, to write your test methods, you need to follow the specifications, not the actual results returned by the methods (functions).

65. Note that we do not prescribe any upper limit on the number of your test cases for any of the classes under test. Depending on how you will use the black-box test-case design techniques (such as equivalence classes, and boundary value analysis), you would end up with different number of test cases for each of the classes under test. But of course, you should use the black-box test-case design techniques in a logical and proper manner, just like how we learned and practiced with them in the lectures.

    - Note: If your test suite has much more or much less than the number of tests that you would normally result from "proper" application of those black-box test-case design techniques, your report will lose some marks. Note: A "few" tests more or less will not lose marks!

3 SUMMARY OF THE LAB AND LEARNING OUTCOMES

Upon completion of this lab, students should have a reasonable understanding of unit testing based on requirements using the JUnit framework. Note that unit testing and JUnit are very comprehensive and it takes quite a lot of time to be an expert in them. So do not expect to be JUnit experts just by completing this lab. If you would like to have a career path in this industry-hot topic, you will need to study this popular framework in more detail and perform more exercises to be skillful.

The unit testing knowledge you gained in this lab can be scaled up to much larger systems, and can be very useful in industry.

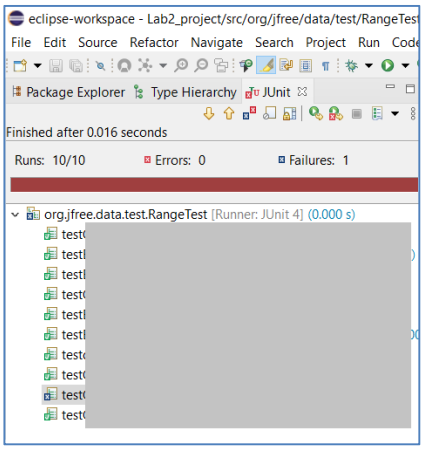
4 DELIVERABLES AND GRADING

Only one submission per group, by either of the two students.

4.1 LAB REPORT (50%)

To be consistent, please use the template Word file "Lab 2 Report Template.doc" provided online.

- Please keep the section names as provided in the lab template file.
- You should upload each lab's report in Word DOC or DOCX format in the course management system.
- A portion of the grade for the lab report will be allocated to organization and clarity.

Marking scheme	
Your unit testing plan	10%
Design of all your unit test-cases • **Make sure to include, in your report: HOW you used the black-box test-case design techniques (such as equivalence classes, and boundary value analysis)** • When applying these techniques, make sure to explicitly follow the steps discussed in the class, e.g., first derive the domain for each input variable, then the equivalence classes, etc. just like how we learned and practiced with them in the lectures. Note: You should not include any test code in this section, but only the design of the test cases using the above methods, before coding them.	15%
Output of test suite execution: Include a screenshot of test suite execution in JUnit, such as:  **(Note: This is just an example. We are not providing the number of test cases for you.)**	10%

List of failed test cases, and the possible defects based on that information	
A discussion on how the team work/effort was divided and managed. Any lessons learned from your teamwork on this lab?	5%
Difficulties encountered, challenges overcome, and lessons learned from performing the lab	5%
Comments/feedback on the lab itself	5%

4.2 JUNIT TEST SUITE (50%)

Your test suite Java files should be ZIPPED as a SINGLE ZIP file and submitted along with the lab report in the course website. The grading criteria for JUnit test suite are as follows:

Marking scheme	
Adherence to requirements (do they test only the requirements, no more, no less?)	10%
Completeness (are there any obvious requirements which have not been tested? Have equivalence classes, and boundary value analysis been followed carefully? Is your unit test code clearly match the test-case design section in your lab report?)	20%
Correctness (do the tests actually test what they are intended to test?)	10%
Readability and understandability (Are the JUnit test methods easy to follow, through commenting or style, etc.?)	10%

Important note: Please store the JUnit test suite you have developed in this lab in a known location. You will use these test suites in Lab 3.

ACKNOWLEDGEMENTS

This lab is part of a software-testing laboratory courseware available under a Creative Commons license. The laboratory courseware has been used by 50+ testing educators world-wide. More details can be found in the courseware's website:

sites.google.com/view/software-testing-labs/

The courseware has been made a reality thanks to many people, including Michael Godwin, Christian Wiederseiner, Yuri Shewchuk, Riley Kotchorek and Negar Koochakzadeh for their helps in developing and testing these exercises.

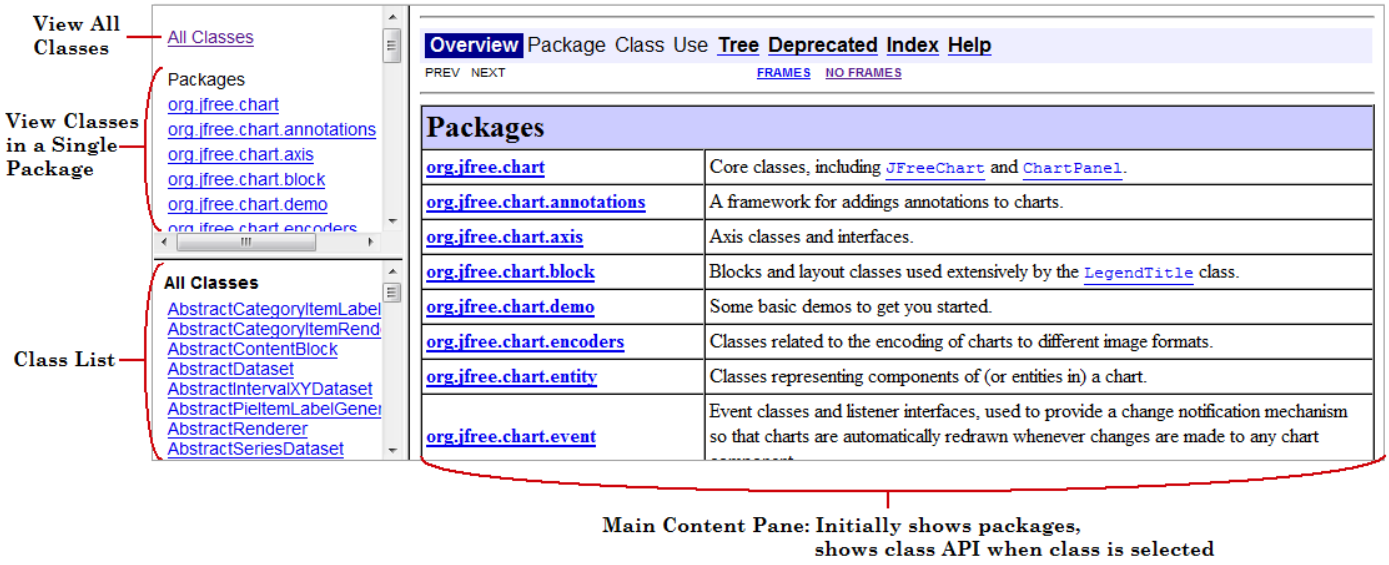
APPENDIX A – ASSERTIONS AVAILABLE IN JUNIT

The following is a list of assertions available for use in JUnit test cases. Note that assertions with 'message' arguments are equivalent to those without the message argument, but will display that message when the test fails on that assertion. These can be useful for debugging purposes. For more information, see:

https://junit.org/junit4/javadoc/latest/org/junit/Assert.html

- `assertEquals(expected, actual)`
- `assertEquals(message, expected, actual)`
- `assertEquals(expected, actual, delta)` - used on doubles or floats, where delta is the allowable difference in precision
- `assertEquals(message, expected, actual, delta)` - used on doubles or floats, where delta is the allowable 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()` - Fails a test with no message.
- `fail(message)` - Fails a test with the given message.
- `failNotEquals(message, expected, actual)`
- `failNotSame(message, expected, actual)`
- `failSame(message)`

APPENDIX B – JAVADOC EXAMPLE



APPENDIX C – CONVENTIONS FOR TEST-CODE ARCHITECTURE AND NAMING IN JUNIT

Just like we use certain naming conventions in our working code for deciding how to name classes, interfaces, and variables, we also use conventions when naming our tests and test classes.

TEST-CODE ARCHITECTURE

Recall from your other courses that software and code architecture are important topics in software engineering. Test-code architecture is a topic which is important in our course context. Essentially, a recommended good practice to modularize unit test code with respect to the the production code (source code under test) is visualized in Figure 23. Note that the numbers (indices) are hypothetical and of course you should use the exact method names instead of the numbers. It is important to follow this architecture in your test code development.

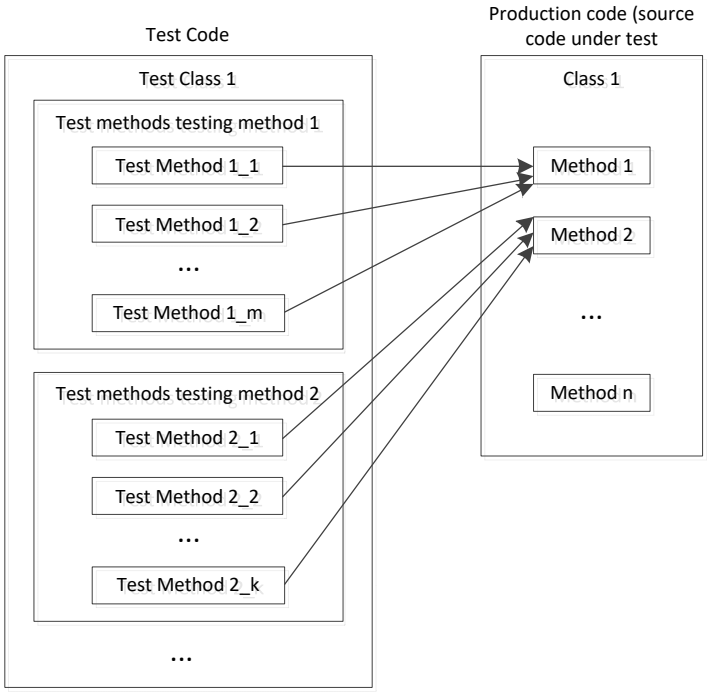


Figure 23 - A good practice to modularize unit test code architecture with respect to the the production code (source code under test)

TEST CLASSES

Some rules are much the same as the Java conventions that you are used to, such as capitalization. Also, each test class should be in its own file and you should avoid defining other classes in the same file.

Test classes should be named by the class they are testing with the appended word 'Test' after the class name. For example, if you were to write unit test code for classes `Range` and `DataUtilities`, you would develop two Java class files each named: `RangeTest.java` and `DataUtilitiesTest.java` and you would develop the classes `RangeTest` and `DataUtilitiesTest` respectively in each of those two files.

TEST METHODS

If you are familiar with JUnit 3.x, you may have become familiar with the naming convention of prefixing the name of each test with the word 'test'. This was a necessity in JUnit 3.x in order for the compiler to understand that the method was indeed a test method. Starting in JUnit 4, this is no longer a necessity. Instead we must include the "annotation" `@Test` before such methods. JUnit 4 testers are divided when it comes to following the rule of prefixing name of each test with the word 'test'. For example you may see some tests declared as:

```
    @Test public void testMyMethod() ...
```

while others would name the method as follows:

```
    @Test public void myMethod() ...
```

You may choose to follow either convention, but be consistent, please choose only one.

Each test should include only one test and the purpose of the test should be immediately clear by reading the method name. Remember that we will not be calling these methods in our code so there is no foul against giving test methods what would otherwise be considered exceptionally long names. For example, the following are some examples of clearly named test methods:

```
    void myMethodThrowsExceptionWhenInputIsNull() {}
    void myMethodReturnsNullWhenInputIsEmpty() {}
    void myMethodSucceedsWhenGivenExpectedInput() {}
```

and here are some poorly named examples:

```
    void myMethod() {}
    void myMethodTest() {}
    void myMethodTest2() {}
    void myMethodSucceeds() {}
    void myMethodFails() {}
    void myMethodThrowsException() {}
```

Naming your methods in this way will help you to understand what the purpose was later on and will also help you understand what has happened when your test suite fails.

ASSERTING FOR EXPECTED EXCEPTIONS IN JUNIT

If the method you are testing can throw an exception, you should test that it does so under the right circumstances. There are several ways to handle this situation.

Alternative 1:

One way (alternative) to do this would be to surround the test code in a try/catch block and include a `fail();` statement as the last line of the code in the try section. An example is shown below:

```
1  package net.codejava;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.fail;
5
6  import org.junit.Test;
7
8  public class UserTest {
9      @Test
10     public void testUserNameTooShort() {
11
12         try {
13             User user = new User();
14             user.setName("Jo");
15
16             fail();
17         } catch (IllegalArgumentException ex) {
18             assertEquals("Username is too short", ex.getMessage());
19         }
20     }
21 }
```

As you can see, we use the *fail()* statement at the end of the catch block so if the code doesn't throw any exception, the test fails. And we catch the expected exception by the catch clause, in which we use *assertEquals()* methods to assert the exception message. You can use this structure to test any exceptions.

And there are many resources on this in the internet, see:

https://www.google.com/search?q=junit+handle+exception

Alternative 2:

The second alternative is to use a specific annotation of JUnit. JUnit 4 provides an annotation that will in essence do this for you. Simply include '`(expected=Exception.class)`' after the `@Test` annotation as follows:

```
@Test (expected=NullPointerException.class)
void myMethodThrowsExceptionWhenInputIsNull() {}
```

In the above example, if a `NullPointerException` is thrown at any time during the test, the test will pass, otherwise it will fail.

APPENDIX D – UNDERSTANDING THE STEPS OF A TEST METHOD IN XUNIT

There are normally four steps to every test; the boundaries between each of these steps should be made immediately clear when reading test code.

SETUP

Note: to learn more about how to implement setup and tear methods, see Appendix C.

Setup for a test usually happens in one or more of the following places:

- a setup method that is run once before any tests are executed (`@BeforeClass` annotation)
- setup method that is run every time before each test is executed (`@Before` annotation)
- setup code inside of your test method

Use the setup to construct the scenario for the test which usually includes constructing data objects but may also include running methods or inputting data. Using the suite-wide initialization (`@BeforeClass`) is not very common, it is best to initialize any data you need for each test in your other setup methods unless you can be absolutely sure that that data being used cannot be manipulated by the tests and therefore cause your subsequent tests to be unreliable. It is however best to abstract as much as possible from each test into your setup methods in order to make your tests more consistent and easier to read.

Although it is possible to write test cases with no setup at all, in most tests you will write, the bulk of your code for each will be found here.

EXERCISE

Most, if not all, of the unit tests you will code for this lab will include an 'exercise' of only one line as you will be testing one function at a time. The exercise is simply the execution of the code of interest for the particular test.

VERIFY

Much of the time, similar to the previous step, all of your verification for one test is executed in one statement. This is usually done through assert statements `assertEquals()`, `assertTrue()`, or `assertNull()` for example. Much of the time, it is also possible to include your exercise and verification in the same statement. However this is usually avoided in favour of clarity.

TEAR-DOWN

Including tear-down code in Java tests is much less common than in other languages. Normally tear-down code is simply used for cleaning up after a test has executed so you will need to implement this step to close a file if one is opened during your setup for example.

More reading:

We encourage you to read and learn more about the above test phases in Google:

google.com/search?q=xunit+setup+pattern+teardown