SENG 438- Software Testing, Reliability, and Quality

Assignment 2

Automated Requirements-Based API Unit Testing using JUnit

Instructor: Dr. Behrouz Far (far@ucalgary.ca)

Teaching Assistants: Fatemeh Sharifi (fatemeh.sharifi1@ucalgary.ca)

Munima Jahan (munima.jahan@ucalgary.ca)
Fauziya Shaikh (fauziya.shaikh@ucalgary.ca)

Department of Electrical and Computer Engineering

University of Calgary

Due Date: February 15, 2019; 23:59

TABLE OF CONTENTS

1 INTRODUCTION

1.1 OBJECTIVES

The main objective of this assignment 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.
- To utilize and work with mock objects in test-code development.

1.2 THIS LAB IS A GROUP WORK

All the tasks of this lab should be completed in groups of four/five (as you formed in lab 1). The report will also be completed as a group.

1.3 OVERVIEW

This lab can be divided into three main sections. Similar to lab 1, the first section is familiarization. During the familiarization stage, students will be shown how to set up a JUnit test project in Eclipse using JUnit, create a unit test, and how to navigate Javadocs.

After familiarization, the second section of the lab is unit test generation based on the requirements specified in Javadocs. In this stage, students will develop unit test suites for two classes. These test suites will be comprised of unit tests which have been generated according to the requirements.

During test code development in this lab, students will also develop and work with mock objects. In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways. In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:

- supplies non-deterministic results (e.g. the current time or the current temperature)
- has states that are difficult to create or reproduce (e.g. a network error)
- is slow (e.g. connects to database, which would have to be initialized before the test)
- does not yet exist or may change behavior
- would have to include information and methods exclusively for testing purposes (and not for its actual task).

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.

1.4 TESTING TOOLS

The main testing tool for this lab is JUnit [4]. JUnit is a popular free unit testing tool and framework for Java. For more information on JUnit, see class notes and JUnit reference website: http://www.junit.org.

Another tool which is being used for the purpose of testing is Javadoc [2]. While Javadoc is not exclusively or explicitly a testing tool, in the context of this lab it will be used as the format to store requirements specification to use to derive test suites.

Javadoc allows developers to create Application Programming Interface (API) documentation within source code together with code itself. Creating 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 http://java.sun.com/j2se/javadoc.

1.5 SYSTEM UNDER TEST

The system to be tested for this lab is JFreeChart [3]. JFreeChart is an open source Java framework for chart calculation, creation and display. This framework supports many different (graphical) chart types, including pie charts, bar charts, line charts, histograms, and several other chart types. To get started with the JFreeChart system, download the "JFreeChart v1.0.zip" file from D2L and extract the entire archive to a known location. 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 actual releases of JFreeChart. The versions have been 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 other 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. A snapshot of four different types of charts drawn using JFreeChart is shown in Figure 1.

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.

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.

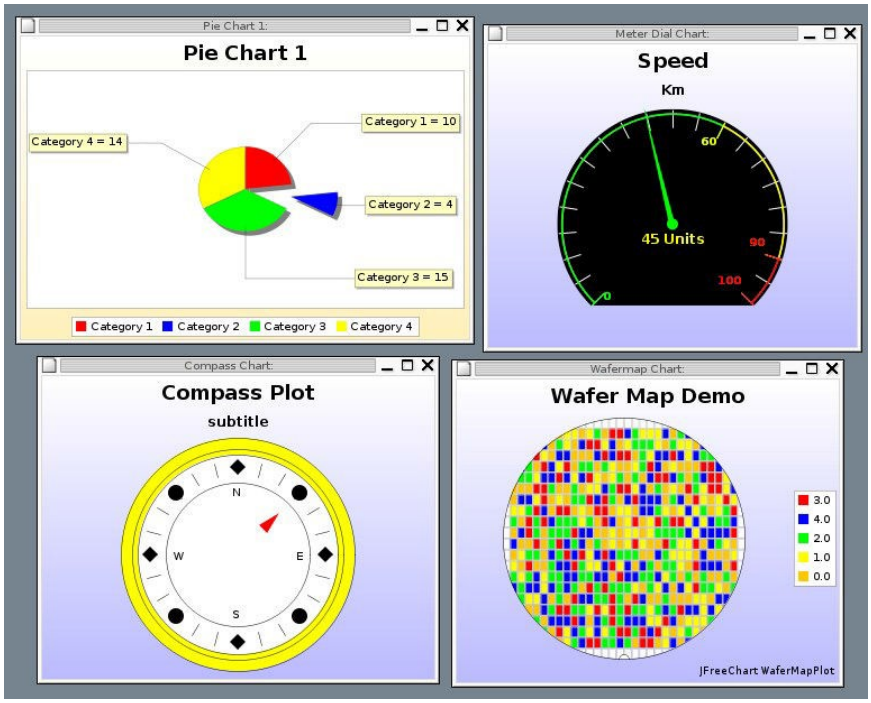

Figure 1 - A snapshot of four different types of charts drawn using JFreeChart.

2 INSTRUCTIONS

This section details the instructions for executing the lab. All sections of this lab should be completed as a group as specified earlier.

2.1 FAMILIARIZATION

All students of each group should perform this section of the lab together on a single computer. Ensure that everyone understands the concepts in this section before moving on to the rest of the lab.

1. If you haven't done so already, download and extract the *JFreeChart v1.0.zip* file from D2L (in Assignment2 file under the Labs folder).

2.1.1 Create an Eclipse Project

2. Open Eclipse.

   Note: All the figures shown are from Eclipse Java 2018-12 downloaded and installed from [1]

3. Open the *New Project* dialog by selecting the *File -> New -> Project…*

4. Under the folder Java, ensure that *Java Project* is selected and click *Next*.

5. The dialog should now be prompting for the project name. Enter *JFreeChart* in the *Project Name* field, and then click *Next*.

2.1.2 Add the Necessary Java Libraries

6. The *Java Settings* dialog should now be displayed. This dialog has four tabs along the top: *Source*, *Projects*, *Libraries*, and *Order and Export*. Move to the *Libraries* tab, and click the *Add External JARs (or Libraries)…* button.

7. Select the *jfreechart.jar* file from the known location and click *Open*. Click *Add External Libraries…* again, this time add all the .jar files from the *lib* and *lib/jMock* directory where you have unzipped the *JFreeChart v1.0.zip* file. The Java Settings dialog should now look like Figure 2, below. (This document and the provided libraries are based on jMock. However, you may decide to use other mocking frameworks such as Mockito.)

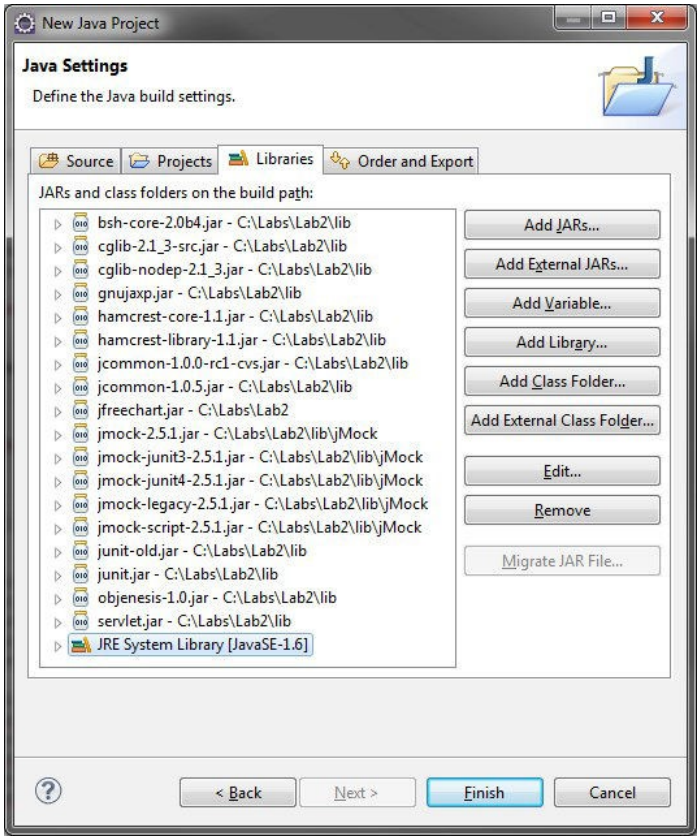

Figure 2 - The Java Settings dialog after adding required archives

8. Click *Finish*. The project (SUT) is now set up and ready for testing. To run the demo classes, 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).

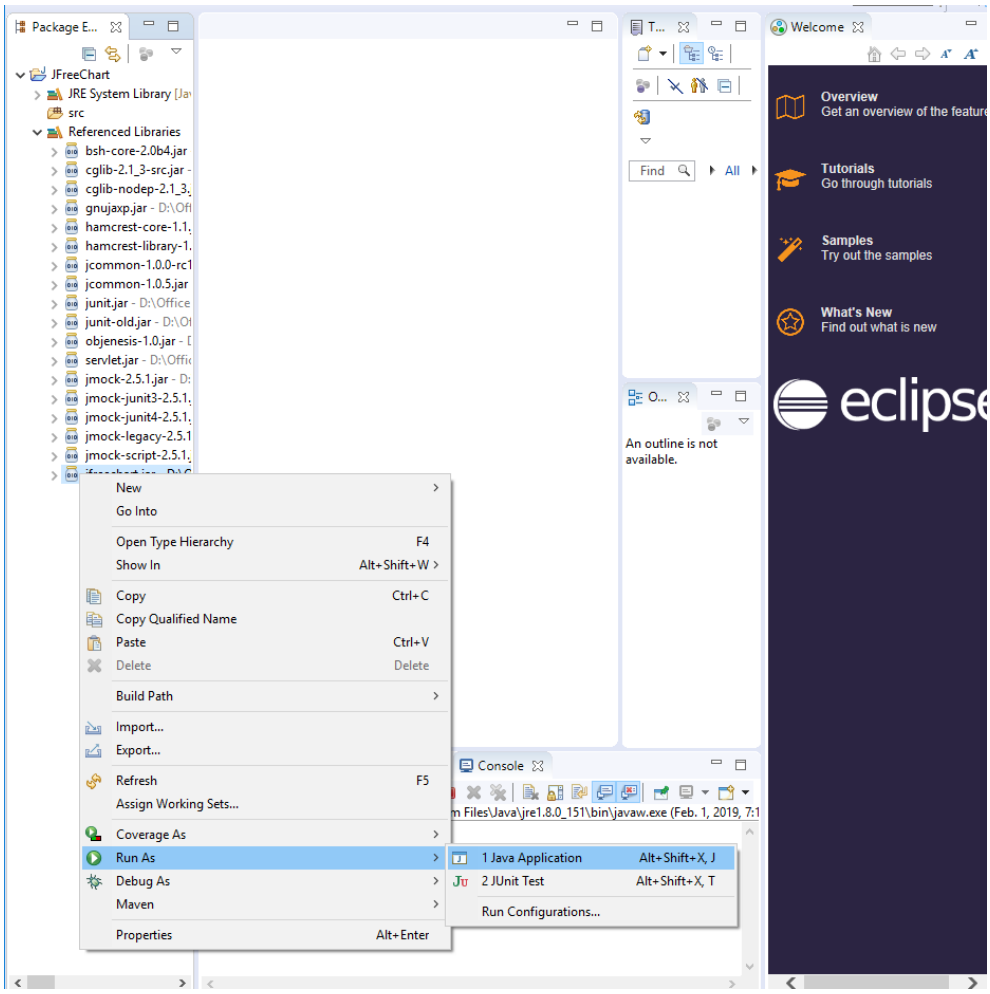

Figure 3 - Running JFreeChart

9. In the *Select Java Application* dialog, select any of the four demo applications (e.g., *TimeSeriesChartDemo1*), and click *OK* as shown in Figure 4.

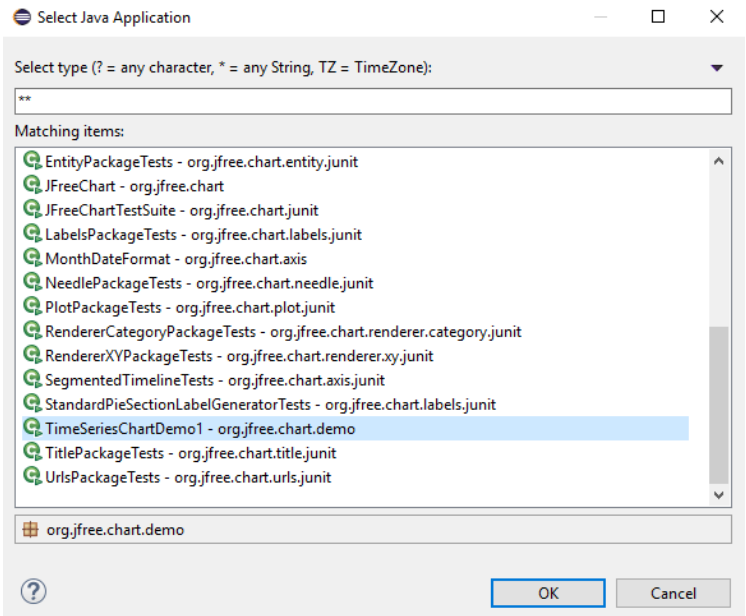

Figure 4 - The Select Java Application dialog

2.1.3 Create a Simple JUnit Test

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

10. In the package explorer, expand the *Referenced Libraries* list item to show all the archives that the project uses.
11. Expand the jfreechart.jar archive to expose all the packages that are contained in that archive.
12. Expand the `org.jfree.data` package within the archive to show all the .class files contained in that package.
13. 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.
14. click on the `Range` class (has a green 'C' icon, denoting it is a class), and then from the eclipse menu 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.

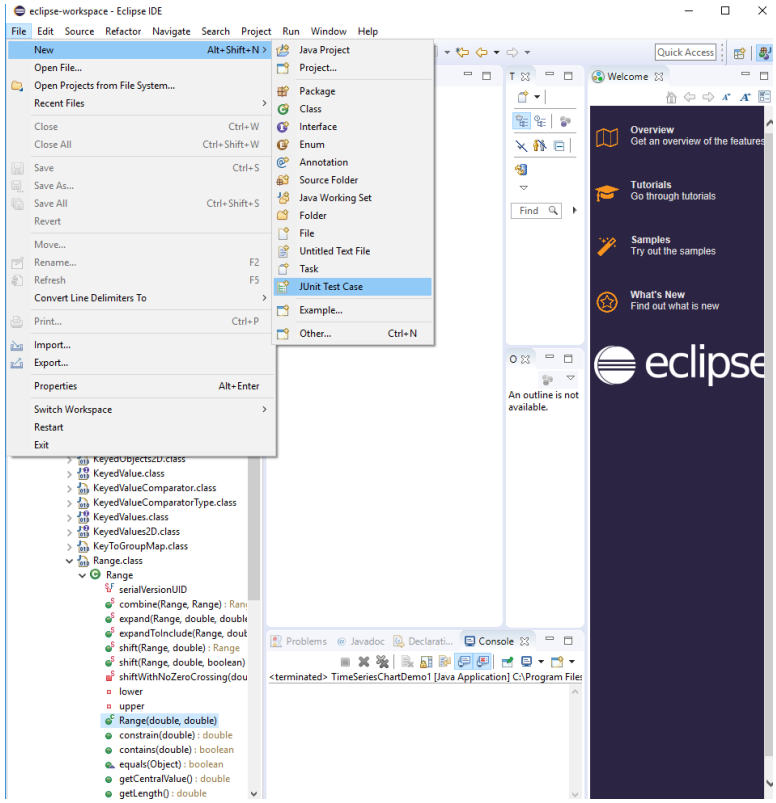

Figure 5 – Selecting JUnit Test Case

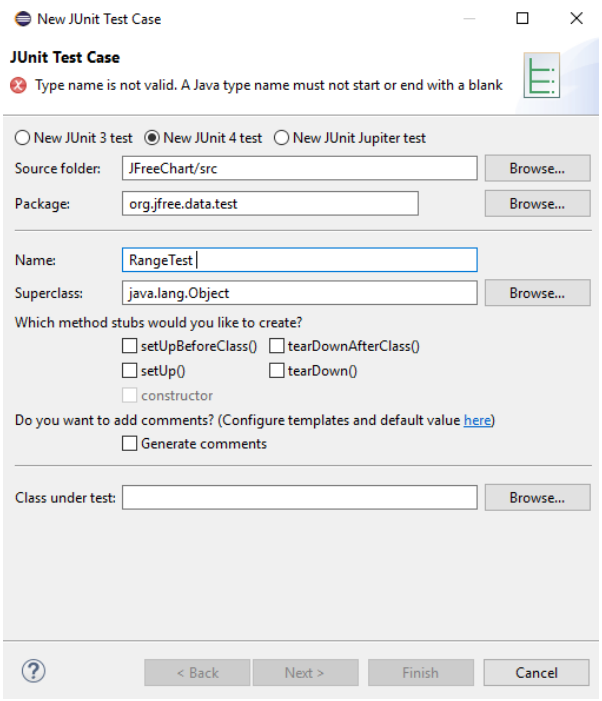

Figure 6 - The New JUnit Test Case dialog

15. Click *Finish*.

16. As a practice, write a simple test case for the `getCentralValue()` method. See Figure 7.

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

import static org.junit.Assert.*;
import org.jfree.data.Range;
import org.junit.*;

public class RangeTest {
 private Range exampleRange;
 @BeforeClass  public static void setUpBeforeClass()
throws Exception {
 }


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


 @Test
 public void centralValueShouldBeZero() {
  assertEquals("The central value of -1 and 1 should be 0",
                    0, exampleRange.getCentralValue(), .000000001d);
 }
 @After
 public void tearDown() throws Exception {
 }
  @AfterClass
 public static void tearDownAfterClass() throws Exception {
 }

 }
```

Figure 7 – The `RangeTest` test class after addition of a simple test case

17. 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*.

18. 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 similar to Figure 8 below.

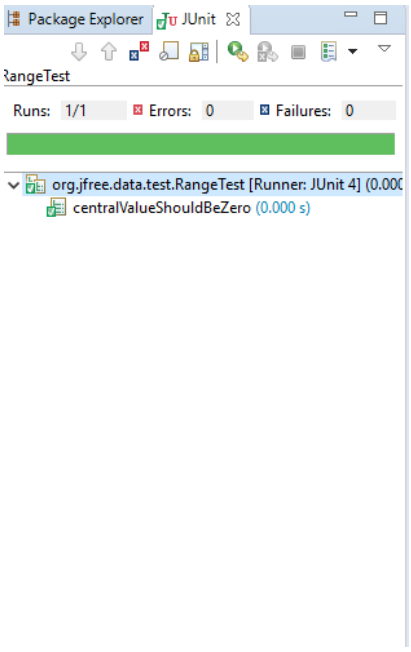

Figure 8 - JUnit view showing passed test

2.1.4 Navigate Javadoc API Specifications

The test generation section of this lab (Section 2.2) will require you to generate unit tests for a number of classes based on specifications (requirements) contained in the Javadocs for those classes. If you're already familiar with Javadoc, feel free to skip this section and continue from Section 2.2.

19. Unzip the *JFreeChart–ModifiedJavadoc.zip* file and open the file *index.html*. This is the Javadoc for (a slightly modified version of) JFreeChart. Note the location of different elements in the Javadoc as shown in Appendix A.

    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.

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

21. In the class list, click on the `ColorBar`. The main content pane now shows the API specifications of the `ColorBar`. 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.

22. 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.

2.2 Development of Unit Test Code

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 to help ensure all requirements are tested. In either ways, all the group members should be able to answer the questions from all section (even the sections they did not write) in demo session.

2.2.1 Test Requirements (Classes to be tested)

23. In this section, you will be required to create unit tests for several classes, testing them against their specifications.

The two classes to be tested are

- `org.jfree.data.Range` (in the package `org.jfree.data`): Has 15 methods.
- `org.jfree.data.DataUtilities` (in the package `org.jfree.data`): Has 5 methods. Take your time to browse the API specifications of each of these classes in Javadoc.

2.2.2 Using Mocking Objects in Unit Test Code

24. Note that some methods in `DataUtilities` use the interfaces `Values2D` and `KeyedValues`. Although there may be other ways, in order to test these methods for this lab, you should utilize Mocking to test `DataUtilities`. Because the methods take in interfaces as parameters, you will not know how the inherited classes may function. Mocking allows us to return any values or throw any exceptions we want, when we want. Even so, you may find drawbacks to this approach; you should discuss these in your report.

    To get you started, include the following example (that follows jMock notation) in your `DataUtilities` test code: Note that you can use any mocking framework, but the example given here are in jMock.

```
 @Test
 public void calculateColumnTotalForTwoValues() {
// setup
  Mockery mockingContext = new Mockery();
  final Values2D values = mockingContext.mock(Values2D.class);
mockingContext.checking(new Expectations() {
   {
    one(values).getRowCount();
will(returnValue(2));
one(values).getValue(0, 0);
will(returnValue(7.5));
one(values).getValue(1, 0);
    will(returnValue(2.5));
   }
  });
  // exercise    double result =
DataUtilities.calculateColumnTotal(values, 0);
  // verify
  assertEquals(result, 10.0, .000000001d);
  // tear-down: NONE in this test method
}
```

    Note that a better implementation would include the `Mockery` and `Values2D` objects being initialized in the `setUp()` method of your test class.

2.2.3 Test Plan and Test-case Design (The design component of the assignment)

25. As with any testing to be done, to begin with, a plan must be first created. Document this test plan (free format), as it will be included with your lab report.

26. After you documented your test plan, you should discuss in your report how you are designing 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 such equivalence classes, boundary value analysis, etc.. 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. You should ensure that the requirements are adequately tested.

27. Carry out your test plan and create your test-cases on paper (your lab report) first. To keep your workload manageable, we would like you to create test cases for all 5 methods of `org.jfree.data.DataUtilities` and choose 5 out of 15 methods for `org.jfree.data.Range`.

2.2.4 Write your Test Code based on your Test-case Design

28. The next step is to code your test code in the JUnit framework based on the list of test cases you have designed on paper. 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. For writing your test methods, please use the example discussed earlier in Section 2.1.3.
29. Before writing your test methods, you should first read the class notes on JUnit about testing code standards and patterns and also naming conventions and follow them in your test code development.
30. If you have divided the tests and completed them individually, then upon completion of the tests, review each other's tests, looking for any inconsistencies or defects in the tests themselves.
31. Execute the test suite you have created on JFreeChart v1.0.zip. Note that the classes have random defects in them intentionally, and thus several of your tests should fail. Therefore, to write your test methods, you need to follow the specifications, not the actual results.

3 SUMMARY

Upon completion of this lab, students should have a reasonable understanding of unit testing based on unit 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

4.1 Demo (20%)

The objectives for the demo are a) Preparing you for technical presentations, b) an early assessment of your work to give you a second chance to submit a high quality report, and c) making sure everybody in the team contributes evenly.

It is mandatory for all team members to attend the demo session and explain the TAs in the lab what they have done for this assignment. For this particular assignment, Lab4 is the demo day. You are expected to almost finish the assignment by the lab hour. All the team members should attend the lab. The TAs will go through the groups and each group member must demonstrate examples of developed tests for each method that they designed (and bugs they found) – at least two tests per student, which one uses mocking. They should also explain their strategy for designing the tests.

NOTE1: Student who miss the demo session or are unable to demo what is detailed above are considered as less-contributors and may lose up to the entire assignment 2's mark.

NOTE2: You still have time to further improve your test suite, after the demo session and before the deadline.

NOTE3: The TAs will ask questions and all the group members should be able to answer the questions individually,

NOTE4: All the students should demo all the parts including mocking even if they did not write the code for that part

4.2 LAB REPORT (20%)

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

If desired, feel free to rename the sections, as long as the headings are still descriptive and accurate.

A portion of the grade for the lab report will be allocated to organization and clarity.

Marking scheme	
A detailed description of the testing strategy for unit testing and your test-case design approach, i.e., how you used the black-box test-case design techniques equivalence classes, and boundary value analysis. Also list the name of the test cases you have designed and identify which one covers which parts of the strategy (which partition, which class, etc.) Include a discussion about what you feel are the benefits and drawbacks about using mocking.	15%

A discussion on how the team work/effort was divided and managed. Any lessons learned from your teamwork on this lab?	2%
Difficulties encountered, challenges overcome, and lessons learned from performing the lab	2%
Comments/feedback on the lab itself	1%

4.3 JUNIT TEST SUITE (60%)

Your zipped Eclipse project including any external library for mocking and all test suite Java files, as a SINGLE file, should be submitted along with the lab report in D2L. The JUnit test should be executable AS IS. No restricting, importing, etc. should be needed.

The grading criteria for JUnit test suite are as follows:

Marking scheme	

Clarity
//are they easy to follow, through commenting or style, etc.?
//The test cases should also have comments that shows which test follows which part of the test strategy. E.g.,
// this test covers a maximum value for variable X and normal values for variable Y and Z in method A(X,Y,Z)
	10%

Adherence to requirements //do they cover all the classes and partitions described in the test strategy?)	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 suite clearly match the test-case design section in your lab report?
	20%

Correctness
// do test run without error?
// do the tests actually test what they are intended to test?
	20%

Important note: Please store the JUnit test suite you have developed in this lab in a known location. It will be re-used in the next Labs .

5 REFERENCES

[1] "Eclipse.org," Internet: http://www.eclipse.org

[2] "Javadoc," Internet: http://java.sun.com/j2se/javadoc/index.jsp

[3] "JFreeChart," Internet: http://www.jfree.org/jfreechart

[4] "JUnit," Internet: http://www.junit.org

6 APPENDIX A – JAVADOC EXAMPLE