**Lab 3-White-box testing (test-case design using code coverage)**

Developed and maintained by: Dr. Vahid Garousi

[vgarousi@gmail.com](mailto:vgarousi@gmail.com)

[www.vgarousi.com](http://www.vgarousi.com)

Part of the Open Software Testing Laboratory Courseware:

[sites.google.com/view/software-testing-labs](https://sites.google.com/view/software-testing-labs/)

**Table of Contents**

[Revision history of this document: 2](#_Toc154428252)

[1 Introduction 2](#_Toc154428253)

[1.1 Objectives 2](#_Toc154428254)

[1.2 This lab is a group work 2](#_Toc154428255)

[1.3 Viewing and browsing the lab-doc PDF file using its document map (outline) 2](#_Toc154428256)

[1.4 Testing tools 3](#_Toc154428257)

[1.4.1 A short review of code coverage tools 4](#_Toc154428258)

[1.4.2 Our choice of coverage tool: EclEmma 7](#_Toc154428259)

[1.5 System Under Test 8](#_Toc154428260)

[1.5.1 Purpose of the System Under Test (SUT) 8](#_Toc154428261)

[1.5.2 Usage of the system 8](#_Toc154428262)

[2 Familiarization 9](#_Toc154428263)

[2.1 Both students of each group shall have a working set up in their own machines (laptops, PC etc.) 9](#_Toc154428264)

[2.2 Project setup in the IDE 9](#_Toc154428265)

[2.2.1 Create an Eclipse project 9](#_Toc154428266)

[2.2.2 Import your test suite (two test class files) from Lab2 10](#_Toc154428267)

[2.2.3 We need to use the JUnit version 4, like the previous lab 13](#_Toc154428268)

[2.2.4 A short online video for how to set up the project 13](#_Toc154428269)

[2.3 Running the Lab2 test suite and measuring its code (test) coverage using the tool 13](#_Toc154428270)

[2.4 Measuring the coverage of the SUT code-base, not test-code 16](#_Toc154428271)

[2.5 Small-scale practice for developing new test cases to increase coverage 17](#_Toc154428272)

[2.6 Errors versus failures in JUnit test-case executions 19](#_Toc154428273)

[3 Instructions for the lab-work 19](#_Toc154428274)

[3.1 First, write your test plan for white-box unit testing 19](#_Toc154428275)

[3.2 Test-suite improvement to increase the coverage ratio 19](#_Toc154428276)

[3.3 Analyzing and discussing how your additional test cases (test methods), in lab3 compared to lab2, improved code coverage of methods under test in Range and DataUtilities 21](#_Toc154428277)

[3.3.1 Summary table of test-suite size from in lab3 compared to lab2 21](#_Toc154428278)

[3.3.2 Discussing details of design and development of additional test cases 22](#_Toc154428279)

[3.4 Conduct small-scale data-flow coverage analysis manually 22](#_Toc154428280)

[3.5 Conduct small-scale mutation analysis (testing) manually 23](#_Toc154428281)

[4 Summary 25](#_Toc154428282)

[5 Deliverables 25](#_Toc154428283)

[5.1 Lab Report 25](#_Toc154428284)

[5.2 JUnit test suite 25](#_Toc154428285)

[Acknowledgements 25](#_Toc154428286)

[References 25](#_Toc154428287)

[Appendix A – Example data-flow coverage calculated manually 26](#_Toc154428288)

May 9, 2022

# Revision history of this document:

|  |  |
| --- | --- |
| Summer 2008 | First version was developed by Dr. Vahid Garousi and his team at University of Calgary |
| September 2010-2017 | Various improvements were made |
| December 2020 | The lab document was updated to align with the latest version of Eclipse IDE. The coverage tool is now embedded into Eclipse. |
| Fall 2021 | Made various improvements using student comments |

# Introduction

This lab has a similar focus to the previous lab, as it is once again unit testing. Unit testing will be performed using JUnit [4] in Eclipse [2]. As with the previous lab, students will start by familiarizing themselves with the usage of the testing tools followed by implementation (enhancement) of the test suite.

The major difference between the testing being performed in this lab and the previous lab (#2) is that this lab shows the students a different technique in deciding what test cases to develop. We will practice with white-box testing in this lab, as we have learned it in the lectures. To develop the test cases in this lab, instead of basing it on the requirements of the code, students will design their test cases on the control and data flow of the code for the SUT, as we have learned it in the lectures.

## Objectives

The objectives of this lab are to introduce students to the concepts of determining the adequacy of a white-box test suite based on code coverage. In white-box testing, it is important to measure the adequacy of a test suite based on completeness defined by the portion of the code which is exercised. This definition can take several forms, including control-flow coverage criteria: statement (or node) coverage, branch (or edge) coverage, condition coverage, path coverage or data-flow coverage criteria.

After completing the lab, students will be able:

* To use code coverage tools to measure test adequacy and become aware of similar tools for other programming environments
* To understand some of the benefits and drawbacks of measuring test adequacy with code coverage tools
* To gain an understanding of how data-flow coverage works and be able to calculate it by hand

## This lab is a group work

All the tasks of this lab should be completed in groups of two students. The report should also be completed as a group. Only one lab submission (report and code-base) per group should be submitted, by one of the students.

## Viewing and browsing the lab-doc PDF file using its document map (outline)

We know that reviewing and browsing large documents is always more convenient when having a document “map” (outline). We have enabled this feature in the PDF file of the lab document. You need to download the PDF and open it using the Adobe software (as shown in Figure 1) – not opening the file in Canvas via your browser. You need to enable the “Bookmark” mode in the PDF reader software, as shown in Figure 1. Then, you can easily click on each section or sub-section and the PDF reader will jump to that section, enabling you to see the “big picture” of the document and also to conveniently browsing the document.

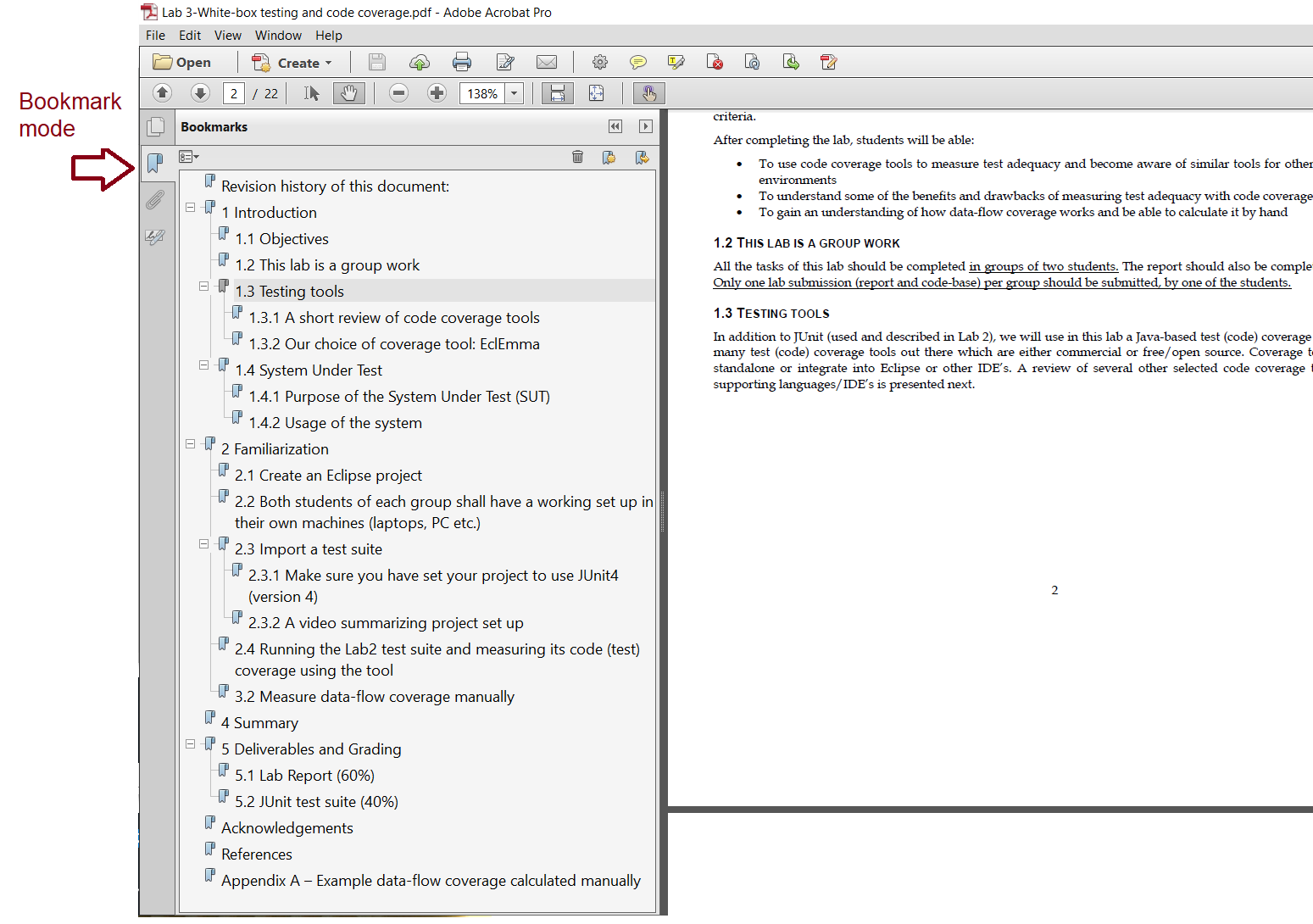


Figure 1 – Viewing the lab doc PDF file with the document map (outline)

## Testing tools

In addition to JUnit (used and described in Lab 2), we will use in this lab a Java-based test (code) coverage tool. There are many test (code) coverage tools out there which are either commercial or free/open source. Coverage tools either run standalone or integrate into Eclipse or other IDE’s. A review of several other selected code coverage tools and their supporting languages/IDE’s is presented next.

### A short review of code coverage tools

|  |  |
| --- | --- |
| Clover   * Java **c**ode **c**overage * Eclipse and IntelliJ plug-ins * Free for open source projects * www.atlassian.com/software/clover | http://www.eclemma.org/images/smallscreen.gif |
| EclEmma   * Java coverage * Eclipse plug-in * Open source * [www.eclemma.org](http://www.eclemma.org) | http://www.eclemma.org/images/screen.png |
| NCover   * C# code coverage tool * Self-contained executable or can be integrated with an IDE * [www.ncover.com](http://www.ncover.com) | http://www.sliver.com/dotnet/NCoverBrowser/NCoverBrowser.png |
| Coverlipse   * Java code coverage * Eclipse plug-in * Open source * Calculates both control-flow and data-flow coverage * No longer maintained, but works with earlier versions of Eclipse * [coverlipse.sourceforge.net](http://coverlipse.sourceforge.net/) | http://coverlipse.sourceforge.net/allusescoverageuses.png |
| dotCover   * C# code coverage * Visual Studio plug-in * [www.jetbrains.com/dotcover](http://www.jetbrains.com/dotcover) | http://sticklebackplastic.com/image.axd?picture=Windows-Live-Writer%2fxunitcontrib-reshar.4---dotCover-support_13E4%2fimage_2.png |
| Chrome browser: Coverage tab in the Chrome DevTools module  <https://developers.google.com/web/tools/chrome-devtools/coverage> |  |
| Coco tool   * C/C++/C# coverage * Available as a plug-in for most IDEs or stand-alone application * [www.froglogic.com/coco/](http://www.froglogic.com/coco/) | pictures/manual.tmp006.png |

### Our choice of coverage tool: EclEmma

In the case of earlier versions of Eclipse (in year around 2017), code coverage was not a “built-in” feature of the Eclipse Java IDE. But in recent versions of Eclipse, the EclEmma code coverage tool has been included “built-in” inside Eclipse. To check whether your Eclipse installation includes the EclEmma coverage plug-in, inside Eclipse, go to *HelpàAbout àInstallation Details*, menu. You should see EclEmma as shown in Figure 2. If, for whatever reason, your recent Eclipse installation does not include the EclEmma coverage plug-in, you can easily install EclEmma into Eclipse via: [www.eclemma.org](http://www.eclemma.org).

EclEmma is a popular tool and is based on the *JaCoCo* code coverage library ([www.jacoco.org/jacoco/](http://www.jacoco.org/jacoco/)). You can see various tutorials about EclEmma in YouTube by searching for “EclEmma”.

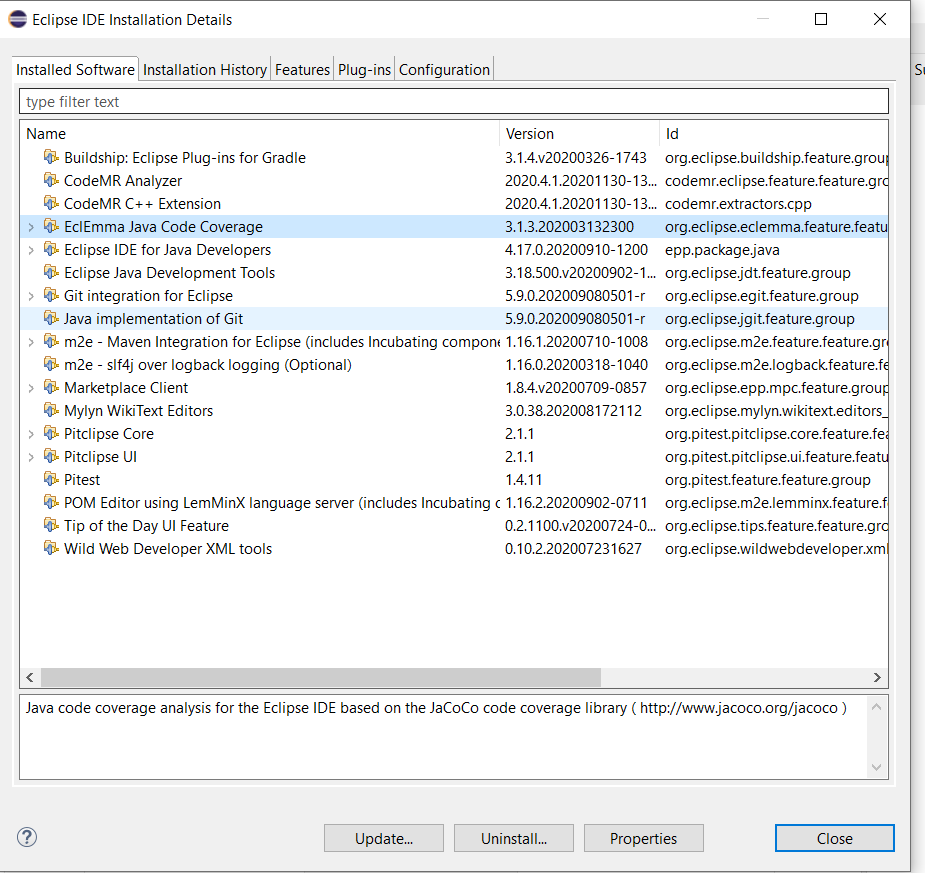


Figure 2 – Seeing the list of installed software and plug-in’s in Eclipse and ensuring the installation of the EclEmma coverage plug-in

## System Under Test

The system to be tested for this lab is JFreeChart [3], the same SUT used in Lab #2. 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.

### Purpose of the System Under Test (SUT)

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 developers as an open source off-the-shelf framework.

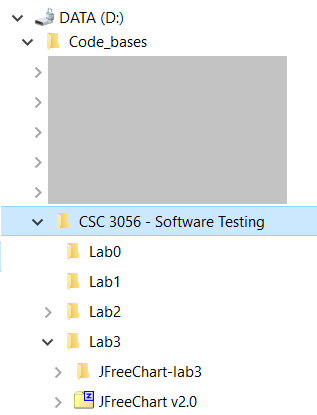
### Usage of the system

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.

# 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. The TAs and the instructor might ask each student questions related to any part of the lab for the purpose of students assessment.

1. To get started with the JFreeChart system, download the “*JFreeChart v3.0 for Lab 3.zip*” file from Lab 3 artifacts. Extract the entire archive to a known location. We recommend that you structure your folder structure like the following; having a separate folder for each lab in this course:



More information on how to get started with these files will be provided in the familiarization stage. Note that the versions of JFreeChart distributed for this lab do not correspond with actual releases of JFreeChart, rather versions in which we have made a few small modifications for the purposes of this lab.

Also note that since we will do white-box testing, test-case design from SUT’s source-code, we have provided to you the full source-code of JFreeChart this time. In lab 2, only the compiled executable version of JFreeChart was provided to you.

## Both students of each group shall have a working set up in their own machines (laptops, PC etc.)

Although this is a group work, both students of each group shall have a working set up in their own machines (laptops, PC etc.), since all students shall learn the course concepts individually and also as a group.

## Project setup in the IDE

### Create an Eclipse project

1. Open Eclipse Java IDE.
2. Open the *New Project* dialog by selecting the *File -> New -> Project*…
3. Ensure that *Java Project* is selected and click *Next*.
4. The dialog should now be prompting for the project name. Enter *Lab3\_JFreeChart* in the *Project Name* field.
5. Check the Create project from existing source radio button, and click *Browse…* to select the directory that the JFreeChart distribution was extracted to. The *New Java Project* dialog should now look like Figure 3 below (you need to put the path into which you have unzipped the JFreeChart source files). Click *Finish*.

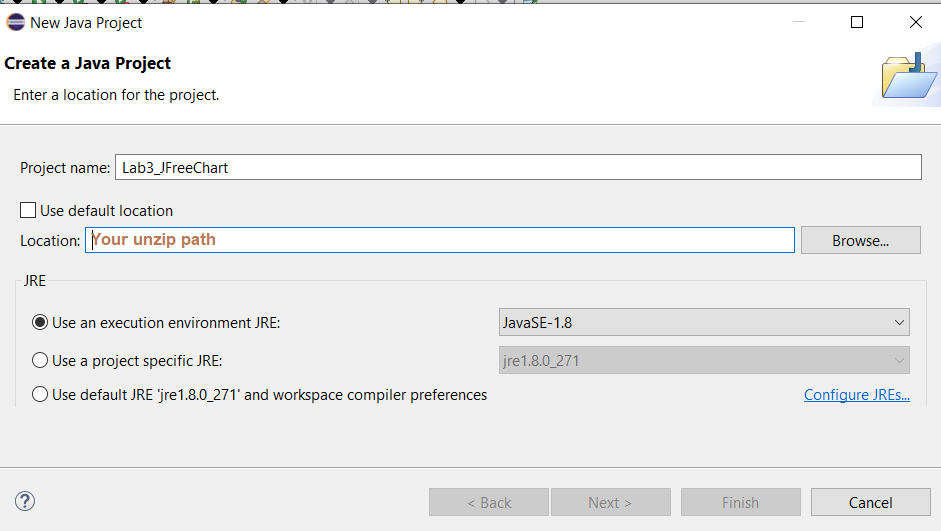


Figure 3 - New Java Project dialog with name and source path filled in

1. The project (SUT) is now set up and ready for testing. Verify that all the sources and reference libraries are shown as in Figure 4 below.

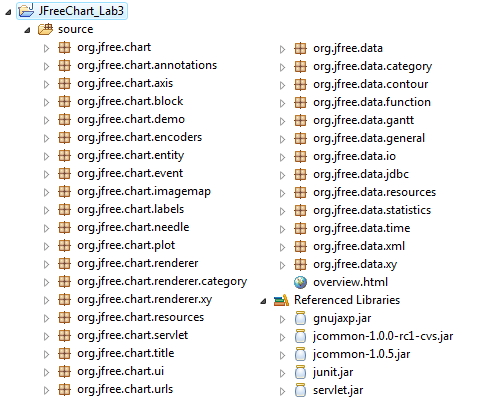
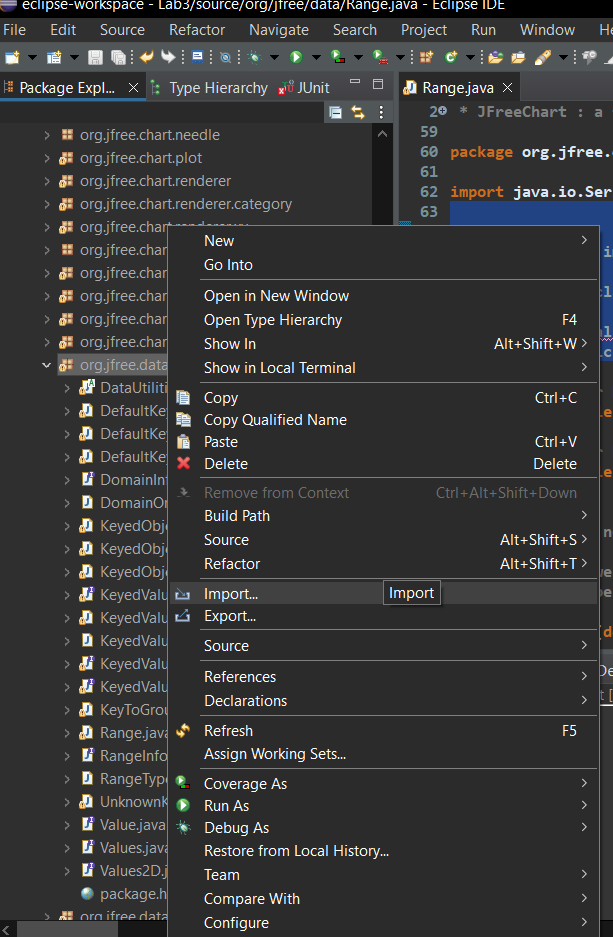


Figure 4 - Packages and archives that should be included in the newly-created project

### Import your test suite (two test class files) from Lab2

For the purpose of learning test coverage measurement and improvement in practice, **the test suite that you designed and developed in Lab 2** will be used.

1. Right click on the org.jfree.data package in the *Package Explorer*. Select *Import…*. As shown below:



1. In the *Import* dialog, select the *File System* option (in the *General* category) and click *Next*.
2. In the new panel on the *Import* dialog, click on the *Browse…* button to choose the directory you import your files from, then navigate to the directory from your previous lab (Lab2) containing your test files: RangeTest.java and DataUtilitesTest.java. Click *OK*.
3. Check your DataUtilities and Range test classes as shown in Figure 5 below. Then click *Finish*.

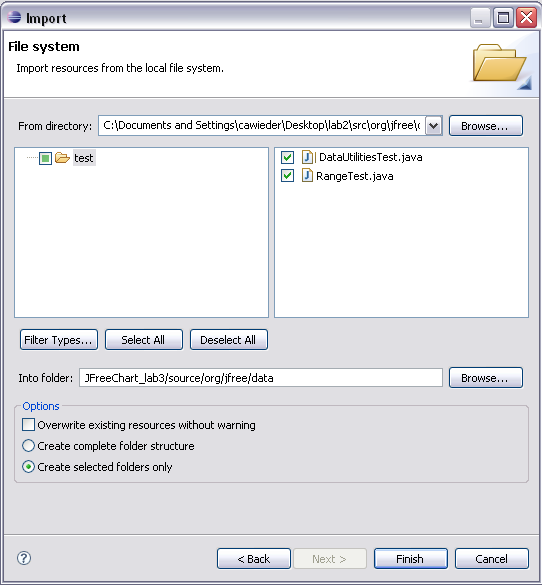


Figure 5 - Import dialog with Lab 2 test classes selected

The test classes selected are now included in the org.jfree.data package in the new project. BUT: please make sure this is the case and that the package name in your test class is defined as org.jfree.data.

Note: If you see an error in the newly-added test classes (RangeTest.java and DataUtilitesTest.java) about *@test* labels being unrecognizable, you can easily fix it by adding JUnit4 library to your project, as follows:

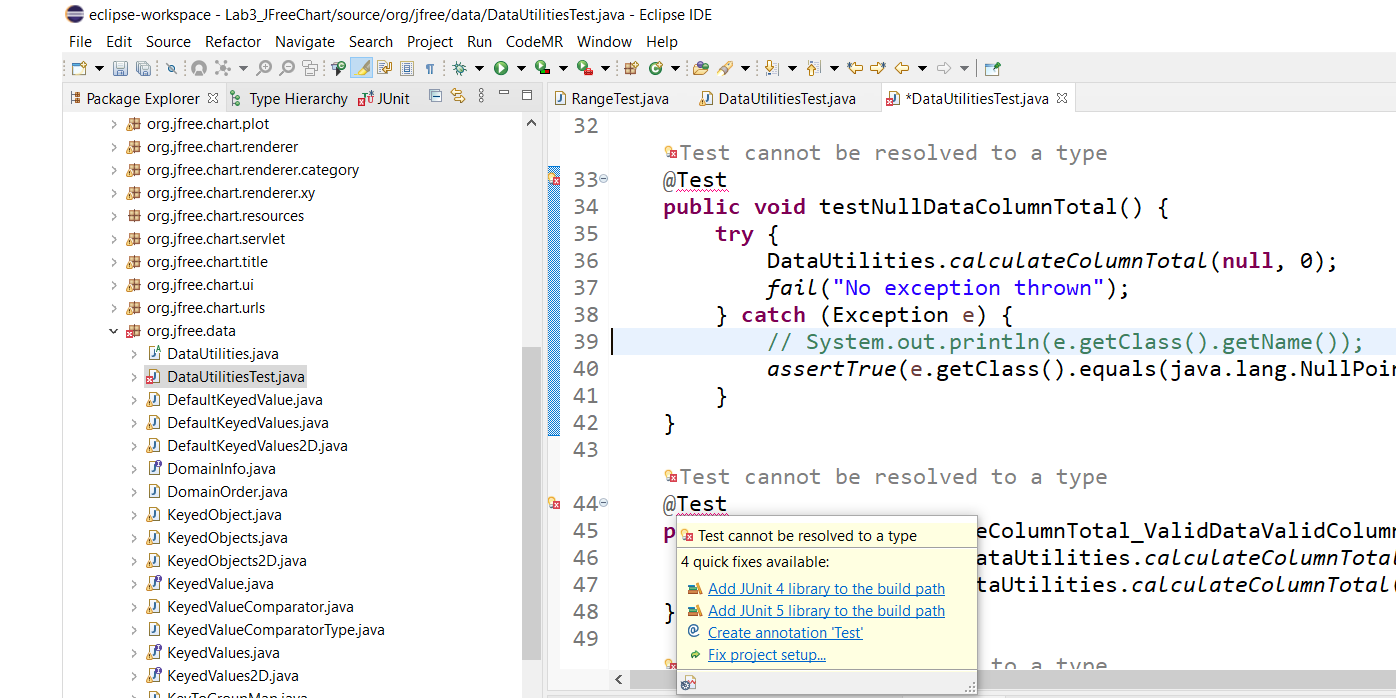


Figure 6 – Adding JUnit4 library to your project, to fix the *@test* errors

You may also get some minor errors in the top of the two new test files, about the package locations, as shown below. You can fix them by removing the “.test” from the package location text in top of the code, since the current package of these two files are now: org.jfree.data .

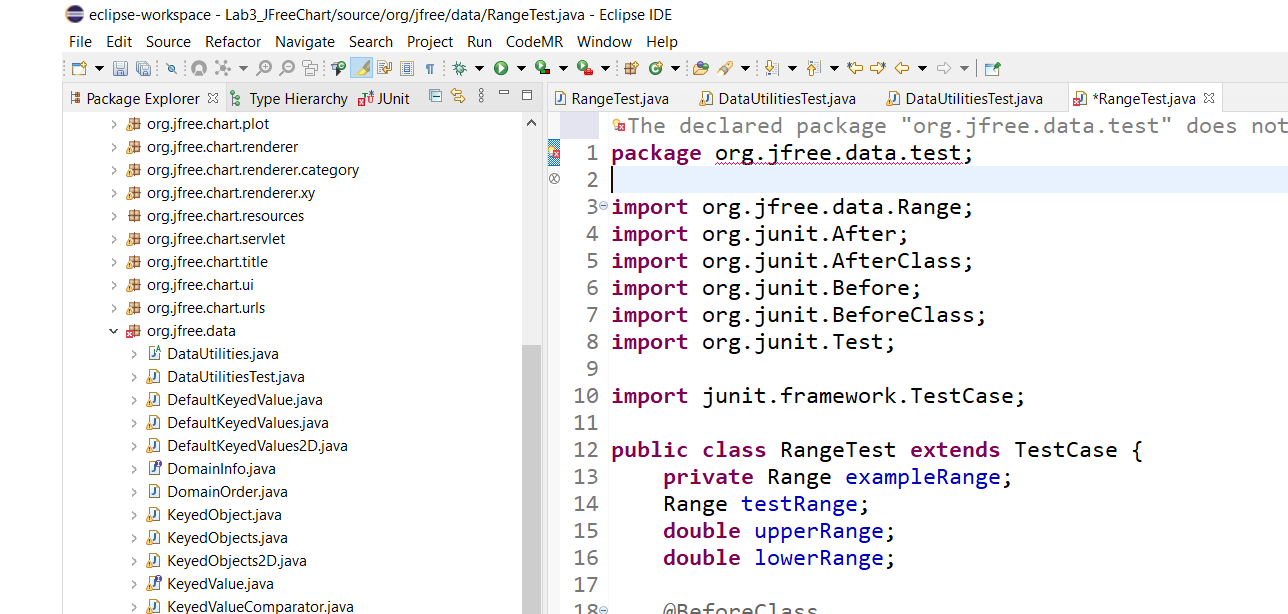


Figure 7 – Fixing the package location identified in top of the two new test files

### We need to use the JUnit version 4, like the previous lab

We would like to remind you that, for these labs, we need to use the JUnit version 4, as we also used in the previous Lab2. If you need to call how to set up your Java project to use JUnit 4, refer to the corresponding steps in lab2 document.

### A short online video for how to set up the project

A short online video by Dr. Garousi explains a summary of setting up a Java project for Lab3 (discuss in all the previous steps): [youtube.com/watch?v=snW7Wlp8D08](https://www.youtube.com/watch?v=snW7Wlp8D08)

## Running the Lab2 test suite and measuring its code (test) coverage using the tool

In Lab2, we ran test suites as JUnit suites and observed the outcomes. In Eclipse, there are two ways of running JUnit test suites, as shown in Figure 8. The 2nd way is to right click on a single JUnit test class or a folder in Project Explorer and choose “Coverage As” à “JUnit Test”. In this way, the test class(es) will run and also the coverage will be calculated automatically by Eclipse.

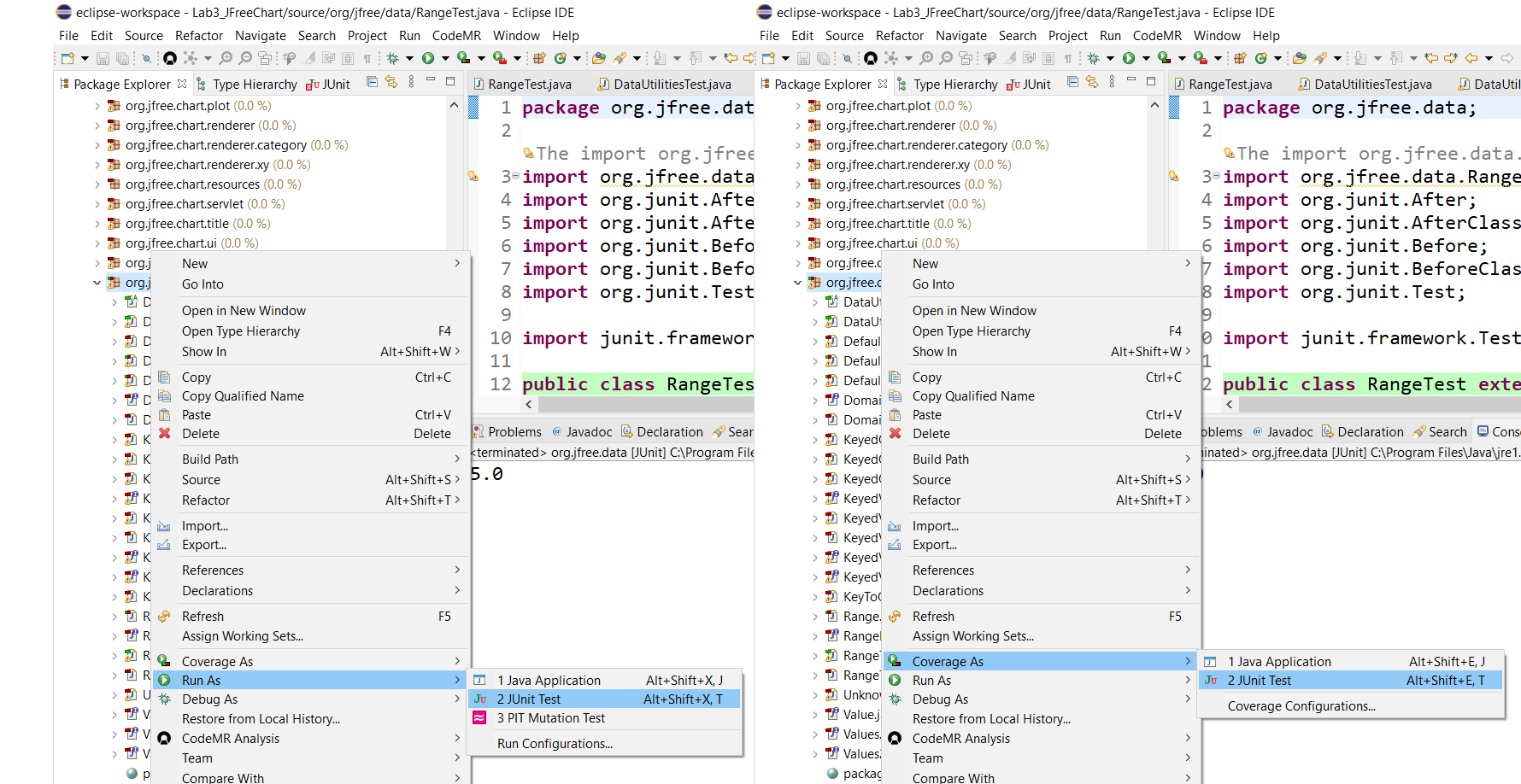


Figure 8 – Two ways of running JUnit test suites (with and without test coverage measurement)

Run the above command and observe what you see. Eclipse should automatically bring up the “coverage” tab in the bottom frame, like as shown in Figure 9. IF, for whatever reason the “coverage” tab is not automatically shown, you can show it in the screen by following the steps in Figure 10.

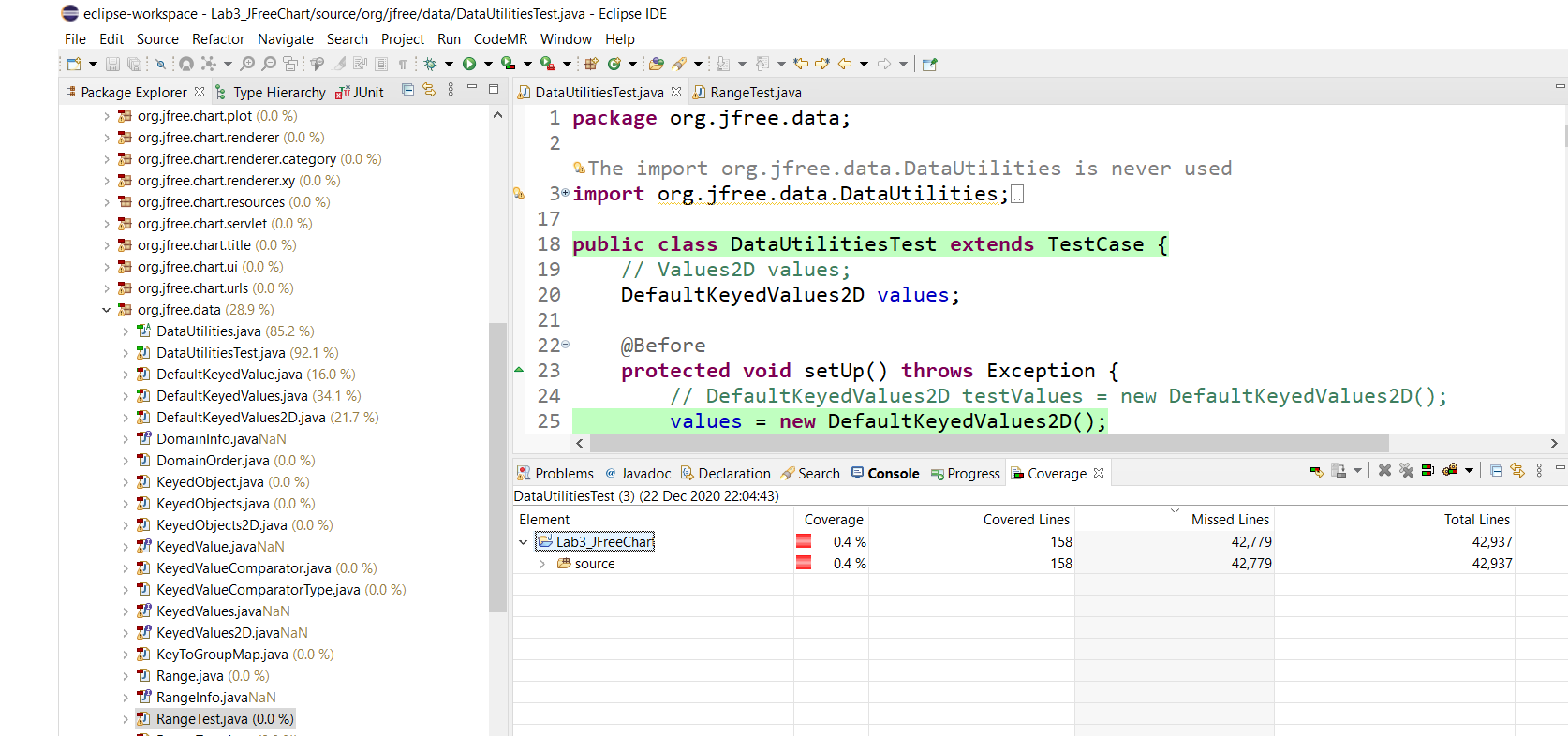


Figure 9 – The “coverage” tab in Eclipse

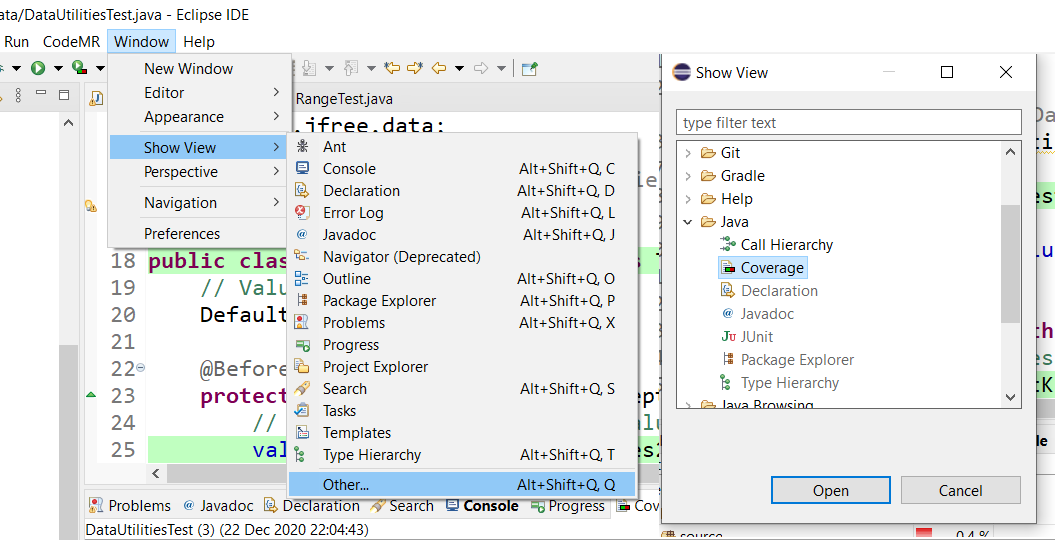


Figure 10 – Showing the “coverage” tab in Eclipse

After running a JUnit test suite in Eclipse, we use the data shown in the “coverage” tab to inspect the coverage results. Depending on what test cases you have in your RangeTest.java and DataUtilitesTest.java test files, you will get different coverage values (results). For a particular test suite that we have developed, we received the coverage value, as shown in Figure 11.

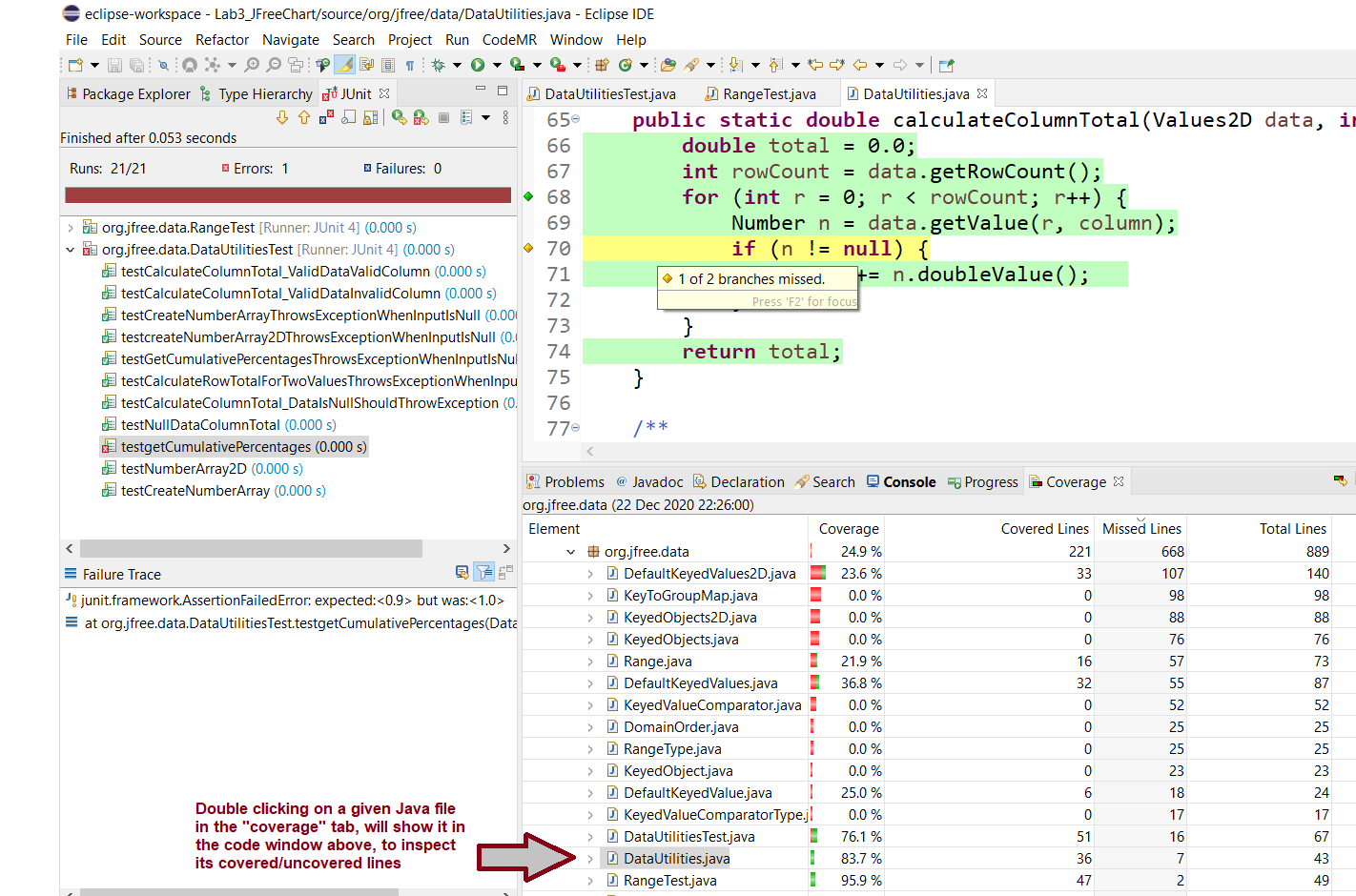


Figure 11 – Inspecting the coverage results after running a JUnit test suite in Eclipse

Now, it is your turn to carefully inspect the coverage results. Take your time to review and make sense of the coverage metrics of each Java class. For example, in Figure 11, we see that the coverage of DataUtilites.java class is quite high (83.7%). This is quite expected since we have a specific file (test suite: DataUtilitesTest.java) to test DataUtilites.java. For the case of Range.java, the coverage is lower (21.9% as seen in Figure 11) in the case of our test suite, since it has much less test cases (inside file RangeTest.java), thus it is testing that unit less thoroughly compared to how DataUtilitesTest.java is testing DataUtilites.java.

Let us note that, as shown in Figure 11, double clicking on a given Java file in the "coverage" tab, will show it in the code window above, letting the test engineer inspect its covered/uncovered lines. Do this and inspect the code-base for yourself. The green color means that the given code line is covered by the test suite, and red means uncovered. Yellow color means partial coverage of a given code line. For example, we see in Figure 11 that line 70 inside DataUtilites.java has been partially covered by the test suite: 1 of its 2 branches is missed (it is an if statement).

We can also switch the type of coverage metrics in the coverage tab: Line coverage, Branch coverage, etc. as shown in Figure 12. The default (first) view is often line coverage values. Change the coverage metrics in your review and inspect the results.

**Notes:**

* **Make sure that you discuss these with your group mate and ensure you understand how everything works in terms of coverage.**
* **Make sure to map your observations in this part to what you have learned in the lectures about code coverage**
* **Feel free to ask the instructor or the TA if you have any issues so far.**

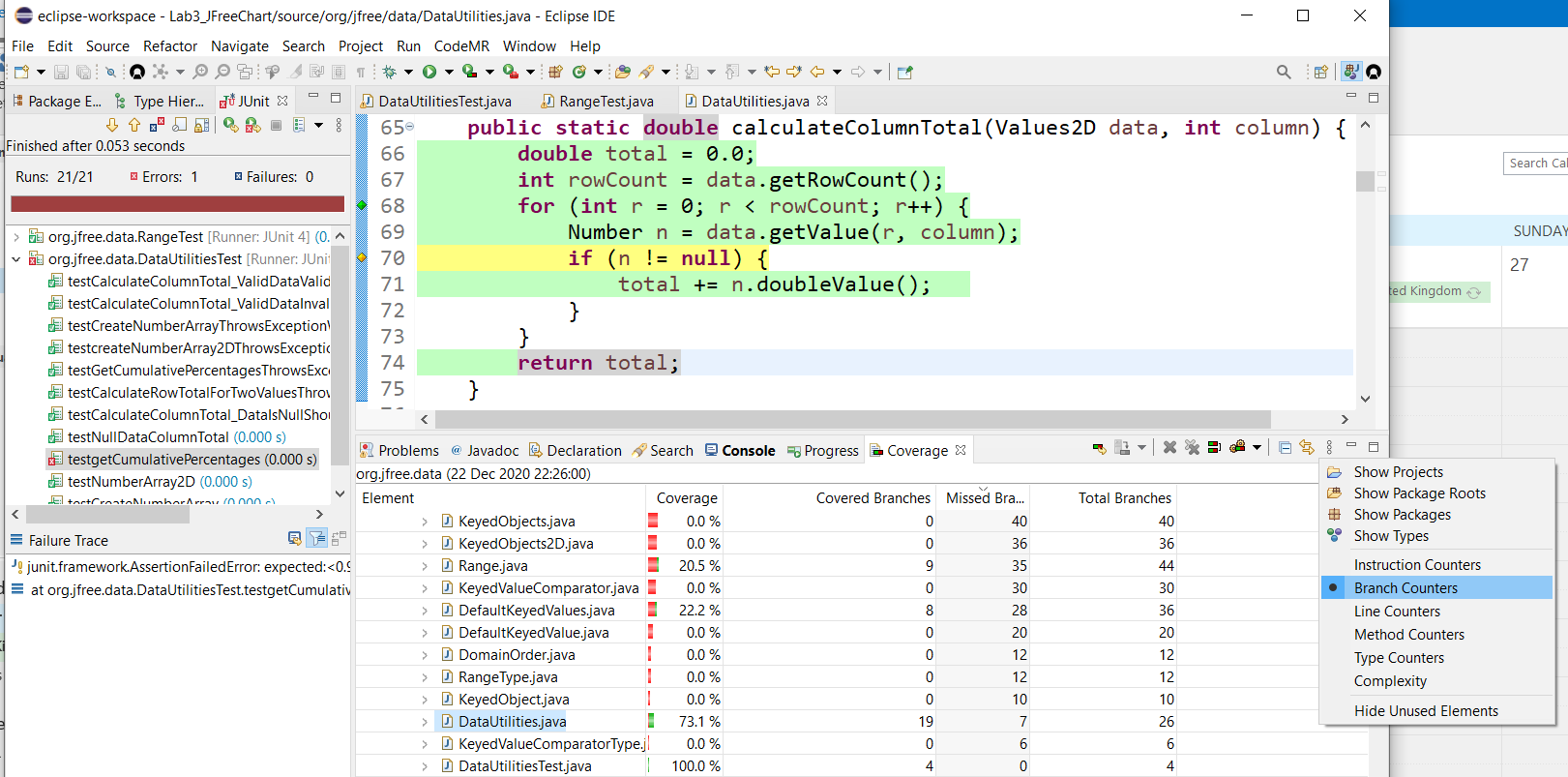


Figure 12 – Switching the type of coverage metrics in the coverage tab: Line coverage, Branch coverage, etc.

## Measuring the coverage of the SUT code-base, not test-code

We should mention that only measuring and analyzing the coverage of system under test (SUT), often called “production” code, as opposed to test code, makes sense. We have seen in the past years that some students measured and reported the coverage of test-code, in their reports. Let us note that this is totally **meaningless** (see Figure 13)**!** Do you know why? It is obvious that when you run a test suite, all of it will run and the coverage of test-code will be 100%. **Code coverage is ONLY meaningful for the SUT (also called: production code), as we want to know how much of the production code is covered / exercised / tested by our automated test code.**

Thus, make sure to not measure nor report the coverage of test-code itself, in your lab report.

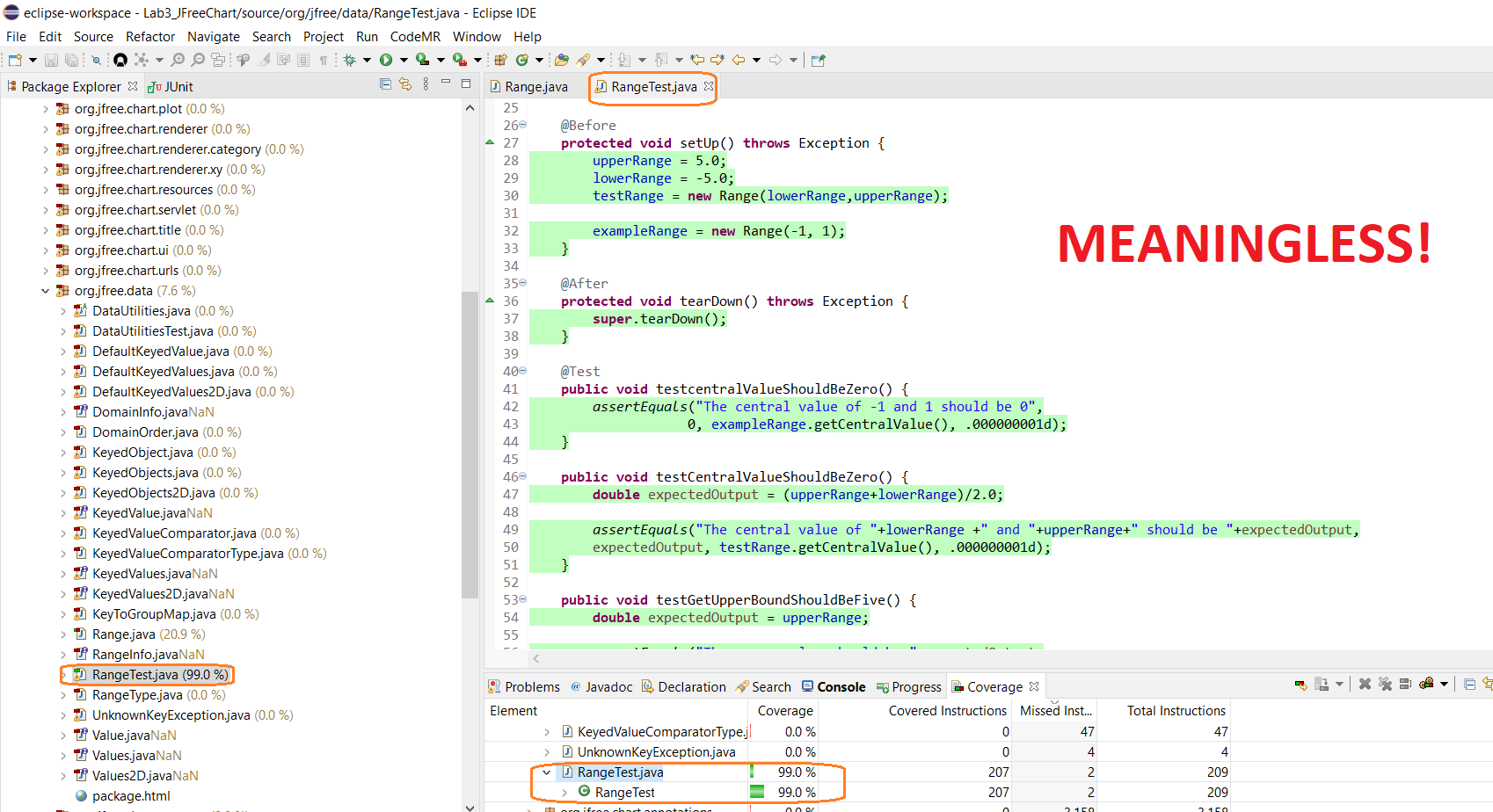


Figure 13 –Measuring, reporting and analyzing the coverage of test-code is meaningless

## Small-scale practice for developing new test cases to increase coverage

1. For example, we show in Figure 14 an example coverage status of the *Range* class. The branch coverage of the entire class is 20.5% here, and it shall be increased to minimum 70% branch coverage.

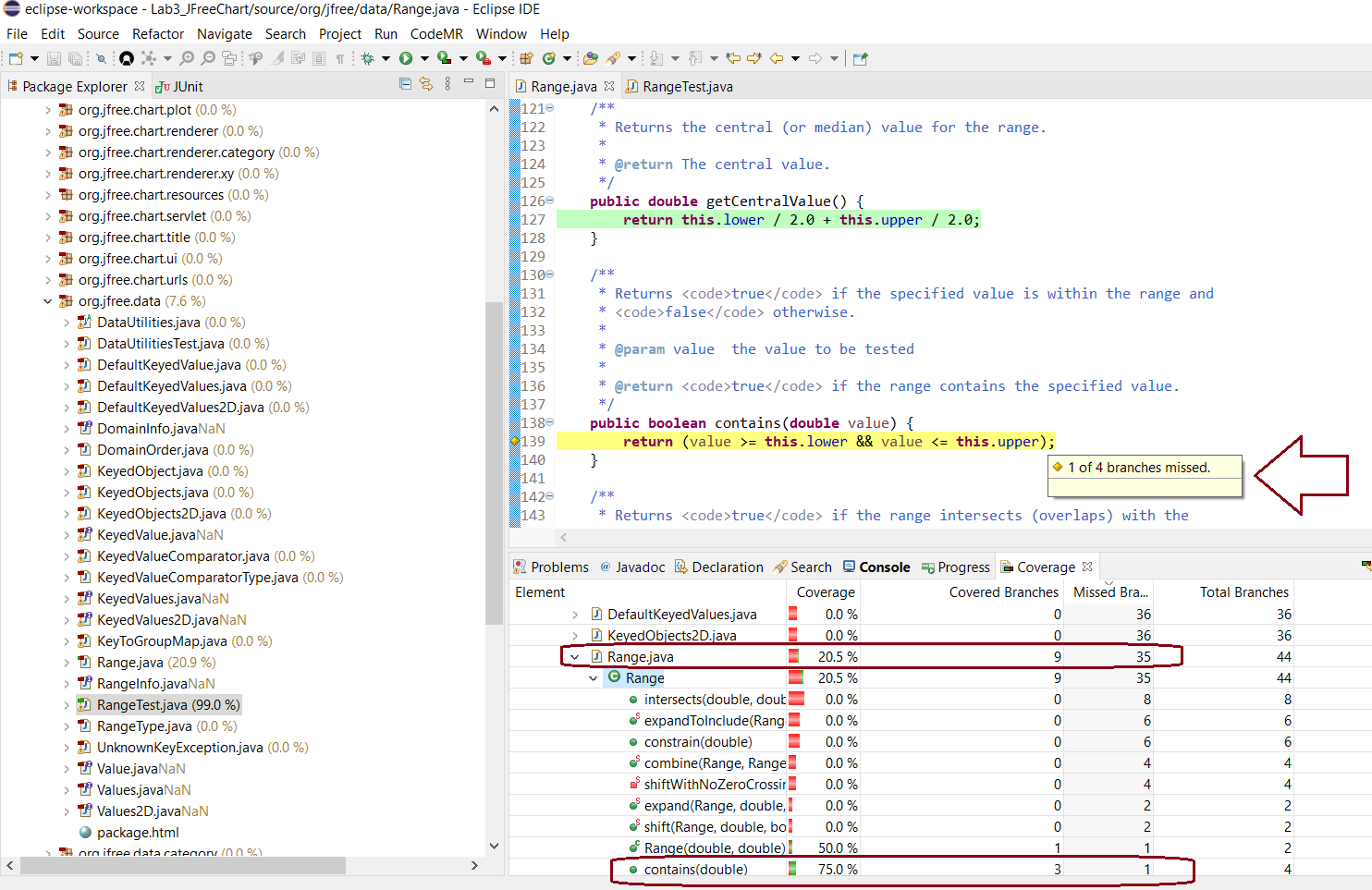
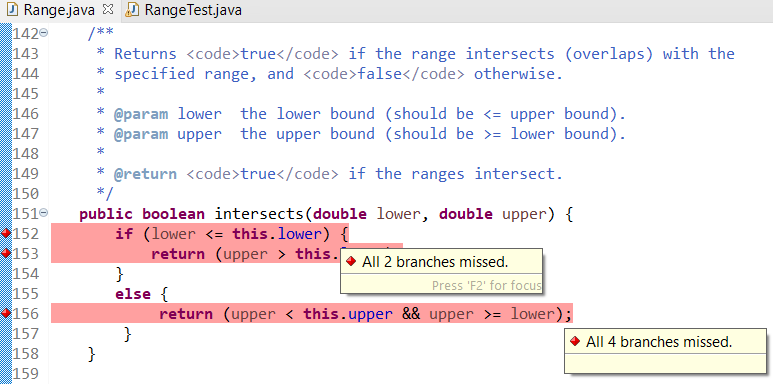


Figure 14 –An example coverage status of the *Range* class: Branch coverage of the entire class is 20.5%



1. We can do this by looking at the branch coverage data shown and determine which branches are missed (not covered) by our test suite. We see in Figure 14, for example, that the code of *Range* class has 35 branches in total and only 9 of them have been covered (a ratio of 20.5%). We then need to look in the code to see the yellow and red area. Move your mouse to such areas and pause. You will then see a hint box opened and tell you how many branches are missed (see the examples in Figure 14).

## Errors versus failures in JUnit test-case executions

1. As mentioned in the Lab2 document also, when your test-code is executed, it is an important test automation principle that if you see any “Errors” in the JUnit execution view, you shall fix them ASAP. **Note that Errors are different from Failures in test automation.** If you see any Errors, you shall carefully inspect your test code, fix the issue (e.g., using try/catch blocks if an error is due to uncaught exceptions), rerun the test suite and ensure that the number of errors shown by JUnit=0. For instance, if you are getting unwanted exceptions in your test code (from the SUT), you shall add the proper try/catch blocks, to convert the Error to Failure in test execution.
2. **There should be no "Errors", but there could be "Failures", when executing your final test-suite code.**
3. Similar to previous labs, we have intentionally injected several defects into the SUT, thus a number of your test case will (shall) correctly fail and you should leave them as they are. In fact, you shall report the list of failed test cases (indicating faults in the SUT) in your report.
   * **Note: The SUT versions used in labs 2 and 3 are (intentionally) different, i.e., there are different defects in each of them. This has been designed to simulate real-world testing scenarios for students, i.e., when a software system evolves, it gets to have different faults in it, across its different versions. Thus, the failures of test cases in your lab2 test suite when executed on lab2’s SUT version could and would be different when you run your lab2 test suite again on the lab3’s SUT version.**

# Instructions for the lab-work

After going through the familiarizations steps in the previous section, we now provide you with instructions steps for you to do the lab-work.

## First, write your test plan for white-box unit testing

1. As with any testing to be done, to begin with, you should first develop a test plan and document it in your report. This plan test should include:

* How you plan to develop tests to achieve the above adequacy (coverage) criteria. Note: In the test plan, do **NOT** provide the design/list of test cases (as it will be done/discussed in the next section), but you should instead describe the approach to do the white-box test-case design, not the TC design itself. The plan shall be written BEFORE your test-case design, and you should explain how you plan to increase the coverage of your Lab2 test suite to meet the thresholds given above
* Division of work-load for test-case design and development: Information about who (which student member of the group) will design and develop which tests

Note Your test plan for white-box unit testing should be detailed enough (e.g., at least half a page of A4).

## Test-suite improvement to increase the coverage ratio

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 pair programming / pair testing to help ensure quality assurance.

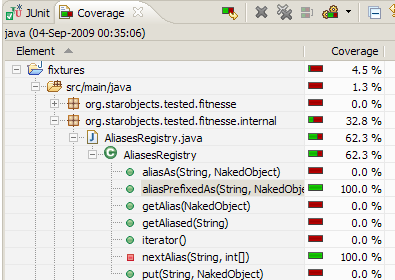
1. You need to develop more unit tests for the two classes under test that we have worked with since Lab 2:
   * org.jfree.data.Range and
   * org.jfree.data.DataUtilities

By doing so, your goal should be to increase code coverage of those two classes, when the test suite runs. For this lab, we want you to increase the coverage of your test suite from Lab2 to the following coverage thresholds:

* **Minimum 90%** **line coverage and 70% branch coverage on the entire Range class**
* **100% line coverage and 100% branch coverage on the entire DataUtilities class (**If for any reason, you cannot achieve the 100% levels mentioned above, explain the reason, and increase the coverage as much as possible**)**

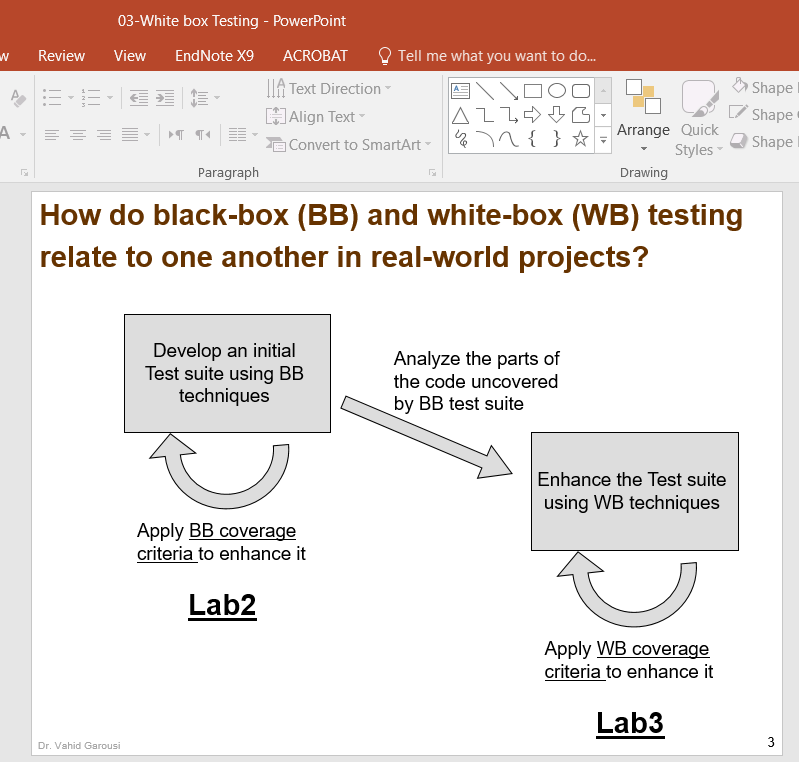
**Reminder from the lectures:**

* **T**o remind students once more **what class-level coverage ratio means, we provide another example of code coverage view of an open-source software below. To increase class-level coverage ratios, the test engineer does not have to design/ develop test cases (methods) for all methods of a class under test (CUT), but has the choice of choosing selectively or randomly a number of test methods, and increasing their test coverage.**
* **Essentially, if for example our goal is** 90% **class-level** line coverage, test engineer has to develop tests running (covering) 90% of all code lines inside the class under test.
* In the case of our lab3, it will be natural to first design and develop as many new test cases as possible to the methods for which you had already developed the tests in lab2 and if those added test cases are not enough to reach the above threshold goals, you will need to design and develop new test cases for “other” methods of the class under test



**Notes:**

* ONLY FOR THIS CASE 🡪 If your JUnit test-suite from Lab2 is already comprehensive enough to provide coverage values more than or equal the above two thresholds (for either of the classes under test): we still need you to design/develop at least 5 new test cases (methods) for each class under test, and add them to your JUnit test-suite, to learn how to increase code coverage by adding new test cases.
  + To do this, let us assume that your lab2 suite already provides LC% line coverage and BC% branch coverage. Note that, as per above, LC>=90, BC>=70.
  + Your test requirements is to increase each coverage value by 10%, to (LC+10)% line coverage and (BC+10)% branch coverage.
  + To do that, you need to find any part (code of any methods) of the classes Range and DataUtilities, which have not reached full line coverage or branch coverage, and design / add new test cases to increase those coverage values.
* Although the focus in adequacy criteria has changed (it is now on source code), to develop the test cases, test *oracle* (expected output of a method) should always be derived from the requirements (as provided in the Javadocs of the SUT in Lab2), and not from code.
* **Very important note:** To design / develop additional test cases (test methods) to increase test (code) coverage in lab3 to reach (meet) the provided coverage thresholds (goals), you should do ONLY white-box testing (test-case design using code coverage), and NOT Black-box Testing (test-case design using software requirements), which was the topic of lab2. Again, if you have not fully learned (or are not sure) about the difference of white-box (WB) testing (test-case design using code coverage), and black-box (BB) Testing (test-case design using software requirements), you should ensure attending all the lectures and reviewing the lecture problem examples to learn the concepts. Also, we have discussed in details in the lectures, how WB and BB testing complement each other in a typical test process:



* + **You should NOT fix the defects in the SUT code**: During your white-box testing activity (designing new test cases using code coverage, to improve code coverage), you may notice/find any of the intentionally-injected defects. You should NOT fix any of those defects in the code in your lab-work. Just leave them as they are; and continue with your white-box testing activity.

1. Carry out your test plan, use white-box (WB) testing approach by developing new test cases (test methods) for any method of the above two classes, which have low coverage values.
2. White-box testing approach means that, for any missed line-of-code and branch, you need to design on paper, and then develop in JUnit additional test method, and add them to your test suite (test class), so that the missed lines-of-code and branches are “covered” (executed / tested) by the test suite. After you add a few test methods, run your test suite and check if the coverage is slowly going up. Once you reach the above thresholds (highlighted above), you can stop your test-suite improvement activity.
3. As an effective software test-automation engineer, you need to keep each test case (for a single control flow path for example) in a separate method, for example: testMethodXPositiveValues() and testMethodXNegative Values(), instead of a single testMethodX(). This will help keep test cases consistent, and make the measured coverage metrics more meaningful.
4. Note that, similar to Lab2, the SUT classes have intentional random defects in them, and thus several of your test cases could and should fail. Therefore, to develop test oracles in your test code, you need to follow the specifications, not the actual results by the SUT code.
5. **Again, like Lab2, and always when developing JUnit tests, you need to make sure that you should not get any “Errors” in JUnit when you run your test suite, similar to Lab2. But getting “Failures” from test executions is fine, as they will denote faults in the SUT.**
6. If you have divided the test cases, to be developed, 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. Include all the updates made during the peer-review process in your lab report.
7. Measure the code coverage (only control flow metrics as listed above) of your entire test suite, and record detailed coverage information for each class and method. Include this information (plus screenshot from coverage data) in your lab report.

## Analyzing and discussing how your additional test cases (test methods), in lab3 compared to lab2, improved code coverage of methods under test in Range and DataUtilities

1. Recall that, in Lab2, you developed test methods for five methods from each of classes: Range and DataUtilities
2. During the previous steps of lab3, you improved the test coverage of certain methods under test, or you developed new white-box test cases for some other methods (in the previous steps), to increase overall the coverage ratio to the threshold given by the lab doc above

### Summary table of test-suite size from in lab3 compared to lab2

1. As the first step in this section, include and fill the following table in your lab report, showing the increase in number of test cases for those 10 methods under test from Lab2 to Lab3. Note that, obviously, the number of test cases for each case in the table below may stay equal or go up.

**Note:** If in lab3 the previous steps, you have chosen additional methods from the two classes under test, to develop white-box test cases for (using coverage data), include them also in the following table. Obviously, for those methods that you had no test methods in lab2, but developed new test cases starting from lab3, they will have "0" in terms of the number of test cases (test methods) that you had done in Lab2.

|  |  |  |  |
| --- | --- | --- | --- |
| **Class under test** | **Method under test** | **Number of test cases (test methods) in Lab2** | **Number of test cases (test methods) in Lab3** |
| Range | combine(Range range1, Range range2) | 6 | 9 |
| contains(double value) | 8 | 11 |
| … | … | … |
| … | … | … |
| … | … | … |
| **Totals** | **Total number of your test cases for Range class in Lab2=x** | **Total number of your test cases for Range class in Lab3=x** |
| DataUtilities | calculateColumnTotal(Values2D data, int column) | 6 | 7 |
| calculateRowTotal(Values2D data, int row) | 9 | … |
| createNumberArray(double[] data) | 11 | … |
| createNumberArray2D(double[][] data) | … | … |
| getCumulativePercentages(KeyedValues data) | … | … |
| **Totals** | **Total number of your test cases for DataUtilities class in Lab2=x** | **Total number of your test cases for DataUtilities class in Lab3=x** |

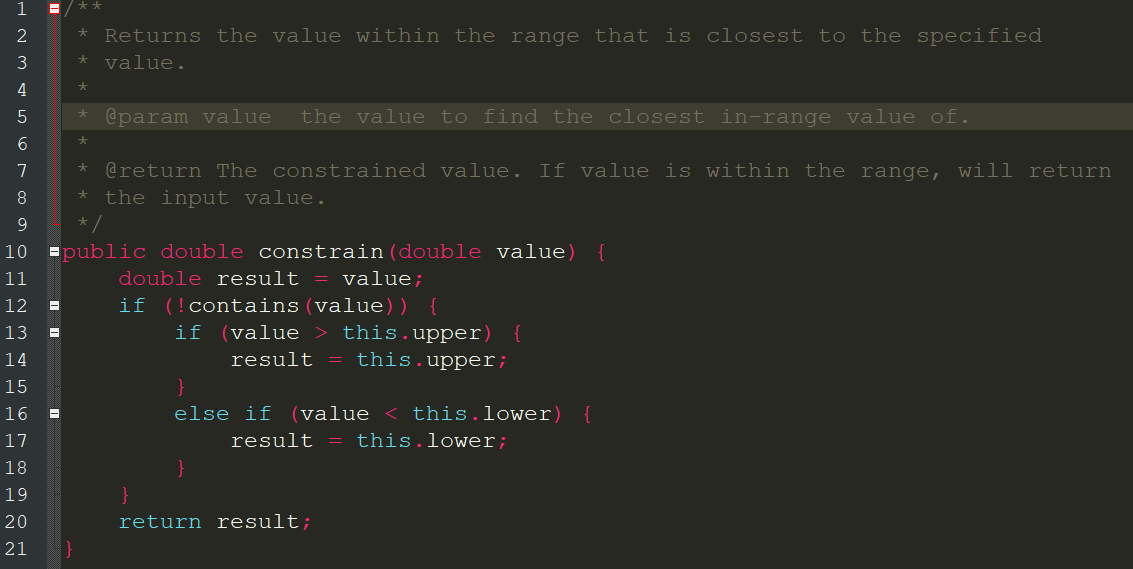
**Note: The above is just an example. We are NOT providing the number of test cases for you.**

### Discussing details of design and development of additional test cases

1. Next, we want you to discuss some details of how your design and development of additional JUnit test cases (test methods), in lab3 compared to lab2, improved code coverage of methods under test.
2. For this purpose, from among your coverage improvement tasks done in the previous steps, choose randomly two methods under test from Range (from a total of five that you had tested), and two methods under test from DataUtilities (from a total of five methods that you had tested for each class), for the following analysis
3. For the above total four (4) methods under test in the SUT, provide discussions in your report on: how you have designed and developed additional test case (test methods) using code-flow coverage information, and how those test cases have increased code coverage.
4. To support your discussion in the report, you need to show screenshots of test-coverage views of the method under test, where the coverage tools has colored the method lines in green, red and yellow, BEFORE and AFTER you have designed and added your additional test cases. Also, in your lab report, add screenshots of the code of the new test-methods that you have added to increase code coverage. You should also provide some explanations of the uncovered lines of code (in the “before” cases) that you analyzed to design / add the additional test cases. You do not need to design / present control-flow diagrams in your report.

## Conduct small-scale data-flow coverage analysis manually

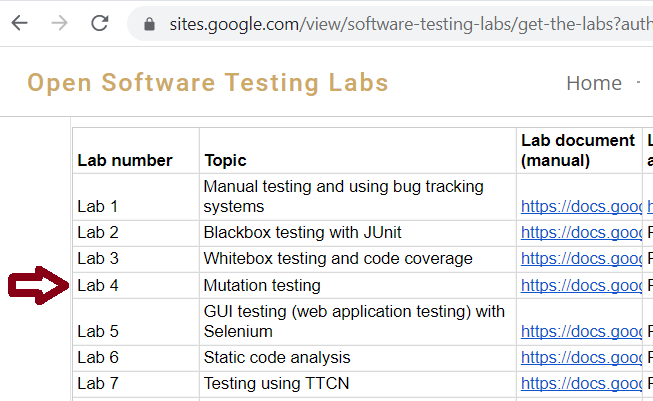
1. To become more familiar with data-flow coverage and achieve a deeper understanding of how coverage tools work, calculate the coverage for the following individual method by hand (not using any tool): For only the Range.constrain(double) method of the class Range under the org.jfree.data package, conduct Data-Flow Analysis (DFA). Its code is shown below. **Important note:** **Use the EXACT code-line NUMBERS below in your work.**



1. To do the DFA analysis, just like done in the lectures in many examples, you need to identify code blocks and number them (using a tabular format), draw the CFG and then the definition and usage sets of code blocks and variables.
   * For an example of how to do this, review the lecture slides and also see the example done in Appendix A.
   * Once you have done the DFA as per above, consider each of the following test cases, and **for each of them separately**, calculate the data-flow coverage ratios (percentage values) by tracing through the code via DFA:
     + TC1: the range object being [2, 5], and the parameter *value* of the method =3
     + TC2: the range object being [-2, 4], and the parameter *value* of the method=11
     + Note: You need to calculate and report two **separate** coverage ratios (percentage values), one for each TC above
   * Details of your analysis will need to be included in your report.

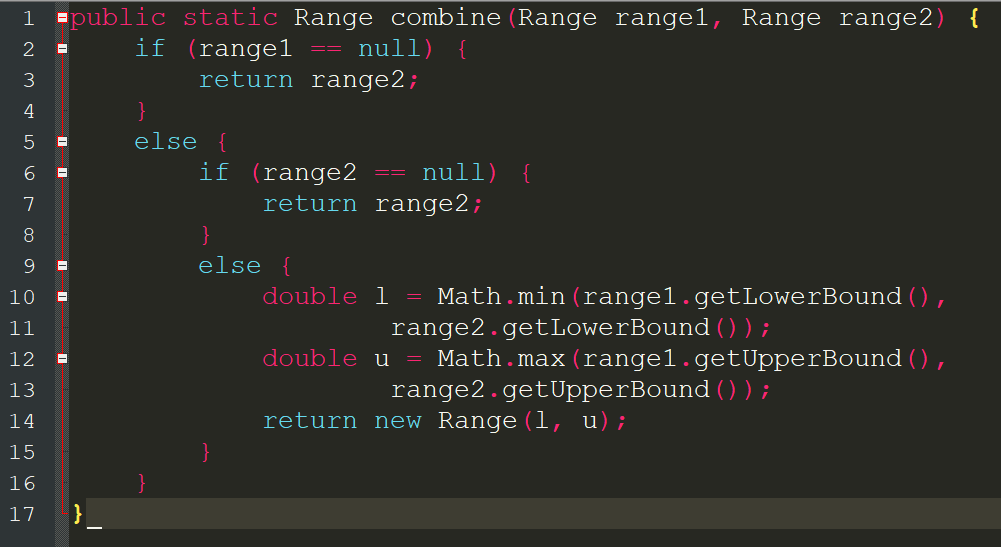
## Conduct small-scale mutation testing (analysis) manually

Since we have learned the concept of mutation testing in the lectures, we would have liked students to use mutation testing tools in the labs. However, the term duration is quite short and thus, we could not have a full lab on mutation testing tools. Note that, if you wish, students can do a lab on mutation testing on their own, using the instructions on the open labs’ website, in this URL: [sites.google.com/view/software-testing-labs/get-the-labs](https://sites.google.com/view/software-testing-labs/get-the-labs) (shown below):



Even if we are not doing any full lab on mutation testing tools, we still want our students to practice and learn mutation testing by conducting the following exercise, by hand and not using mutation testing tools.

1. **First, ensure reviewing all the mutation testing concepts and examples solved in the lectures**
2. For this exercise on manual mutation testing, the method under test is Range.combine(). Its source code is shown below.



1. Note: Before starting to apply mutation testing, on an existing given test suite, as learned in the lectures, all test cases of the suite shall already pass (there should be no failures, and of course no test “errors”). If you have any test failures for the method under test, feel free to safely leave out (comment out) those test cases from your test suites for the purpose of the exercise in this section.
2. We want you to inject, manually (not using any tools), five mutations inside the method under test, each resulting in a mutant method under test. From the lectures, we know that we would select a number of Mutation Operators to create mutants. We want you to choose five random and different types of Mutation Operators to create the five mutant method, as learned in the lectures. Make sure to mention in your report, which five Mutation Operators your group decided to choose and apply. Of course, you should select those Mutation Operators, that will be applicable to the code of the method under mutation test, e.g., if the method does not have any Relational operators, choosing the *Relational operator replacement (ROR)* mutation operator, will not make sense, because you cannot apply it in the code of the above given method.
3. As for the test suite to start with, if in the lab2, you had designed and developed a test suite for your chosen method under mutation testing (and also improved already up to this stage in your lab3), take the final version of that test suite. If however, for this particular method, you have not designed / developed any test suite before this part of the lab3, you need to design a test suite for it, from scratch (empty test suite), for this exercise.
4. To document how you will do the manual mutation testing, inspect the test cases by hand (and using mind / eye), not any tools, and document the following information, by running each test-case in your latest test suite on each of the five mutants, separately. Running production or test code by hand/eye is called [code walk-through](https://www.google.com/search?q=code+walkthrough) or [hand-tracing of code](https://www.google.com/search?q=run+code+by+hand).

Note: You will have five of the following tables in your lab report, one for each mutant. In your report, you should explain why each test case Passes/Fails.

**Status of running each test case on Mutant 1:**

|  |  |  |
| --- | --- | --- |
| **Test case of the latest test suite** | **Outcome of test-case execution on the mutant** | **Explain why the test case fails/passes. See the examples covered in the lectures** |
| TC1 | Pass or Fail |  |
| TC2 | Pass or Fail |  |
| … | … |  |

1. To summarize your mutation-testing process, student should also include and fill the following table (Summary table of all mutants), in your report.
2. Recall from the lectures that, when we execute a given test suite on a given mutant, if any of its test cases fails, it means that the test suite is strong enough already to distinguish the given mutant (injected fault), and thus all is good.
   * However if none of the test cases in the test suite fail (meaning that all test cases PASS), it means that the test suite is not strong enough to distinguish the given mutant, and thus we need to design and add a test case to the test suite to distinguish (kill) the mutant (the new test case should fail, as per the three conditions that we have learned: Reachability, Infection, Propagation).
   * Again, review all the lecture examples
   * If your existing test suite is strong enough, to distinguish (kill) all the five mutants that you have created in this exercise, without the need to add any new test case to the test suite, we still want to choose one single additional mutation operator, and use it create a 6th mutant, for which your existing test suite would not be strong enough, to distinguish (kill) that 6th mutant. Then afterwards, as discussed above, you need to design and add a new test case to the test suite, to distinguish the mutant. If it is impossible to have such a mutant in your case, explain in detail in your report why.
3. **Additional notes:** 
   * **You should not develop nor provide in the report any JUnit test "code" for the new test cases that you design in this section (mutation testing), including the following table. Test cases in our module are always in the form of test cases, that we have learned in the lectures, e.g., test inputs and the expected output/outcome.**
   * **All the mutation testing work should be done manually, on paper.** By "running" your test suite in this section and the table below, we do not mean developing test-code to run the test cases, but to analyze test cases (run them manually in our minds), which is called [code walk-through](https://www.google.com/search?q=code+walkthrough) or [hand-tracing of code](https://www.google.com/search?q=run+code+by+hand).

**Table-Summary table of all mutants**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mutants | Mutation Operator used | Resulting of running the latest test suite | Explain why the test suite fails/passes. See the lecture examples | If needed (the test suite has Passed on the mutant), design and write below a new test case to be added to the test suite to distinguish (kill) the mutant; and discuss “how”/why the new test case will distinguish the mutant. Your answer should include the three criteria learned in lectures: reachability, infection and propagation | Running the improved test suite on the mutant after adding the new test case should FAIL. Verify and discuss below that it indeed fails. |
| 1 |  | Pass or Fail |  |  |  |
| 2 |  | Pass or Fail |  |  |  |
| 3 |  | Pass or Fail |  |  |  |
| 4 |  | Pass or Fail |  |  |  |
| 5 |  | Pass or Fail |  |  |  |

# Summary

Students should now have a good understanding of measuring test suite adequacy based on coverage of the SUT’s code. This should include an understanding of some of the different control flow and data flow coverage criteria, and the effort needed to satisfy these coverage criteria.

Students should now have an idea of some of the tradeoffs that occur when choosing different test suite adequacy criteria for testing.

# Deliverables

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

## Lab Report

* Students will be required to submit a report on their work in the lab as a group.
* You should use the template Word file “Lab 3-Report Template.doc”, provided online under the Lab 3 folder.
* The lab report should be submitted in Word format (not PDF format, etc.)

## JUnit test suite

* JUnit test-suite test-code should not be submitted in Canvas.
* Instead, the test-suite Java files should be in your team’s GitHub repository, and also they should be regularly updated (committed) during the lab work duration (two weeks) by both students, showing that the team has actively worked on the lab work.
* The folders of your team’s GitHub repository should be properly organized for each lab. Reminder: you should only have one single repository for the entire team, and only have one single repository for the entire module, it should have folders for each lab in its root, named as Lab0, Lab1, ...
* Your GitHub repo should ONLY include the two Java test-code files that the lab document has asked to be developed. Do NOT upload all your project, and unwanted files such as .CLASS files, JAR files, etc. Having extra unwanted files in GitHub will lead to mark deductions.

# 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/](https://sites.google.com/view/software-testing-labs/)

# References

[1] "CodeCover," Internet: <http://codecover.org> [Dec 3, 2011]

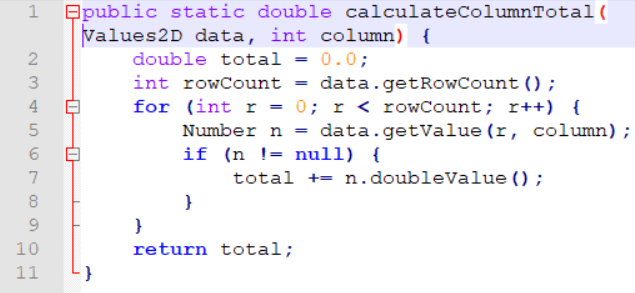
[2] "Eclipse.org," Internet: <http://www.eclipse.org> [Dec 3, 2011]

[3] "JFreeChart," Internet: <http://www.jfree.org/jfreechart> [July 3, 2008]

[4] "JUnit," Internet: <http://www.junit.org> [Dec 3, 2011]

# Appendix A – Example data-flow coverage calculated manually

For demonstration purposes, the DataUtilities.calculateColumnTotal method is analyzed next. To start with, the code is inspected, and the code “blocks” shall be identified in a table like the following. The concept of code “blocks” has been discussed in the lectures. Each code block will be a node in the CFG.



n != null

START

r < rowCount

r >= rowCount

END

**Step 1 -Identifying code “blocks” in a tabular form:**

|  |  |
| --- | --- |
| **Line number(s)** | **Code block (node in the CFG)** |
| L1-3 | 1 |
| L4 | 2 |
| L5-6 | 3 |
| L7 | 4 |
| L10 | 5 |

**Step 2 -Designing the CFG using code block numbers**

Based on the above information, we design the CFG as shows in the right-hand side à

**Step 3- Identifying the variables’ def / use from the CFG, in a tabular form:**

From the control-flow graph (CFG), we find all definitions and uses at each node.

|  |  |  |  |
| --- | --- | --- | --- |
| **Node** | **Defines** | **c-uses** | **p-uses** |
| 1 | data, column, total, rowCount | data |  |
| 2 | r | r | r, rowCount |
| 3 | n | data, r, column | n |
| 4 | total | total, n |  |
| 5 |  | total |  |

From here, the table can be further refined to include all the definition-clear-use paths, with the computation uses and the predicate uses clarified.

**Step 4- Identifying the definition-clear-use paths, in a tabular form:**

* *dcu(v, i)* is the set of definition-clear-computation-use paths if the variable v is defined at node i
* *dpu(v, i)* is the set of definition-clear-predicate-use paths if the variable v is defined at node i

|  |  |  |
| --- | --- | --- |
| **Node** | **dcu(v, i)** | **dpu(v, i)** |
| 1 | dcu(data, 1) = {1, 3} dcu(column, 1) = {3} dcu(total, 1) = {4, 5} | dpu(rowCount, 1) = {(2, 3), (2, 5)} |
| 2 | dcu(r, 2) = {2, 3} | dpu(r, 2) = {(2, 3), (2, 5)} |
| 3 | dcu(n, 3) = {4} | dpu(n, 3) = {(3,4),(3,2)} |
| 4 | dcu(total, 4) = {4,5} |  |
| 5 |  |  |
|  | Total number of def-clear c-use paths to cover = 9 (10 total in the above sets of paths, excluding the path from node 1 to itself) | Total number of def-clear p-use paths to cover = 6 |

**Step 5 - Calculating data-flow coverage ratios (percentage values) for a given test case or test suite, on all the definition-clear paths:**

At this stage, for a given test case or a test suite (set), the coverage ratios of the all-uses, all computation-uses or all predicate-uses can be calculated by tracing the nodes covered and calculating the percentage of uses covered. For example, let us assume that, for a given test case (the test input is an empty set of Value2D data), nodes 1, 2, 5 are executed, then the following definition-clear paths from the above table will be covered:

* def of variable *total* in node 1 and its c-use at node 5: 1-2-5: 1 instance of c-use in above table
* def of variable *rowCount* in node 1 and p-use in edge (2, 5): 1-2-5: 1 instance of p-use in above table
* def of variable *r* in node 2 and p-use in edge (2, 5): 1-2-5: 1 instance of p-use in above table

Thus, 3 (three) definition-clear paths out of the total 9+6 (=15) definition-clear paths (from the table above) would be covered, yielding an all-uses coverage ratio of 3/15=20%.