

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

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:	2
1 INTRODUCTION	2
1.1 Objectives	2
1.2 This lab is a Group work	2
1.3 Testing tools	2
1.3.1 A short review of Code Coverage Tools	3
1.3.2 Our choice: EcEmma	6
1.4 System Under Test	7
1.4.1 Purpose of the System Under Test (SUT)	7
1.4.2 Usage of the System	7
2 FAMILIARIZATION	8
2.1 Create an Eclipse Project	8
2.2 Import a Test Suite	9
2.3 Running the Lab2 test suite and measuring its control-flow Coverage	11
3 INSTRUCTIONS	15
3.1 Test-suite improvement to increase the coverage ratio	15
3.2 Measure Data-flow Coverage manually	17
4 SUMMARY	17
5 DELIVERABLES AND GRADING	17
5.1 JUnit Test Suite (40%)	17
5.2 Lab Report (60%)	18
ACKNOWLEDGEMENTS	18
REFERENCES	18
APPENDIX A – EXAMPLE DATA-FLOW COVERAGE CALCULATED MANUALLY	19

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

1 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 base their test cases on the control and data flow of the code for the SUT, as we have learned it in the lectures.

1.1 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

1.2 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.

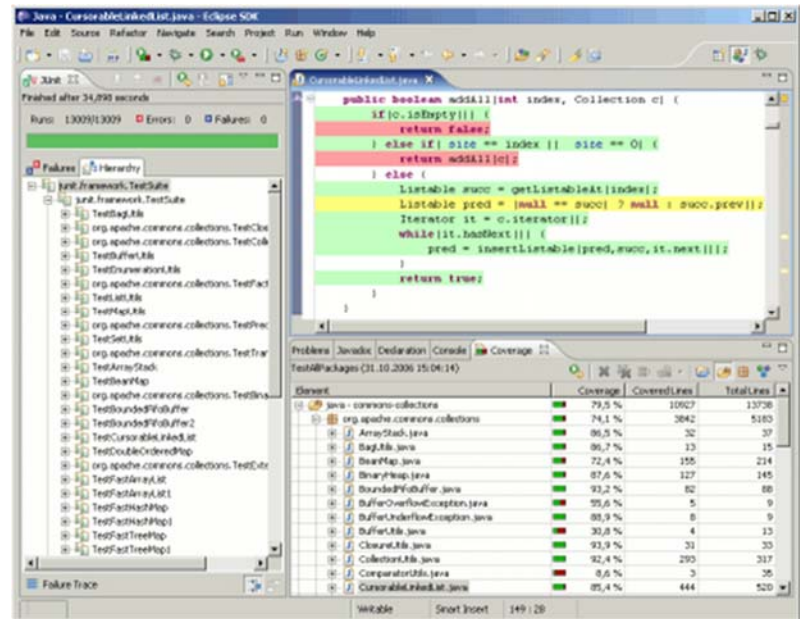
1.3 TESTING TOOLS

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.

1.3.1 A short review of code coverage tools

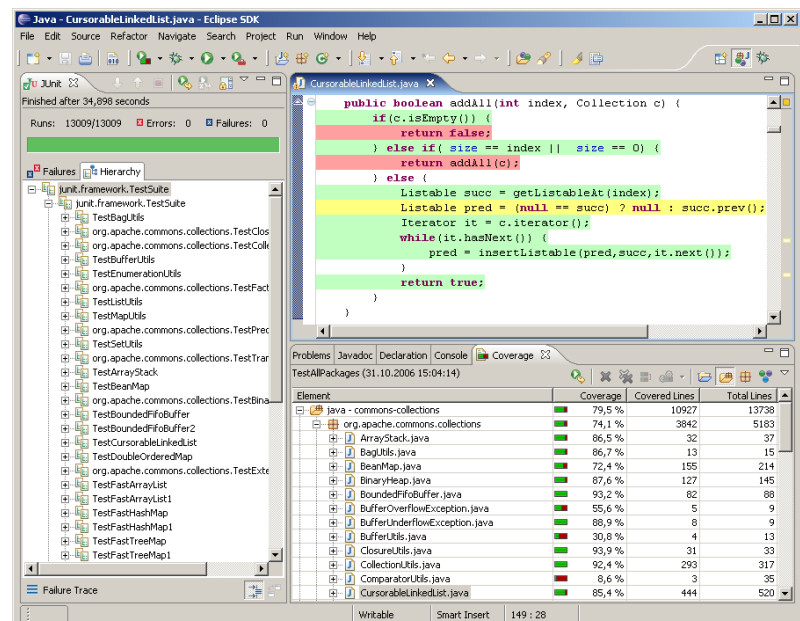
Clover

- Java code coverage
- Eclipse and IntelliJ plug-ins
- Free for open source projects
- www.atlassian.com/software/clover



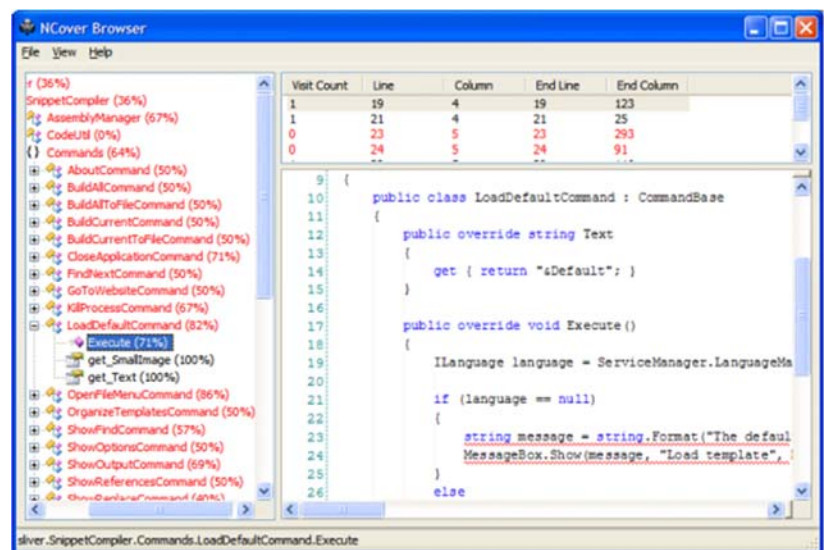
EclEmma

- Java coverage
- Eclipse plug-in
- Open source
- www.eclEmma.org



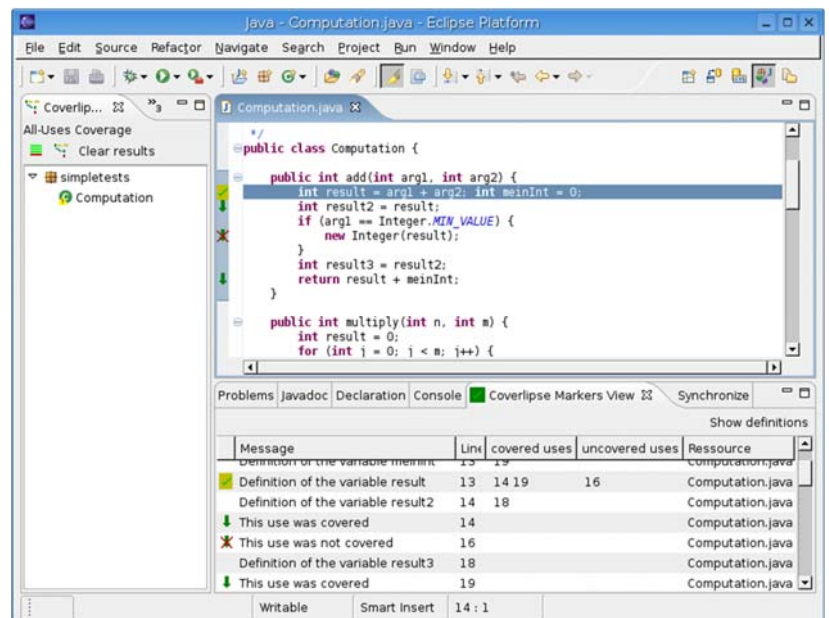
NCover

- C# code coverage tool
- Self-contained executable or can be integrated with an IDE
- www.ncover.com



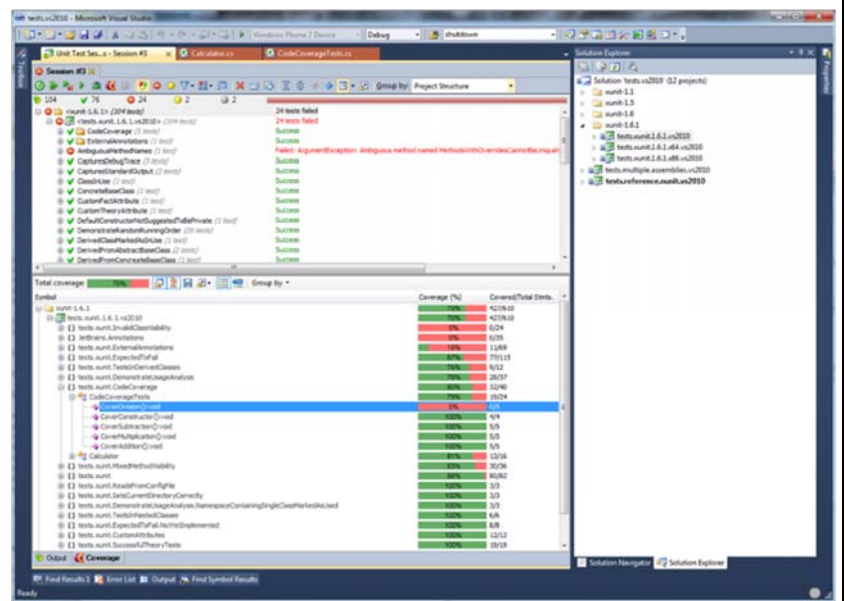
Coverlipse

- 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



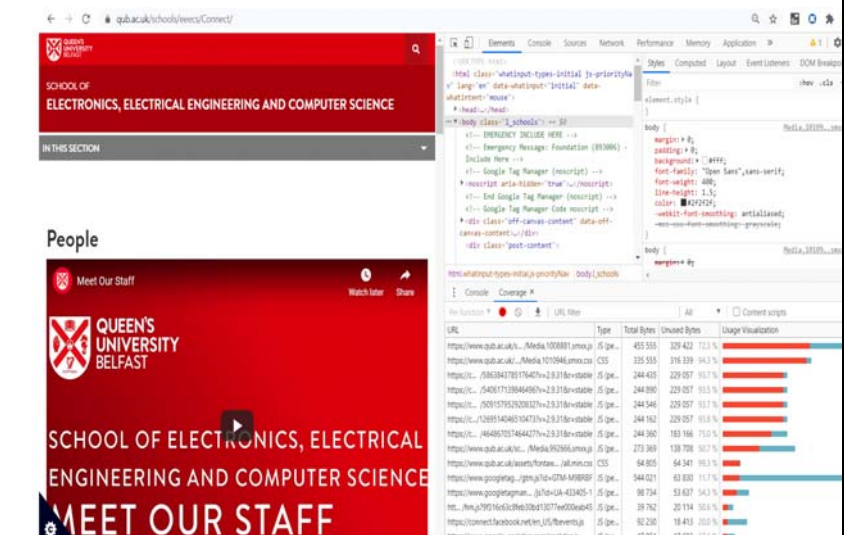
dotCover

- C# code coverage
- Visual Studio plug-in
- www.jetbrains.com/dotcover



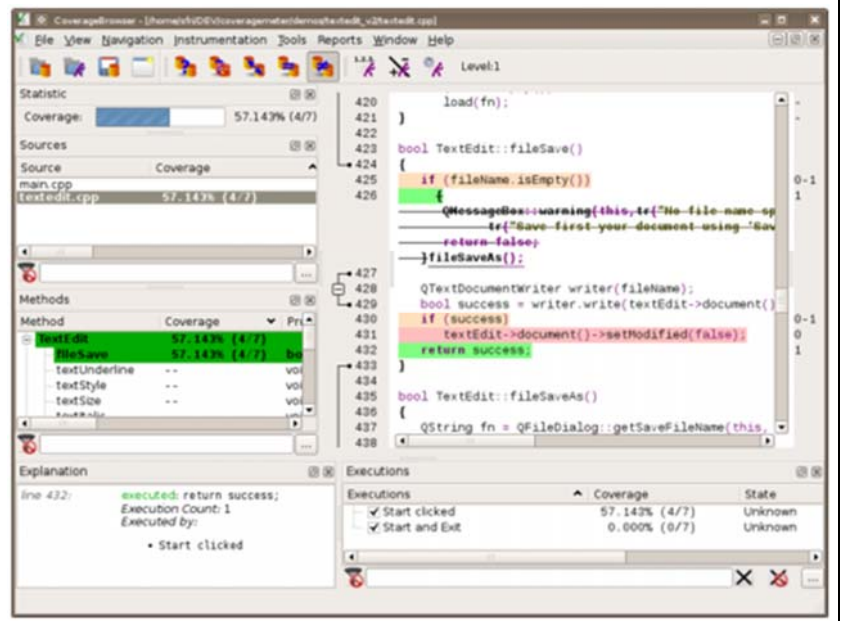
Chrome browser: Coverage tab in the Chrome DevTools module

<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/



1.3.2 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 *Help* → *About* → *Installation Details*, menu. You should see EclEmma as shown in Figure 1. 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.

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

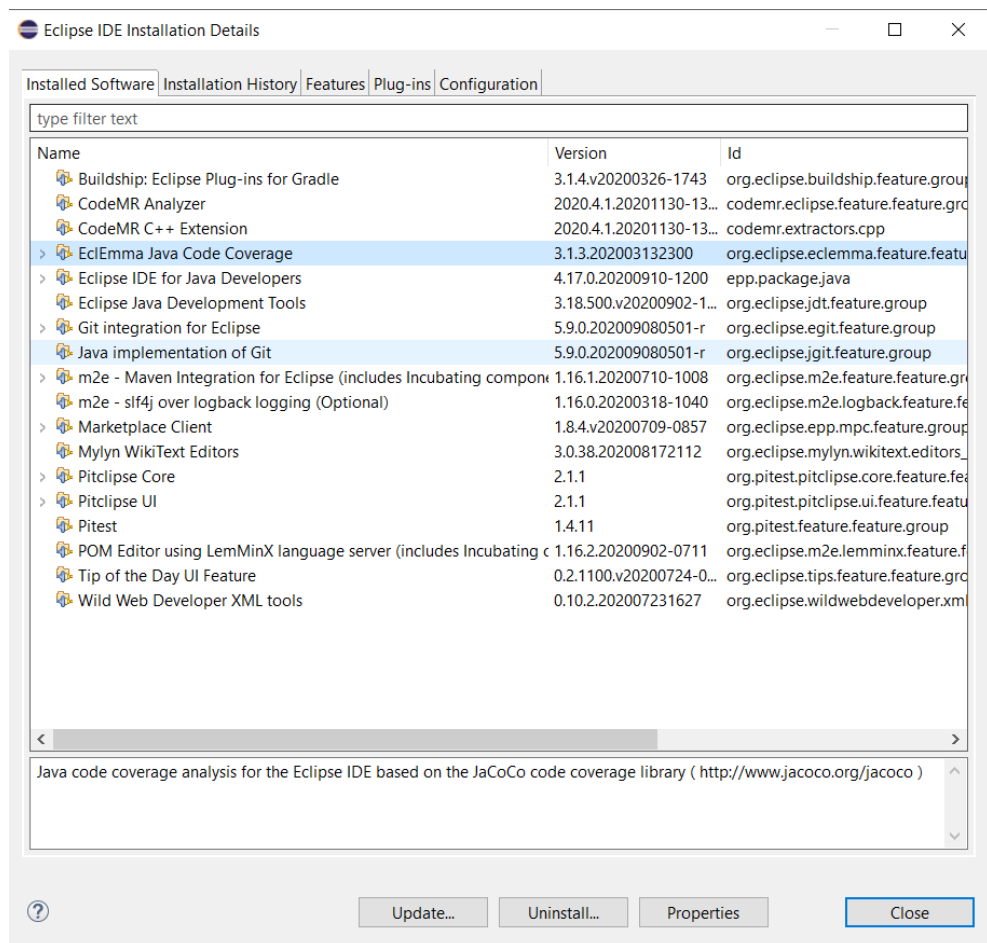


Figure 1 – Seeing the list of installed software and plug-in's in Eclipse and ensuring the installation of the EclEmma coverage plug-in

1.4 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.

1.4.1 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.

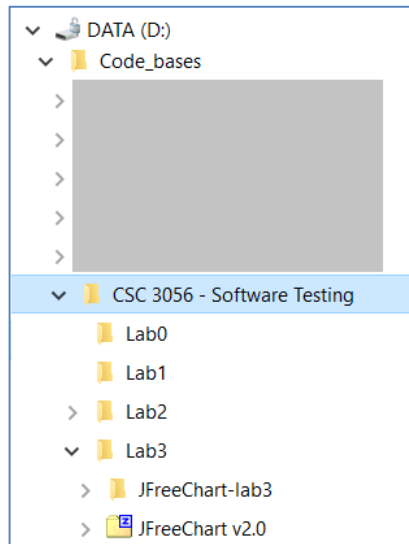
1.4.2 Usage of the system

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.

2 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.

2.1 CREATE AN ECLIPSE PROJECT

2. Open Eclipse Java IDE.
3. Open the *New Project* dialog by selecting the *File -> New -> Project...*
4. Ensure that *Java Project* is selected and click *Next*.
5. The dialog should now be prompting for the project name. Enter *Lab3_JFreeChart* in the *Project Name* field.
6. 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 2 below (you need to put the path into which you have unzipped the JFreeChart source files). Click *Finish*.

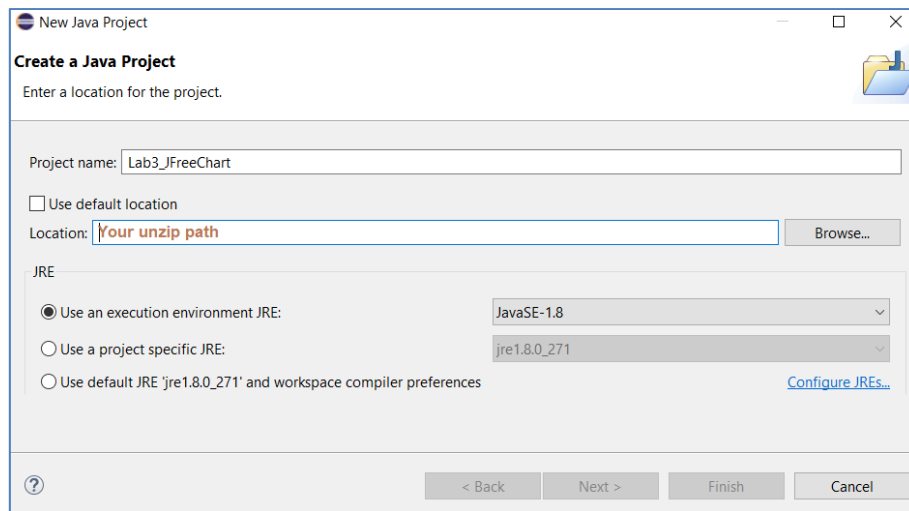


Figure 2 - New Java Project dialog with name and source path filled in

7. The project (SUT) is now set up and ready for testing. Verify that all the sources and reference libraries are shown as in Figure 3 below.

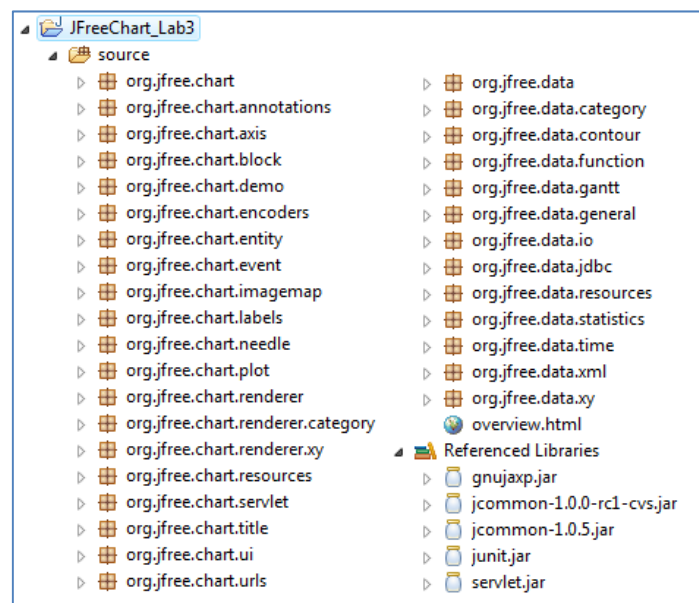


Figure 3 - Packages and archives that should be included in the newly-created project

2.2 IMPORT A TEST SUITE

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. If you no longer have access to this test suite, arrange with the instructor or the TA to retrieve the version of the test suite which you had submitted in Lab 2 for grading.

8. Right click on the `org.jfree.data` package in the *Package Explorer*. Select *Import...*
9. In the *Import* dialog, select the *File System* option (in the *General* category) and click *Next*.
10. 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*.

11. Check your DataUtilities and Range test classes as shown in Figure 4 below. Then click *Finish*.

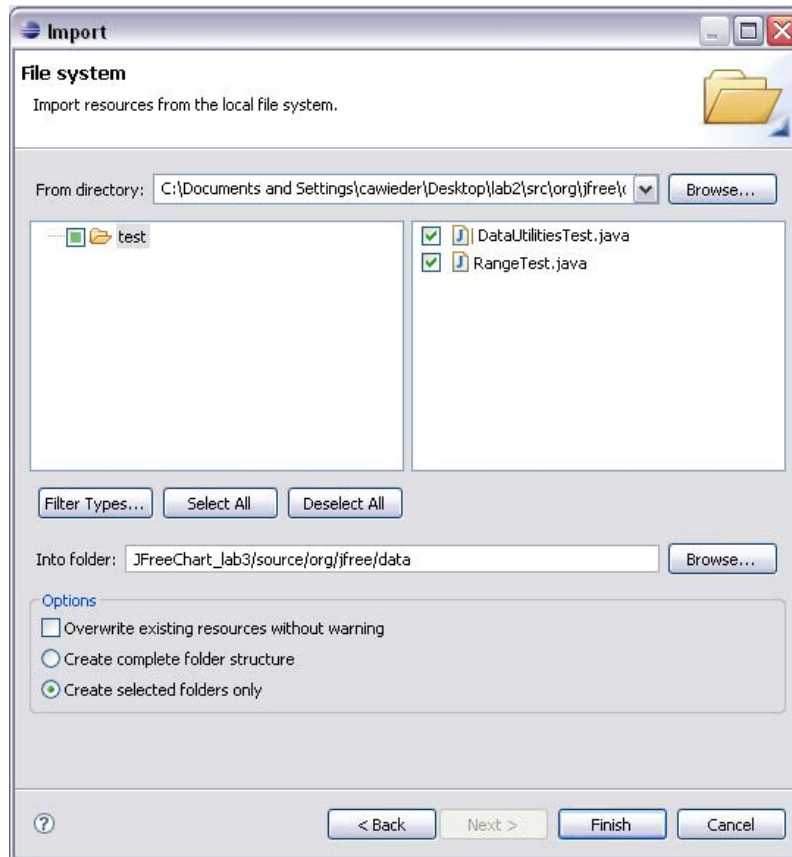


Figure 4 - Import dialog with Lab 2 test classes selected

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 `DataUtilitiesTest.java`) about `@test` labels being unrecognizable, you can easily fix it by adding JUnit4 library to your project, as follows:

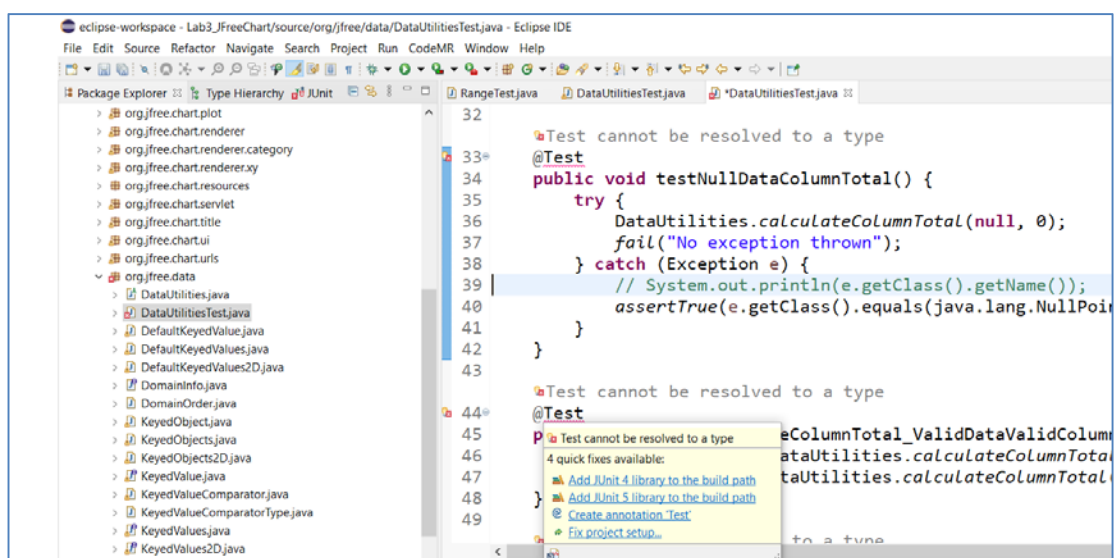


Figure 5 - Adding JUnit4 library to your project, to fix the `@test` errors

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`.

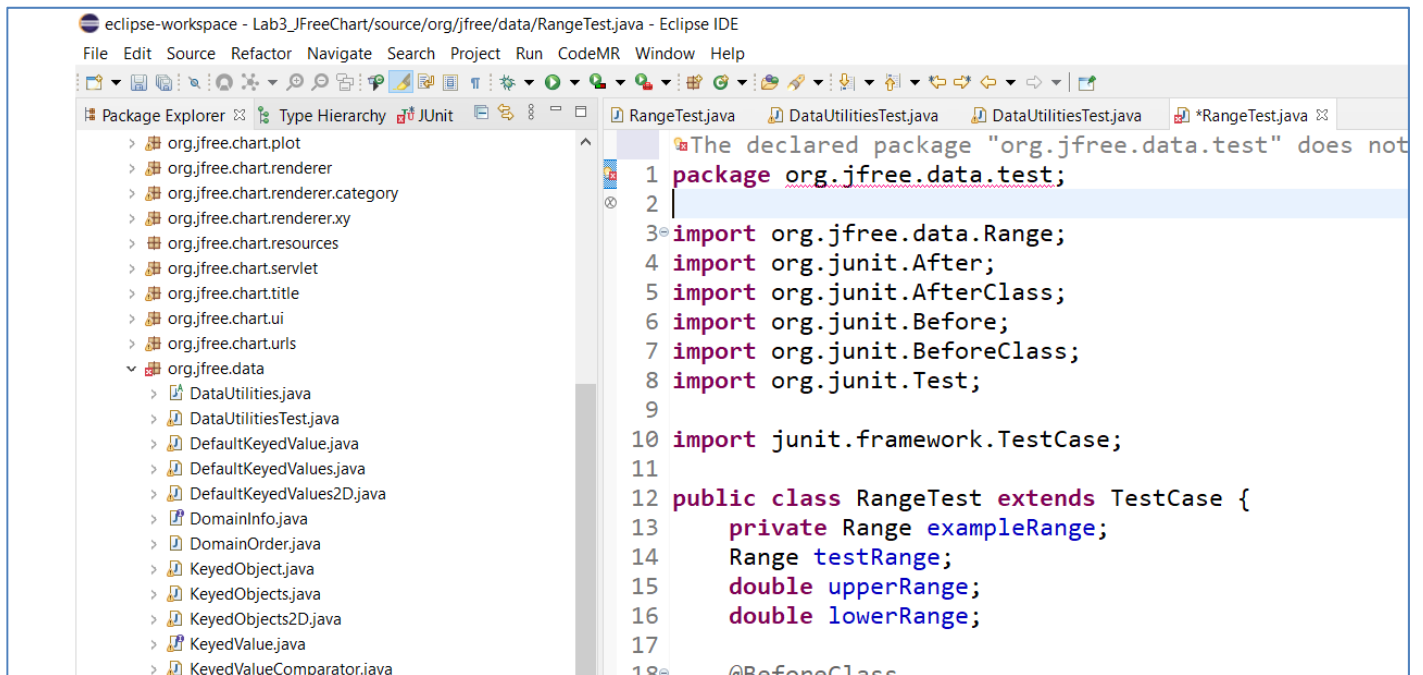


Figure 6 – Fixing the package location identified in top of the two new test files

2.3 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 actually two ways of running JUnit test suites, as shown in Figure 7. The 2nd way is to right click on a single JUnit test class or a folder in Project Explorer and choose “Coverage As” → “JUnit Test”. In this way, the test class(es) will run and also the coverage will be calculated automatically by Eclipse.

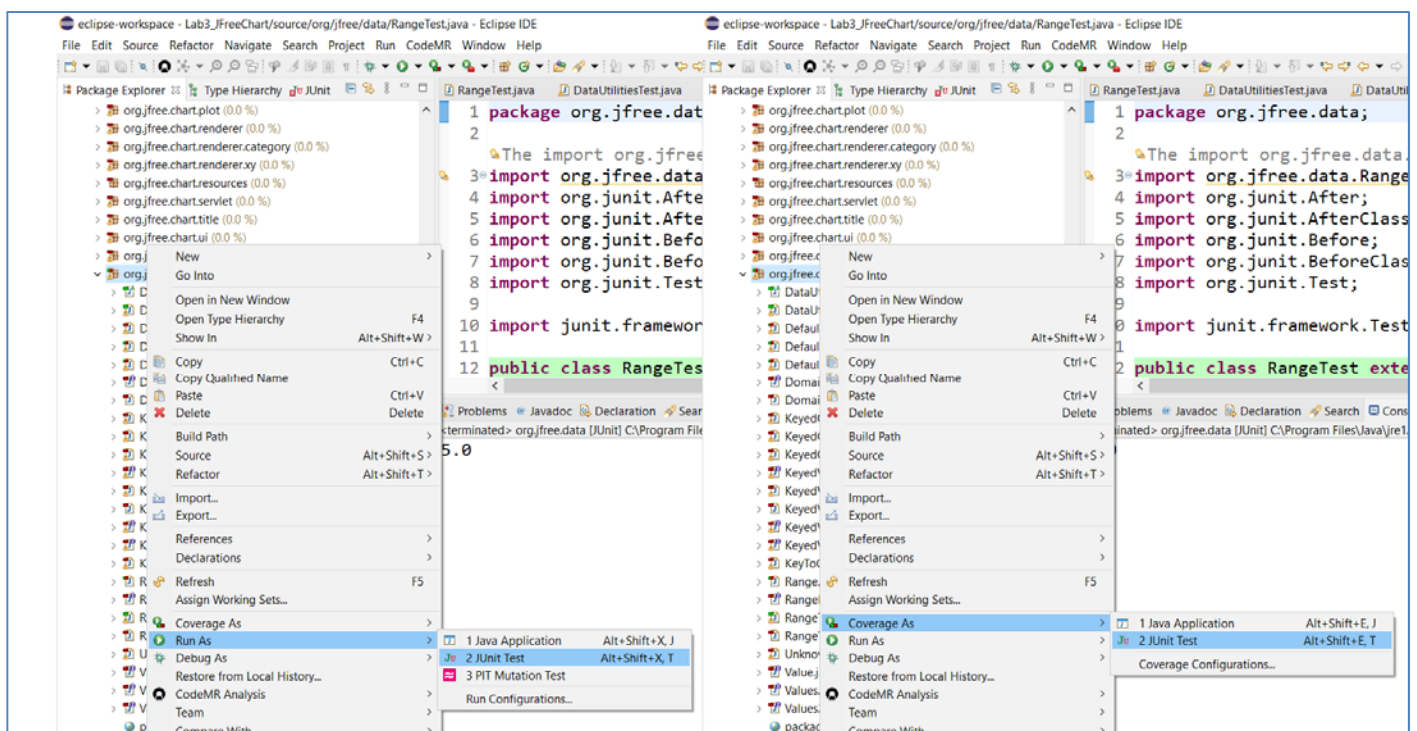


Figure 7 – Two ways of running JUnit test suites

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 8. IF, for whatever reason the “coverage” tab is not automatically shown, you can show it in the screen by following the steps in Figure 9.

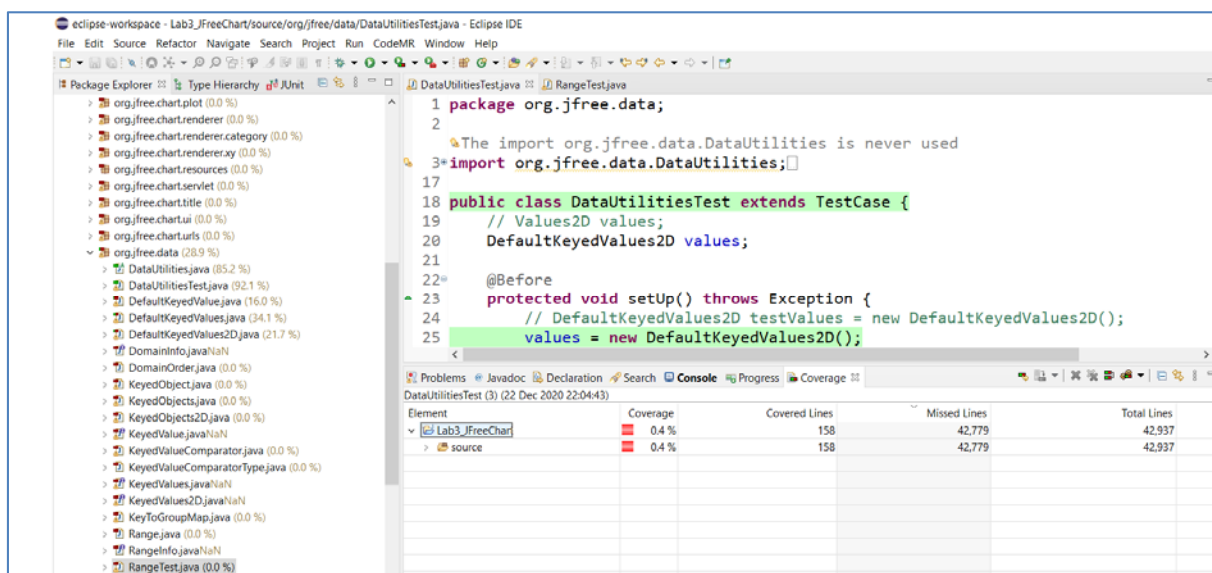


Figure 8 – The “coverage” tab in Eclipse

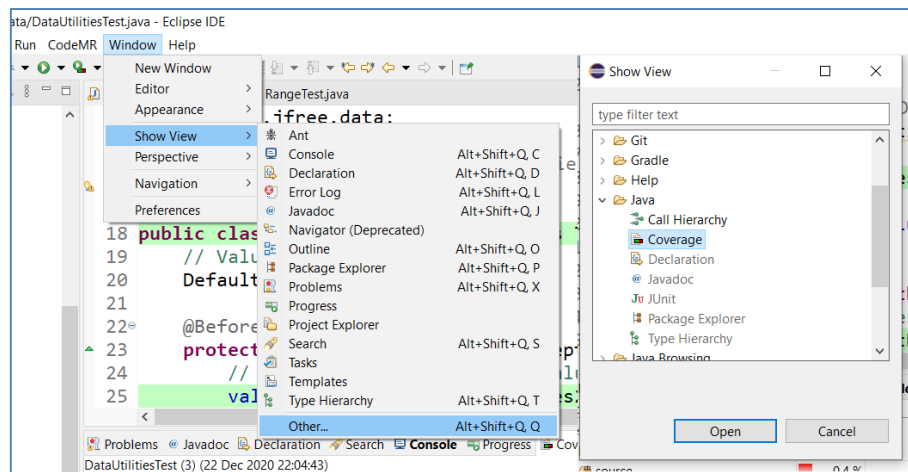


Figure 9 – Showing the “coverage” tab in Eclipse

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 `DataUtilitiesTest.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 10.

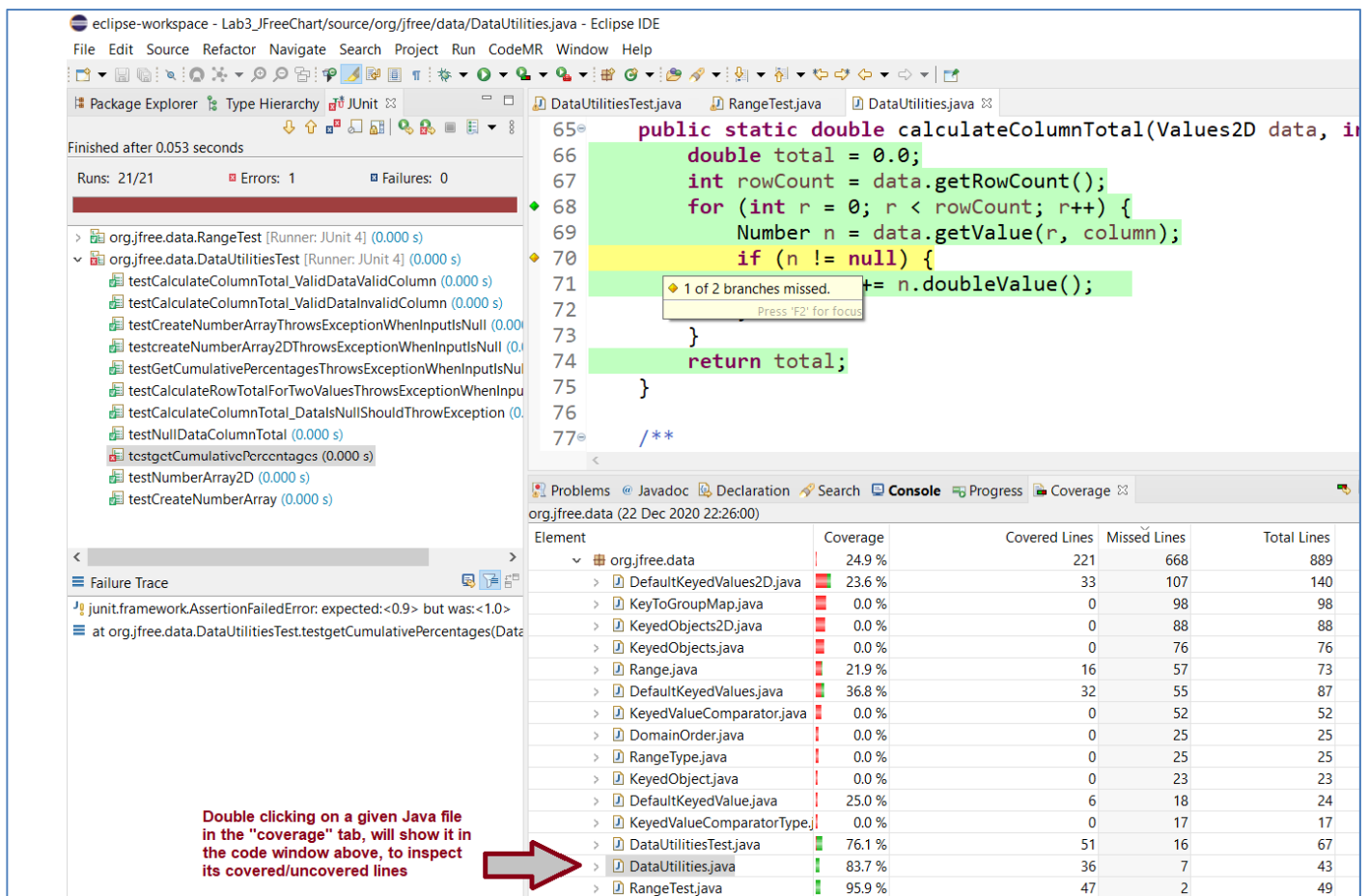


Figure 10 – Inspecting the coverage results after running a JUnit test suite in Eclipse

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 10, we see that the coverage of `DataUtilities.java` class is quite high (83.7%). This is quite expected since we have a specific file (test suite: `DataUtilitiesTest.java`) to test

DataUtilities.java. For the case of Range.java, the coverage is lower (21.9% as seen in Figure 10) 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 DataUtilitiesTest.java is testing DataUtilities.java.

Let us note that, as shown in Figure 10, 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 10 that line 70 inside DataUtilities.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 11. 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.

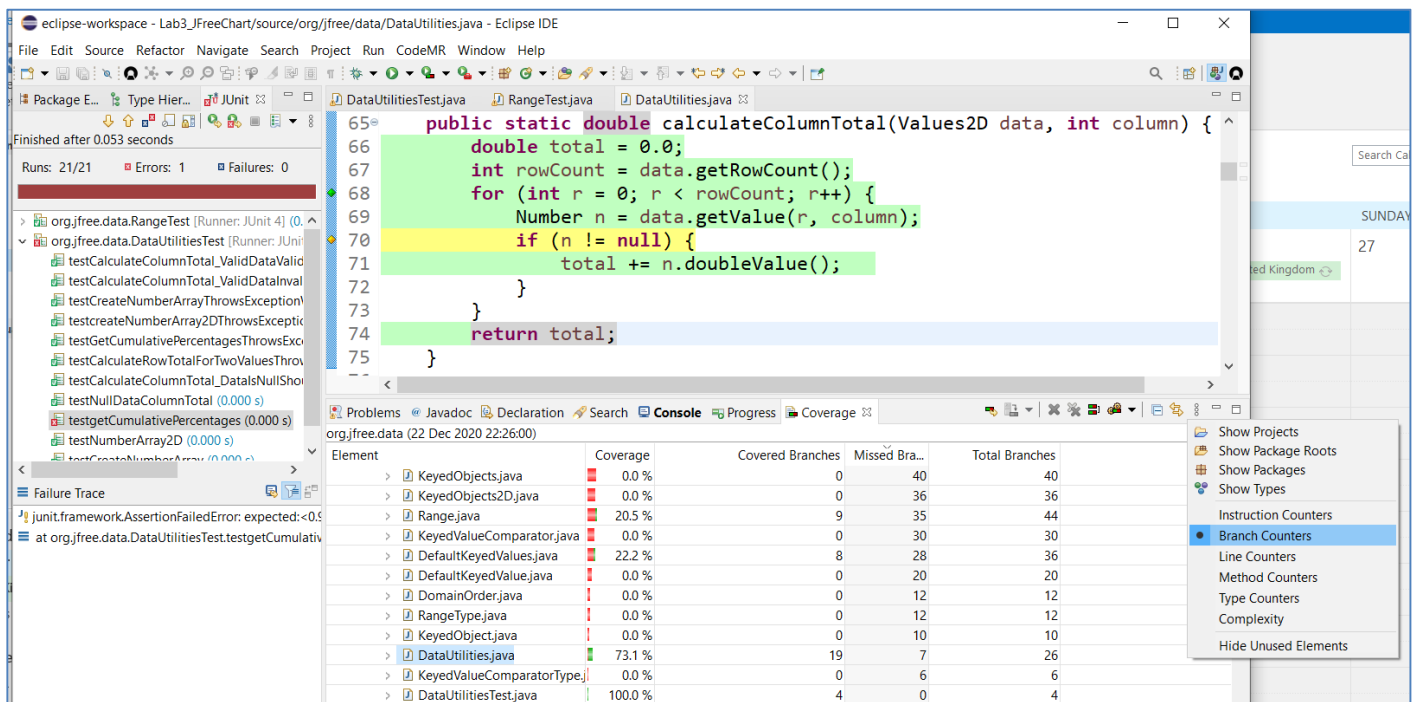


Figure 11 – Switching the type of coverage metrics in the coverage tab: Line coverage, Branch coverage, etc.

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 12)! 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, please do not measure nor report the coverage of test-code itself, in your lab report.

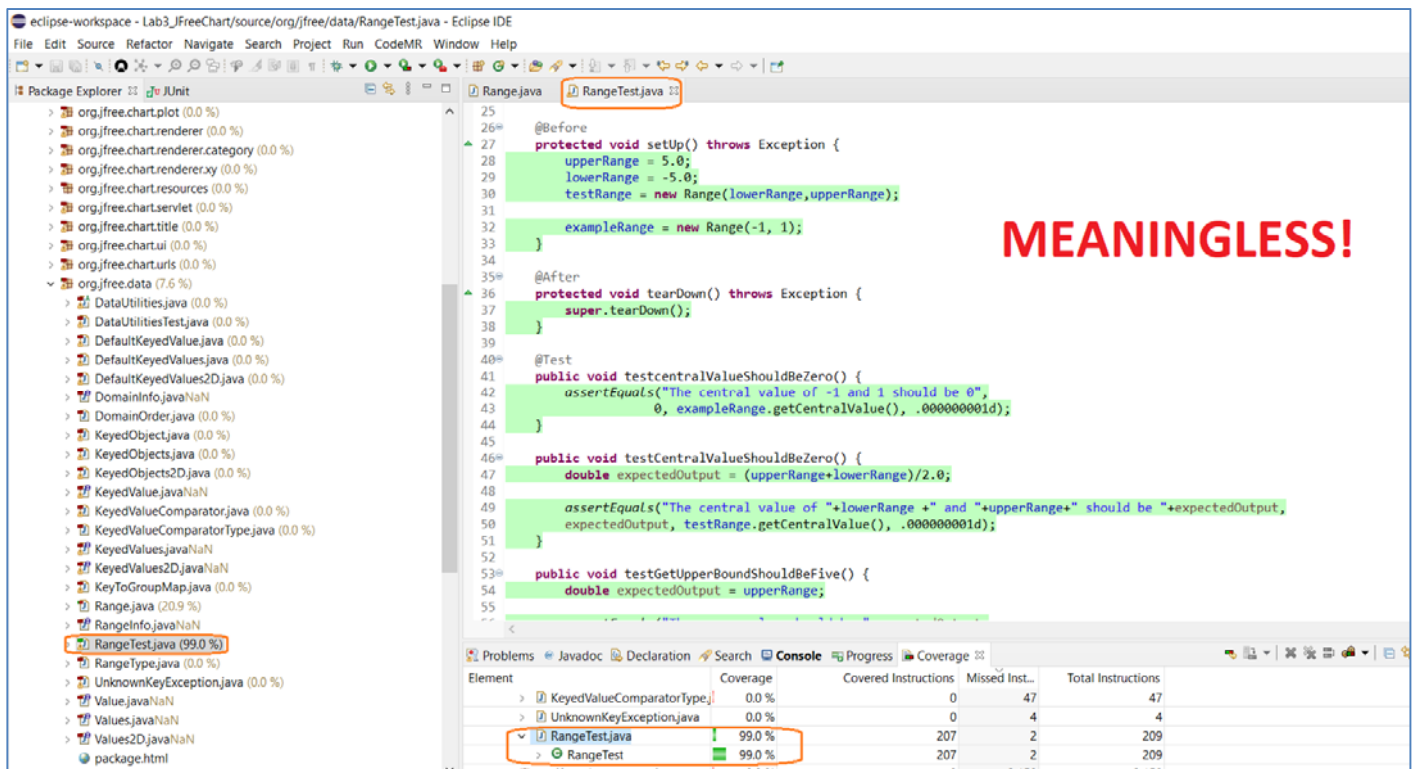


Figure 12 –Measuring, reporting and analyzing the coverage of test-code is meaningless

3 INSTRUCTIONS

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

3.1 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.

12. In this section, 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** on each of `DataUtilities` and `Range` classes
- **Minimum 70% branch coverage** on each of `DataUtilities` and `Range` classes

Note that although the focus in adequacy criteria has changed (it is now on source code), to develop the test cases, the test oracle should always be derived from the requirements (as provided in the Javadocs of the SUT in Lab2).

13. As with any testing to be done, to begin with, you should first develop a test plan. Document this test plan, as it will be included with your lab report. This plan should include:

- information about who (which student member of the group) will design and develop which tests
- how you plan to develop tests to achieve the above adequacy (coverage) criteria

Note: In some rare circumstances, your Lab2 test suite may already be strong enough and when you measure its code coverage, it may be already above the thresholds mentioned above. In that case, you can still further increase the coverage by few percentages above the threshold, by adding a few additional test cases (method), to learn the lab objectives.

14. Carry out your test plan, by developing new test cases (test methods) for the functions of the above two classes, which have low coverage values. For example, we show in Figure 13 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.

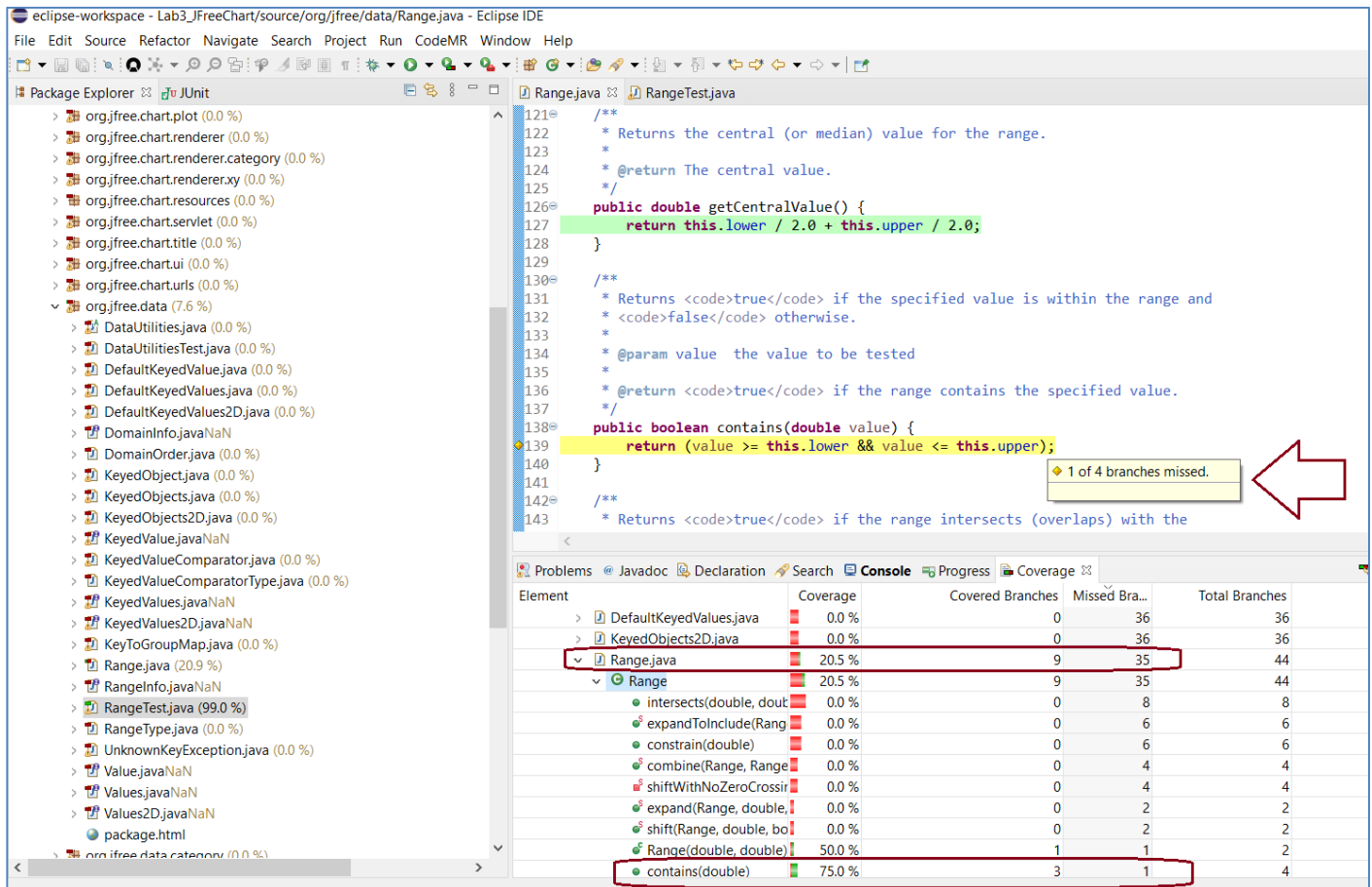
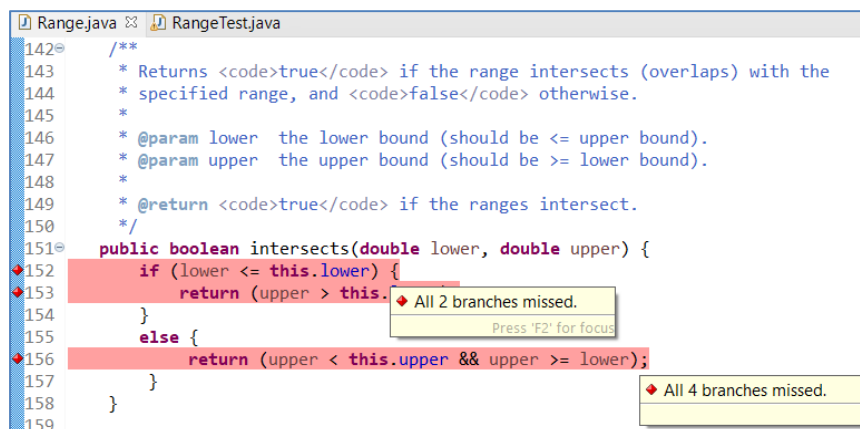


Figure 13 –An example coverage status of the *Range* class: Branch coverage of the entire class is 20.5%



15. 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 13 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 13).
16. For any missed line-of-code or branch, you need to add additional test cases to your test class to cover them. 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.
17. As an effective test engineers, you need to keep each test case (for a single control flow path for example) in a separate method, for example: `testMethodXPositiveValues()` and `testMethodXNegativeValues()`, instead of a single `testMethodX()`. This will help keep test cases consistent, and make the measured coverage metrics more meaningful.
18. 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. But you need to make sure that you do not get any "Errors" in JUnit when you run your test suite, similar to Lab2.
19. 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.
20. 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 (in the screenshot form) in your lab report.

3.2 MEASURE DATA-FLOW COVERAGE MANUALLY

21. 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: For only the `Range.constrain(double)` method of the class `Range` under the `org.jfree.data` package, calculate the data-flow coverage by tracing through the execution of each of your test cases manually. This will need to be included in your report. For an example of how to do this, consult the lecture slides and also see the example done in Appendix A.

4 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 relative difficulty 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.

5 DELIVERABLES AND GRADING

5.1 JUNIT TEST SUITE (40%)

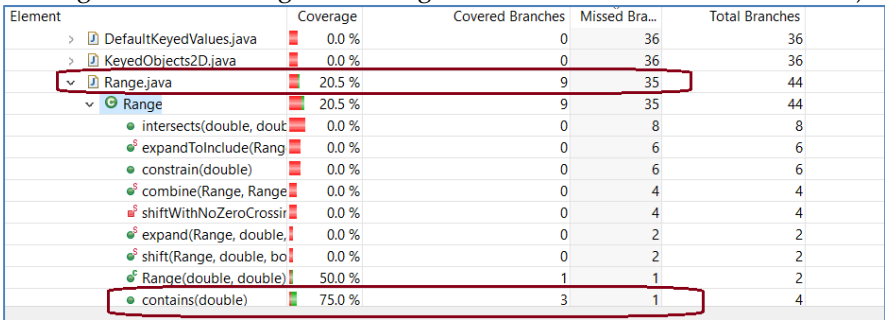
Your entire test suite should be submitted as a single ZIP file along with the lab report. Students will be graded on their unit tests. **Only one submission per group, by either of the two students.**

The grading criteria are as follows:

Marking Scheme	
Achieving code coverage thresholds: lesser coverage than coverage target specified in lab instructions above would decrement your mark proportionally.	30%
Clarity of test-code (is your test code easy to follow, through commenting or style, etc.)	5%
Adherence to requirements: Does your test-code test the SUT code against the requirements (test oracles: assert functions)?	5%

5.2 LAB REPORT (60%)

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 (no PDF etc.).

Marking Scheme	
1-A detailed description of your test plan / test strategy for white-box unit testing	5%
2-A high-level description of five selected test cases you have designed and developed using code-flow coverage information, and how they have increased code coverage (each worth 3%)	15%
3-A detailed report of the coverage achieved of <u>each class and its methods</u> (a screenshot from the code coverage results showing the coverage "bars" with % values would suffice), such as:	10%
 <p>Note: the levels should be above the given threshold.</p>	
4-Manual data-flow coverage calculations for Range.constrain() method	10%
5-A comparison on the advantages and disadvantages of requirements-based test generation (black-box testing) and coverage-based test generation (white-box testing)	15%
6-A discussion on how the team work/effort was divided and managed	2%
7-Any difficulties encountered, challenges overcome, and lessons learned from performing the lab	2%
8-Comments/feedback on the lab itself	1%

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

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/

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]

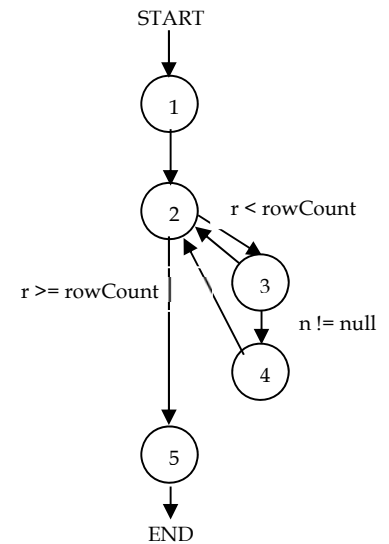
APPENDIX A – EXAMPLE DATA-FLOW COVERAGE CALCULATED MANUALLY

For demonstration purposes, the `DataUtilities.calculateColumnTotal` method is analyzed next. To start with, the code is inspected, and the control/data-flow should be shown in a data-flow graph.

```
public static double calculateColumnTotal(Values2D data, int column) {
    double total = 0.0;
    int rowCount = data.getRowCount();
    for (int r = 0; r < rowCount; r++) {
        Number n = data.getValue(r, column);
        if (n != null) {
            total += n.doubleValue();
        }
    }
    return total;
}
```

From the control-flow graph, 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.

- $dcu(v, i)$ 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	$dcu(data, 1) = \{3\}$ $dcu(column, 1) = \{3\}$ $dcu(total, 1) = \{4, 5\}$	$dpu(rowCount, 1) = \{(2, 3), (2, 5)\}$
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	Total number of def-clear p-use paths to cover: 6

At this stage 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, if for a particular test case, nodes 1, 2, 5 are executed (when the test input is an empty set of Value2D data), then the following definition-clear paths from the above table will be covered:

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

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