

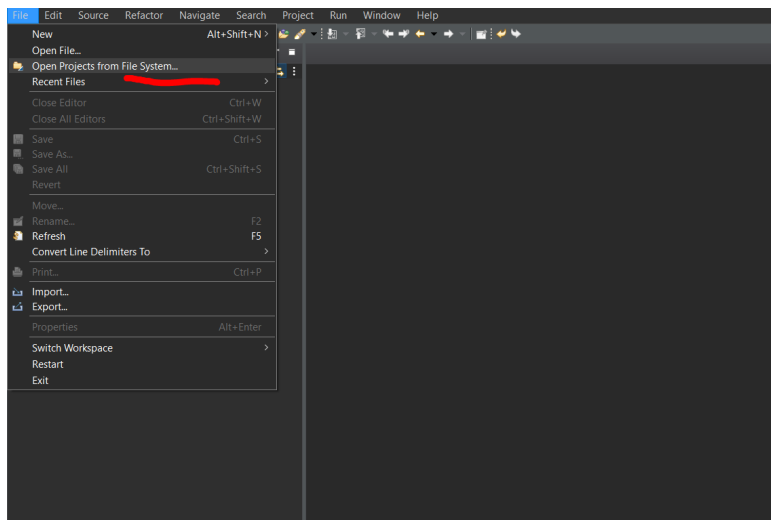
SENG 438 - Software Testing, Reliability, and Quality

Lab. Report #2 – Requirements-Based Test Generation

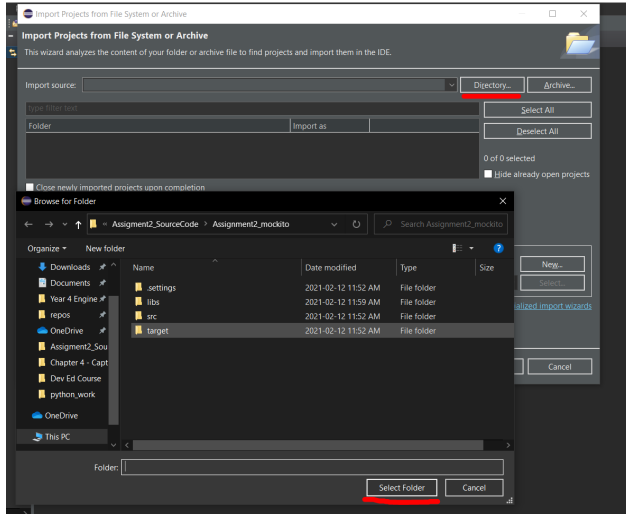
Group #:	24
Student Names:	Tyler Chan
	Takahiro Fujita
	Kyle deBoer
	Caroline Zhou

TO IMPORT THE PROJECT INTO ECLIPSE (SHOULD USE ECLIPSE SINCE WE USED **MAVEN**)

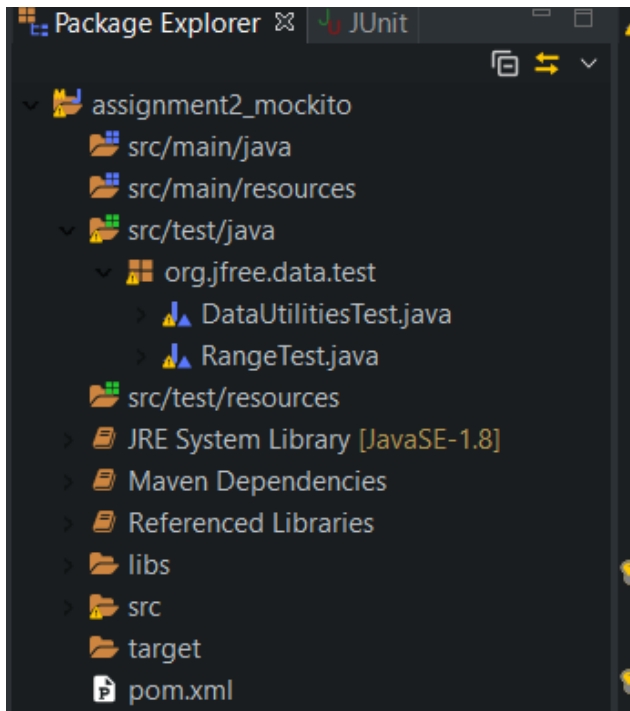
1. DOWNLOAD THE ZIP FILE AND UNZIP IT
2. IN ECLIPSE, CLICK FILE → OPEN PROJECTS FROM FILE SYSTEM



3. WHEN THIS OPENS UP, CLICK ON DIRECTORY WHICH OPEN THE WINDOW FOLDER, AND NAVIGATE TO “Assignment2_mockito” FOLDER AND CLICK SELECT FOLDER, (There is no need to import the default artifacts, they are included in the project’s folder named ‘lib’ with relative path access.



4. Navigate to “src/test/java” folder. Open the “org.jfree.data.test” package. Within it are the two relevant test suites. Run them as a JUnit test



- 5.

1 INTRODUCTION

This second assignment of SENG 438 encourages students to create simple JUnit tests for classes using knowledge gained from class discussions and their own research into different mocking frameworks. The students were to choose testing methods such as “Boundary Value Testing” which were topics covered during lectures in their JUnit test classes.

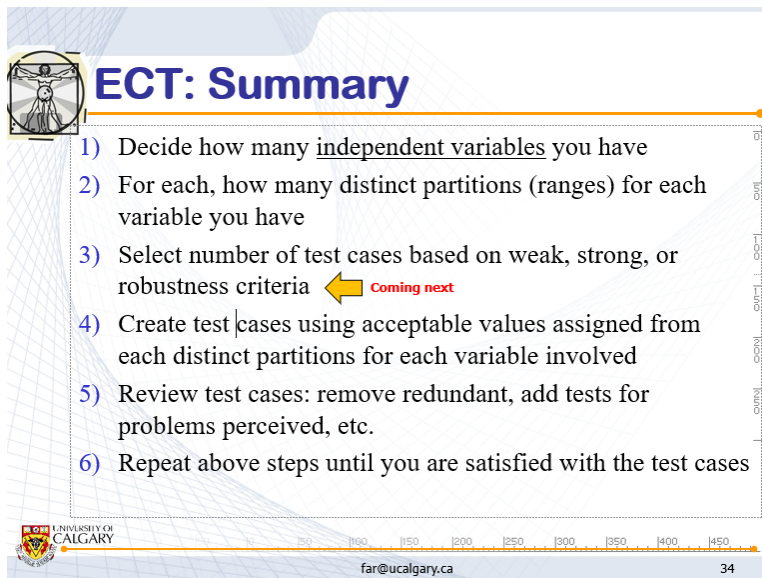
Our group had a general understanding of the basic idea of JUnit testing, and its basic structure (as covered in class). For example, we learned that a JUnit has 3 key sections: setup, where necessary variables are created; test, where Exercise and Verify step is taking place; and teardown, which restores the initial state.

Overall, the lab itself was quite meaningful and gave good practice in writing JUnit tests and implementing mock objects as needed.

2 DETAILED DESCRIPTION OF UNIT TEST STRATEGY

We first researched JUnit testing techniques we could potentially use in this assignment. Here are some of the testing methods we looked into, so we can potentially use these during the test design phase. We explicitly follow the steps of strong equivalence class partitioning technique, boundary values analysis, and robustness technique when designing our test cases for each method. After designing test cases using each technique for each method, we also pruned the test cases if it's needed. This is done to avoid duplication of tests that test the same functionality for a specific method. For a few methods, we did not use the boundary value analysis and robustness techniques because it is not applicable.

Technique 1: Equivalence Class Partitioning <Steps>



ECT: Summary

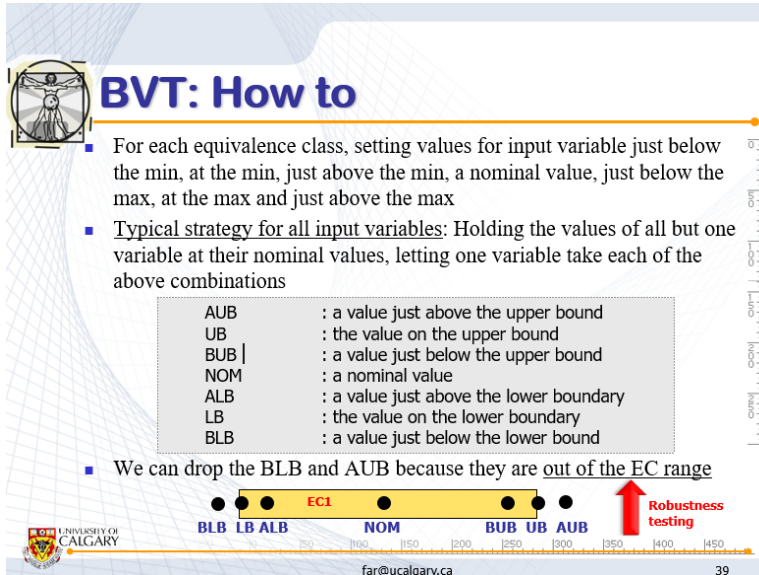
- 1) Decide how many independent variables you have
- 2) For each, how many distinct partitions (ranges) for each variable you have
- 3) Select number of test cases based on weak, strong, or robustness criteria **← Coming next**
- 4) Create test cases using acceptable values assigned from each distinct partitions for each variable involved
- 5) Review test cases: remove redundant, add tests for problems perceived, etc.
- 6) Repeat above steps until you are satisfied with the test cases

UNIVERSITY OF CALGARY
far@ucalgary.ca
34

Technique 2: Boundary Value Testing

(Attempted to come up with test cases within the EC range.)

Ex: boundary condition, inequality and counter<loop>



BVT: How to

- For each equivalence class, setting values for input variable just below the min, at the min, just above the min, a nominal value, just below the max, at the max and just above the max
- Typical strategy for all input variables: Holding the values of all but one variable at their nominal values, letting one variable take each of the above combinations

AUB	: a value just above the upper bound
UB	: the value on the upper bound
BUB	: a value just below the upper bound
NOM	: a nominal value
ALB	: a value just above the lower boundary
LB	: the value on the lower boundary
BLB	: a value just below the lower bound

- We can drop the BLB and AUB because they are out of the EC range

Diagram illustrating the Boundary Value Testing (BVT) process. A horizontal axis represents the input range from 100 to 450. A yellow box labeled 'EC1' covers the range from 100 to 300. Test points are marked on the axis: BLB (below 100), LB (100), ALB (just above 100), NOM (150), BUB (just below 300), UB (300), and AUB (above 300). A red arrow labeled 'Robustness testing' points upwards from the AUB point.

UNIVERSITY OF CALGARY
far@ucalgary.ca
39

Technique 3: Robustness

(Attempted to come up with test cases outside of the EC range, the BLB and AUB conditions.)

Other Techniques That We Considered:

- Worst Case Testing (WCT)
- Decision Table
 - Make sure to select the most “influential” input first for pruning later.
 - Syntax
 - : we don’t care (conditions section)
 - (?): test case is impossible to execute (actions section)
- Combinatorial Testing

And then we went into the structure of the testing method so it’s easier to manage all the test cases among the four team members.

Technical Structure Ideas:

- Split the test cases into different java files based on classes or methods, or keep all the methods for this SUT in one file?
 - RangeTest.java - contains all the test cases for the 5 methods we chose from the Range class.
 - DataUtilitiesTest.java - contains all the test cases for all the methods in DataUtilities class.
- Name convention options

- a. Potential considerations:
 - i. it should express a specific requirement
 - ii. Should include expected input
 - iii. Should include expected result for that input
 - iv. Include the name of the method or class

In the end, for step 1, we chose to have 1 test Class for each Class and that 1 test class has all the testing in it. For step 2, we each experimented with the style that we preferred, some concluded that the naming convention should just express a specific case requirement (Ex. "Case where Double Array Input is empty, but it is not Null"), others considered the state under test and the expected behavior.

From here we wrote down the specific test for each method in DataUtilitiesClass and the 5 randomly selected Range methods. We mainly stuck to using the Equivalent Class Partitioning method for this assignment, but used other methods where they were needed to test every potential problem within the method.

3 TEST CASES DEVELOPED

Class: DataUtilities

Method: calculateColumnTotal

- Equivalence class partitioning
 - Inputs
 - Data <data type: Values2D>
 - (E) because the number of rows and columns may range between 1-inf.
 - Column only has one value, see case above
 - Column has multiple values, $[1, 2, 3, 4, 5]^T$
 - (U) null
 - Column <data type: int>
 - (E) valid column int with the given data
 - (U) invalid column int with the given data
 - Cases (strong ECP with pruning)
 1. Data's column only has one value with a valid column int
 - Example: $([1], 0)$
 - Expects 1
 2. Data's column only has one value with a invalid column int
 - Example: $([1], 2)$
 - Expects 0
 3. Data's Column has multiple values with a valid column int,
 - $([1, 2, 3,]^T, 0)$
 - Expects sum of 6
 4. Data = null, valid column int / invalid column int

- (null, any number)
- Expects NullPointerException

Method: calculateRowTotal

- Equivalence class partitioning
 - Inputs
 - Data <data type: Values2D>
 - (E) because the number of rows and columns may range between 1-inf.
 - row only has one value, see case above
 - row has multiple values, [1, 2, 3, 4, 5]
 - (U) null
 - Row <data type: int>
 - (E) valid row int with the given data
 - (U) invalid row int with the given data
 - Cases (strong ECP with pruning)
 1. Data's row only has one value with a valid row int
 - ([1], 0)
 - Expects 1
 2. Data's row only has one value with a invalid row int
 - ([1], 2)
 - Expects 0?
 3. Data's row has multiple values with a valid row int,
 - $([1, 2, 3, 4, 5]^T, 0)$
 - Expects sum of 15
 4. Data's row has multiple values with a invalid row int
 - Don't need this, functionality is tested in the second case
 5. Data = null, valid row int / invalid row int
 - (null, any number)
 - Expects InvalidParameterException

Method: createNumberArray

- Equivalence class partitioning
 - Inputs
 - Data <data type: double []>
 - (E) array of double that is not null
 - (E) array = []
 - (U) array = null
 - Cases
 1. Array of double that is not null
 - Expects an array of Number objects
 2. Array of double that is not filled, but not null
 - Expect an empty array of Number objects

3. Array = null
 - Expects InvalidParameterException

Method: createNumberArray2D

- Equivalence class partitioning
 - Inputs
 - Data <data type: double [] []>
 - (E) valid 2D array
 - (E) Array = [] []
 - (U) array = null
 - Cases
 1. Valid 2D array that is filled and is not null
 - Expects an array of Number objects
 2. Valid 2D Array that is not filled but not null
 - Expects an empty but non-null array of Number objects.
 3. Array = null
 - Expects IllegalArgumentException

Method: getCumulativePercentages

- Equivalence class partitioning
 - Inputs
 - Data <data type: KeyedValues>
 - (E) valid key value pairs that are not null, and all values are not null either - test case 1
 - (U) division by zero - test case 2
 - Expects the not a number as the value for all key value pairs, because the divisor is 0.
 - (U) data = null - test case 3
 - Expects IllegalArgumentException
 - Cases (see above)
- Robustness
 - valid key value pair that is not null, and there exists a null value in one of the pairs. - test case 4

Class: Range

Method: contains

- Equivalence class partitioning

- Input
 - value <data type: double>
 - (E) within range double - case 1
 - Expects true
 - (E) out of range double - case 2
 - Expects false
 - (U) null
 - Cases (see above)
- **Boundary Value Testing (Expected Outcome = True)**
 - (LB)) Value input equals to the lower boundary - case 3
 - (UB) Value input equals to the upper boundary - case 4
 - (ALB) Value input just above the lower boundary - case 5
 - (BUB) value input just below the upper boundary - case 6
- **Robustness Testing (Expected Outcome = False)**
 - (BLB)) Value input just below the lower boundary - case 7
 - (AUB) Value input just above the upper boundary - case 8
- Case 1-2: ECP
- Case 3-6: BVT
- Case 7-8: Robustness

Method: equals

- Equivalence class partitioning
 - Input
 - java.lang Object
 - (E) Range object with the same lower and upper bounds
 - Expects true
 - (E) Range object with the same lower bounds, and different upper bounds
 - Expects false
 - (E) not a range object
 - String object
 - Expect exceptions?
 - (E) null
 - (U) not an object
 - Cases
 - Boundary Value Testing
 - Comparing two range objects
 - with the same lower and upper bounds
 - Same lower bound and upper bound is <ex: 0.0001> more(AUB)
 - Same upper bound and lower bound is <ex: 0.0001> less(BLB)
 - Same lower bound and upper bound is <ex: 0.0001> less(BUB)
 - Same upper bound and lower bound is <ex: 0.0001> more(ALB)

Method: getUpperBound

- Equivalence class partitioning
 - Test Cases
 - Range object = null
 - Expects NullPointerException
 - Range object's upper boundary is bigger than the lower boundary
 - Example: (lower boundary, upper boundary) = (1.0, 5.0)
 - Expects: 5.0
 - Failed
 - Actual return value: 1.0
 - Range object's upper boundary equals to the lower boundary
 - Example: (lower boundary, upper boundary) = (1.0, 1.0)
 - Expects: 1

Method: getCentralValue

- Equivalence class partitioning
 - Test Cases
 - When the upper boundary is bigger than the lower boundary
 - Example: (lower, upper) = (1,3)
 - Expects: 2.0
 - When the upper boundary equals to the lower boundary
 - Example: (lower, upper) = (1,1)
 - Expects: 1.0
 - When the upper boundary is bigger than the lower boundary, and the lower boundary is a negative double
 - Example: (lower, upper) = (-1,1)
 - Expects: 0.0
 - When the upper and lower boundary is 0
 - Example: (Upper, lower) = (0,0)
 - Expects: 0.0
 - When the range object equals to null
 - Expects NullPointerException
 - Cases
- Boundary Value Testing
 - Not applicable

Method: intersects

- Equivalence class partitioning
 - Input
 - Lower <data type: double>

- (E) valid input (within the range)
 - (U) null
 - Upper <data type: double>
 - (E) valid input (within the range)
 - (U) null
- Cases
 - Assuming range object = (upper,lower) = (1,5)
 1. Valid lower input with valid upper input
 - (1,3)
 2. Lower bound > other's lower bound
 - (2,5)
 3. Upper bound < other's upper bound
 - (0, 4)
 4. Lower bound is greater than the other's upper bound
 - (6,9)
 5. Upper bound is less than the other's lower bound
 - (-5,0)
- Boundary Value Testing
 - Lower and upper bounds equal to other's lower and upper.
 - (1,5)
 - Upper and lower input is the same, within range
 - Upper and lower input is the same, out of range
- Robustness
 - Lower parameter is higher than the upper parameter value
 - (5,1)

4 HOW THE TEAM WORK/EFFORT WAS DIVIDED AND MANAGED

Our group worked together through the initial steps leading up to and including the step where we wrote the JUnit test for "calculateColumnTotalForTwoValues()". After that, one of our members wrote up part of the Test Plan and Cases and had the rest peer review/come up with other test cases. After that, we once again splitted up into pairs to work on the JUnit tests for some of the harder methods and we each did the other, much easier, methods individually. After that, we all merged our findings/code into two Test Java files (one for Range and one for DataUtilities).

5 DIFFICULTIES ENCOUNTERED, CHALLENGES OVERCOME, AND LESSONS LEARNED

Not too many difficulties were encountered. However, there was a single recurring issue that arose due to the use of Mockito and Maven to write the JUnit test.

Firstly, using Mockito led to the use of a Maven dependency. This in and of itself was already difficult to set up. When one member was able to successfully create and set up a Maven project, the others had great difficulty, encountering many seemingly arbitrary errors that, once resolved, gave way to more errors, greatly hindering progress in this assignment. Much research had to be done, and errors had to be slowly cleared.

Secondly, the configuration file seemed to have troubles working on other devices. For example, a common issue that arose was a “class not found exception”. These errors were not present on some devices, but they were on others. Some random solutions from the internet and several class path configuration changes were finally able to overcome the errors.

The biggest lessons learned is that all machines are different, and the reliance on the configuration settings of another device should be avoided. In other words, just because this setup works on someone else’s computer, doesn’t mean it will work on yours.

During the test case design phase, we also face some difficulties categorizing a few test cases, because these are not the result of using either the equivalent class partitioning nor the boundary value testing techniques. We feel the need to include these tests as they will help to uncover some of the potential bugs within the program. The closest category for these outliers is robustness, because these test cases usually include some type of invalid inputs that will usually throw an exception. When we implemented the nominal test case for `getCumulativePercentages` method (see `When_keyValuesIsNotNull_Expect_correctCumulativePercentage` method), we encountered difficulties setting up the mock object for the method. While we do not have the code or any information as to which `KeyedValues` methods were used in the method `getCumulativePercentages()`. We had to learn the `getCumulativePercentages` method through try and error. At the end, we had to set up return values for all the methods in the `KeyedValues` class, and one other return method in the `KeyedValues`’ parent class. Through this process, we learned the importance of system knowledge which could have helped to fasten the process. Through this assignment, we noticed a few drawbacks of using mocking despite the benefit of isolating tests from their dependencies, which helped us to write concise tests. As we do not know the exact information as to how the method under test is implemented, so we had to incorporate our method with many assumptions as to how the components in this system interact with each other. If any behavior got changed in the return methods in the mock object for one test, we also have to change the testing code accordingly. Therefore, it is crucial to minimize the knowledge duplication when mocking.

6 COMMENTS/FEEDBACK ON THE LAB ITSELF

This lab was way more straight forward than the last lab. There was close to no confusion on what we were supposed to do. It was also very convenient/helpful to have an example JUnit test case code we could follow and see before doing our own. Being able to choose your own mocking framework was also very convenient, though we believe that providing us/showing us a sample framework we could use for this assignment is also convenient since we won’t be left in the dark if we didn’t know any mocking framework we could use.