SENG 637 - Software Testing, Reliability, and Quality

Lab. Report #2 – Requirements-Based Test Generation

Group: 9
Moaz Barakat
Juan Celis
Bogdan Constantinescu
Andy Guevara
Billy Sidharta

1 Introduction

This assignment is about understanding the basics of automated testing through practical experience. In this assignment specifically, the system under test (SUT) is a modified version of JFreeChart which is an open-source Java framework for charts. We are required to create automated test code using JUnit as well as incorporate and work with mock objects in test code development.

The following were used to complete this assignment:

- IDE: Eclipse (2023)
- Unit testing: JUnit 4.x
- Mocking: JMock

Since we are given the requirements only, we applied the recommended black-box test-case design techniques like equivalence class partitioning and boundary value analysis.

What we knew before this assignment:

Black-box (BB) is a software specification-based testing method that examines the functionality of an application without knowing or understanding the implementation. The goal is to generate tests from system requirements. The benefits of using this method is that it can identify missing functionality of a system. However, the drawback is that it fails to identify unexpected functionality of the system.

Equivalence Class Testing (ECT) is a systematic and efficient Black-box test case generation technique. It works by dividing the input set into partition that can be considered the same. In this technique, input sets can generally be divided into all the Expected or legal inputs (E) and all Unexpected (illegal) inputs (U). Additionally, these can also be further subdivided into subsets as required (E1, U1, etc.).

Boundary Value Testing (BVT) is another systematic and efficient BB test case generation technique similar to equivalence class partitioning. The underlying idea is that programmers make mistakes in processing values at and near the boundaries of equivalence classes. It works by focusing on tests at and near the boundaries of equivalence classes.

It is important to outline that tests derived using either of the two techniques may overlap. In this lab, the two classes that are required to be tested are:

- org.jfree.data.Range (in the package org.jfree.data)
- org.jfree.data.DataUtilities (in the package org.jfree.data)

2 Detailed description of unit test strategy

Range Class

Methods

We are required to test 5 out of the 15 methods for org.jfree.data.Range. Below are the 5 methods we decided to test along with their API specifications in Javadoc:

- **double constrain(double value)**: Returns the value within the range that is closest to the specified value.
- **boolean contains(double value)**: Returns true if the specified value is within the range and false otherwise.
- **boolean equals(java.lang.Object obj)**: Tests this object for equality with an arbitrary object.
- **double getCentralValue()**: Returns the central (or median) value for the range.
- **double getLength()**: Returns the length of the range.

The following sections outline the unit test strategy for each method

Note: For analysis and demonstration, we can consider an example range defined by:

- lower bound of 3
- upper bound of 7

constrain

- **Description**: Returns the value within the range that is closest to the specified value.
- **Input**: [value:double] The value to find the closest in-range value of
- **Output**: [double] The constrained value. If value is within the range, will return the input value

Number input variables: 1

Input domain & Equivalent classes: 4

- `E1` = {value: range.lower <= value <= range.upper} //3,4,5,6,7
- `E2` = {value: < range.lower} //2,1,-5 etc.
- `E3` = {value: > range.upper} //8,9,15 etc.
- `U1` = {value: non-numbers, symbols, etc.} //'ab' etc.

Boundary value analysis:

- `BLB` : 2
- `LB` : 3
- `ALB` : 4

- NOM: Anything between 4 and 6
- BUB : 6
- UB : 7
- AUB : 8

Extreme Values:

- Double.MAX_VALUE
- Double.MIN_VALUE

contains

- **Description**: Returns true if the specified value is within the range and false otherwise.
- **Input**: [value:double] The value to be tested
- **Output**: [boolean] true if the range contains the specified value

Number input variables: 1

Input domain & Equivalent classes: 3

- E1 = {value: range.lower <= value <= range.upper} //3,4,5,6,7
- E2 = {value: value < range.lower || value > range.upper} //2,1,-5,8,9,15 etc.
- U1 = {value: non-numbers, symbols, etc.} //'ab' etc.

Boundary value analysis:

- BLB : 2
- LB : 3
- ALB : 4
- NOM: Anything between 4 and 6
- BUB : 6
- UB : 7
- AUB : 8

Extreme Values:

- Double.MAX_VALUE
- Double.MIN_VALUE

equals

- **Description**: Tests this object for equality with an arbitrary object.
- **Input**: [obj:java.lang.Object] the object to test against (null permitted)
- **Output**: [double] true if the input object is an equivalent range

Number input variables: 1

Input domain & Equivalent classes: 3

- E1 = {obj: obj instanceof Range && obj.lower=range.lower && obj.upper=range.upper} //true
- E2 = {obj: obj not an instanceof Range || obj.lower!=range.lower || obj.upper!=range.upper} //false
- E3 = {obj: non-numbers, symbols, etc.} //'ab' etc.

Boundary value analysis:

- obj.lower = range.lower, obj.upper != range.upper
- obj.lower != range.lower, obj.upper = range.upper
- obj.lower != range.lower, obj.upper != range.upper
- obj.lower = range.lower, obj.upper = range.upper

Extreme Values:

- N/A

getCentralValue

- **Description**: Returns the central (or median) value for the range.
- **Input**: None
- **Output**: [double] The central value

Number input variables: `0`

Input domain & Equivalent classes: `2`

- `E1` = {central/median value given a valid range: range.upper>=range.lower} //
- `U1` = {range.upper<range.lower} //

Boundary value analysis: Although there are no input variables, we can still test different boundaries of ranges

- Positive ranges: (lower=+,upper=+)
- Negative ranges: (lower=-,upper=-)
- Mixed ranges: (lower=-,upper=+)
- Same ranges: (lower=upper,upper=lower)

getLength

- **Description**: Returns the length of the range.
- **Input**: None
- **Output**: [double] The length

Number input variables: `0`

Input domain & Equivalent classes: `2`

- `E1` = {length given a valid range: range.upper>=range.lower} //
- `U1` = {range.upper<range.lower} //

Boundary value analysis: Although there are no input variables, we can still test different boundaries of ranges

- Positive ranges: (lower=+,upper=+)
- Negative ranges: (lower=-,upper=-)
- Mixed ranges: (lower=-,upper=+)
- Same ranges: (lower=upper,upper=lower)

Extreme Values:

- (lower=Double.MAX_VALUE,upper=Double.MAX_VALUE)

DataUtilities Class

Methods

We are required to test 5 out of the 5 methods for org.jfree.data.DataUtilities. Below are the 5 methods test along with their API specifications in Javadoc:

- **double calculateColumnTotal(Values2D data, int column)**: Returns the sum of the values in one column of the supplied data table.
- **double calculateRowTotal(Values2D data, int row)**: Returns the sum of the values in one row of the supplied data table.
- **java.lang.Number[] createNumberArray(double[] data)**: Constructs an array of Number objects from an array of double primitives..
- ** java.lang.Number[][] createNumberArray2D(double[][] data)**: Constructs an array of arrays of Number objects from a corresponding structure containing double primitives.
- **KeyedValues getCumulativePercentages(KeyedValues data)**: Returns a KeyedValues instance that contains the cumulative percentage values for the data in another KeyedValues instance..

The following sections outline the unit test strategy for each method

calculateColumnTotal

- **Description**: Returns the sum of the values in one column of the supplied data table. With invalid input, a total of zero will be returned.
- **Input**: [data:Values2D] the table of values (null not permitted), [column:Values2D] the column index (zero-based).
- **Output**: [double] The sum of the values in the specified column.
- **Throws**: [InvalidParameterException] - if invalid data object is passed in.

Number input variables: 2

Input domain & Equivalent classes: 6

- E1 = {column: 0 <= column < data.getColumnCount()}
- E2 = {data: valid data}
- U1 = {column: column <0}
- U2 = {column: column >= data.getColumnCount()}
- U3 = {data: invalid data}

Boundary value analysis:

- Column = 0 [LB]
- Column = -1 [BLB]
- Column = data.getColumnCount() - 1 [UB]

- Column = data.getColumnCount() [AUB]
- Column = >0 <data.getColumnCount() [NOM]

Extreme Values:

- N/A

calculateRowTotal

- **Description**: Returns the sum of the values in one row of the supplied data table. With invalid input, a total of zero will be returned.
- **Input**: [data:Values2D] the table of values (null not permitted), [row:Values2D] the row index (zero-based).
- **Output**: [double] The total of the values in the specified row.
- **Throws**: [InvalidParameterException] - if invalid data object is passed in.

Number input variables: 2

Input domain & Equivalent classes: 6

- E1 = {row: 0 <= row < data.getRowCount()}
- E2 = {data: valid data}
- U1 = {row: row <0}
- U2 = {row: row >= data.getRowCount()}
- U3 = {data: invalid data/null}

Boundary value analysis:

- row = 0 [LB]
- row = -1 [BLB]
- row = data.getRowCount() - 1 [UB]
- row = data.getRowCount() [AUB]
- row = >0 <data.getRowCount() [NOM]

Extreme Values:

- N/A

createNumberArray

- **Description**: Constructs an array of Number objects from an array of double primitives.
- **Input**: [data:double[]] An array of double primitives (null not permitted).
- **Output**: [java.lang.Number[]] An array of Number objects..
- **Throws**: [InvalidParameterException] - if invalid data object is passed in.

Number input variables: 1

Input domain & Equivalent classes: 2

- E1 = {data: valid/not null}
- U1 = {data: invalid/null}

Boundary value analysis:

- length = 0
- length > 0

Extreme Values:

- N/A

createNumberArray2D

- **Description**: Constructs an array of arrays of Number objects from a corresponding structure containing double primitives.
- **Input**: [data:double[]] An array of double primitives (null not permitted).
- **Output**: [java.lang.Number[]] An array of Number objects..
- **Throws**: [InvalidParameterException] - if invalid data object is passed in.

Number input variables: 1

Input domain & Equivalent classes: 2

- E1 = {data: valid/not null}
- U1 = {data: invalid/null}

Boundary value analysis:

- length = 0
- length > 0
- length[0] = 0
- length[0] > 0

Extreme Values:

- N/A

getCumulativePercentages

- **Description**: Returns a KeyedValues instance that contains the cumulative percentage values for the data in another KeyedValues instance. The cumulative percentage is each value's cumulative sum's portion of the sum of all the values.
- **Input**: [data:KeyedValues] the data (null not permitted).
- **Output**: [KeyedValues] The cumulative percentages.
- **Throws**: [InvalidParameterException] - if invalid data object is passed in.

Number input variables: 2

Input domain & Equivalent classes: 6

- E1 = {data: valid/not null, data.value should be numbers}
- U1 = {data: invalid, data.value non numbers}
- U2 = {data: invalid/null}

3 Test cases developed

Range Class

constrain

Based on the test strategy section above, we decided to include the following 8 tests:

Using example test Range(3,7):

constrainValueOfFiveShouldBeFive:

- Test Case ID: `SECT1`
- Input value: `5.0`
- Relevant Condition(s): `E1; NOM`
- Expected Output: `5.0`

constrainValueOfThreeShouldBeThree:

- Test Case ID: `SECT2`
- Input value: `3.0`
- Relevant Condition(s): `E1; LB`
- Expected Output: `3.0`

constrainValueOfSevenShouldBeSeven:

- Test Case ID: `SECT3`
- Input value: `7.0`
- Relevant Condition(s): `E1; UB`
- Expected Output: `7.0`

constrainValueOfTwoShouldBeThree:

- Test Case ID: `SECT4`
- Input value: `2.0`
- Relevant Condition(s): `E2; BLB`
- Expected Output: `3.0`

constrainValueOfEightShouldBeSeven:

- Test Case ID: `SECT5`
- Input value: `8.0`
- Relevant Condition(s): `E3; AUB`
- Expected Output: `7.0`

constrainValueOfDoubleMinValueShouldBeThree:

- Test Case ID: `SECT6`
- Input value: `Double.MIN_VALUE`
- Relevant Condition(s): `E2; BLB`
- Expected Output: `3.0`

constrainValueOfDoubleMaxValueShouldBeSeven:

- Test Case ID: `SECT7`
- Input value: `Double.MAX_VALUE`
- Relevant Condition(s): `E3; AUB`
- Expected Output: `7.0`

constrainValueOfDoubleNanShouldBeDoubleNan:

- Test Case ID: `SECT8`
- Input value: `Double.NaN`
- Relevant Condition(s): `U1`
- Expected Output: `true`

contains

Based on the test strategy section above, we decided to include the following 8 tests:

Using example test Range(3,7):

containsValueOfFiveShouldBeTrue:

- Test Case ID: `SECT9`
- Input value: `5.0`
- Relevant Condition(s): `E1; NOM`
- Expected Output: `true`

containsValueOfThreeShouldBeTrue:

- Test Case ID: `SECT10`
- Input value: `3.0`
- Relevant Condition(s): `E1; LB`
- Expected Output: `true`

containsValueOfSevenShouldBeTrue:

- Test Case ID: `SECT11`
- Input value: `7.0`
- Relevant Condition(s): `E1; UB`
- Expected Output: `true`

containsValueOfTwoShouldBeFalse:

- Test Case ID: `SECT12`
- Input value: `2.0`
- Relevant Condition(s): `E2; BLB`
- Expected Output: `false`

containsValueOfEightShouldBeFalse:

- Test Case ID: `SECT13`
- Input value: `8.0`

- Relevant Condition(s): `E2; AUB`
- Expected Output: `false`

containsValueOfDoubleMinValueShouldBeFalse:

- Test Case ID: `SECT14`
- Input value: `Double.MIN_VALUE`
- Relevant Condition(s): `E2; BLB`
- Expected Output: `false`

containsValueOfDoubleMaxValueShouldBeFalse:

- Test Case ID: `SECT15`
- Input value: `Double.MAX_VALUE`
- Relevant Condition(s): `E2; AUB`
- Expected Output: `false`

containsValueOfDoubleNanShouldBeFalse:

- Test Case ID: `SECT16`
- Input value: `Double.Nan`
- Relevant Condition(s): `U1`
- Expected Output: `false`

equals

Based on the test strategy section above, we decided to include the following 6 tests:

Using example test Range(3,7) as the first range:

equalsSameRangeShouldBeTrue:

- Test Case ID: `SECT17`
- Input Used: `Range(3,7)`
- Relevant Condition(s): `E1`
- Expected Output: `true`

equalsDifferentUpperValuesShouldBeFalse:

- Test Case ID: `SECT18`
- Input Used: `Range(3,10)`
- Relevant Condition(s): `E2`
- Expected Output: `false`

equalsDifferentLowerValuesShouldBeFalse:

- Test Case ID: `SECT19`
- Input Used: `Range(4,7)`
- Relevant Condition(s): `E2`
- Expected Output: `false`

equalsDifferentLowerAndUpperValuesShouldBeFalse:

- Test Case ID: `SECT20`
- Input Used: `Range(1,10)`
- Relevant Condition(s): `E2`
- Expected Output: `false`

equalsNullRangeShouldBeFalse:

- Test Case ID: `SECT21`
- Input Used: `null Range()`
- Relevant Condition(s): `E2`
- Expected Output: `false`

equalsDifferentObjectShouldBeFalse:

- Test Case ID: `SECT22`
- Input value: `double`
- Relevant Condition(s): `E3`
- Expected Output: `false`

getCentralValue

Based on the test strategy section above, we decided to include the following 5 tests:

centralValueShouldBeFive:

- Test Case ID: `SECT23`
- Range Used: `Range(3,7)`
- Relevant Condition(s): `E1; Positive Range`
- Expected Output: `5.0`

centralValueShouldBeNegativeFive:

- Test Case ID: `SECT24`
- Range Used: `Range(-7,-3)`
- Relevant Condition(s): `E1; Negative Range`
- Expected Output: `-5.0`

centralValueShouldBeNegativeTwo:

- Test Case ID: `SECT25`
- Range Used: `Range(-7,3)`
- Relevant Condition(s): `E1; Mixed Range`
- Expected Output: `-2.0`

centralValueShouldBeThree:

- Test Case ID: `SECT26`
- Range Used: `Range(3,3)`
- Relevant Condition(s): `E1; Same Range`
- Expected Output: `3.0`

centralValueShouldBeZero:

- Test Case ID: `SECT27`
- Range Used: `Range(-Double.MAX_VALUE,Double.MAX_VALUE)`
- Relevant Condition(s): `E1; Extreme Range`
- Expected Output: `0.0`

getLength

Based on the test strategy section above, we decided to include the following 5 tests:

lengthValueShouldBeFour:

- Test Case ID: `SECT28`
- Range Used: `Range(3,7)`
- Relevant Condition(s): `E1; Positive Range`
- Expected Output: `4.0`

lengthValueShouldBeThree:

- Test Case ID: `SECT29`
- Range Used: `Range(-7,-4)`
- Relevant Condition(s): `E1; Negative Range`
- Expected Output: `3.0`

lengthValueShouldBeTen:

- Test Case ID: `SECT30`
- Range Used: `Range(-7,3)`
- Relevant Condition(s): `E1; Mixed Range`
- Expected Output: `10.0`

lengthValueShouldBeZero:

- Test Case ID: `SECT31`
- Range Used: `Range(3,3)`
- Relevant Condition(s): `E1; Same Range`
- Expected Output: `0.0`

lengthValueShouldBeDoubleMaxValue:

- Test Case ID: `SECT32`
- Range Used: `Range(0,Double.MAX_VALUE)`
- Relevant Condition(s): `E1; Extreme Range`
- Expected Output: `Double.MAX_VALUE`

DataUtilities Class

calculateColumnTotal

Based on the test strategy section above, we decided to include the following 6 tests:

Using Mock with 3Rowsx3Columns {1,2,3 4,5,6 7,8,9}:

calculateColumnTotalShouldBeTweleve:

- Test Case ID: `SECT33`
- Input value: data-`Mock[]`,column-`0`
- Relevant Condition(s): `E1; E2; LB`
- Expected Output: `12.0`

calculateColumnTotalShouldBeEighteen:

- Test Case ID: `SECT34`
- Input value: data-`Mock[]`,column-`2`
- Relevant Condition(s): `E1; E2; UB`
- Expected Output: `18.0`

calculateColumnTotalShouldBeFifteen:

- Test Case ID: `SECT35`
- Input value: data-`Mock[]`,column-`1`
- Relevant Condition(s): `E1; E2; NOM`
- Expected Output: `15.0`

calculateColumnTotalShouldBeZeroForColumnNegativeOne:

- Test Case ID: `SECT36`
- Input value: data-`Mock[]`,column-`-1`
- Relevant Condition(s): `U1`
- Expected Output: `0.0`

calculateColumnTotalShouldBeZeroForColumnThree:

- Test Case ID: `SECT37`
- Input value: data-`Mock[]`,column-`3`
- Relevant Condition(s): `U2`
- Expected Output: `0.0`

calculateColumnTotalShouldRaiseInvalidParameterException:

- Test Case ID: `SECT38`
- Input value: data-`null Mock[]`,column-`0`
- Relevant Condition(s): `U3`
- Expected Output: `InvalidParameterException`

calculateRowTotal

Based on the test strategy section above, we decided to include the following 6 tests:

Using Mock with 3Rowsx3Columns {1,2,3 4,5,6 7,8,9}:

calculateRowTotalShouldBeSix:

- Test Case ID: `SECT39`
- Input value: data-`Mock[]`,row-`0`

- Relevant Condition(s): `E1; E2; LB`
- Expected Output: `6.0`

calculateRowTotalShouldBeTwentyFour:

- Test Case ID: `SECT40`
- Input value: data-`Mock[]`,row-`2`
- Relevant Condition(s): `E1; E2; UB`
- Expected Output: `24.0`

calculateRowTotalShouldBeFifteen:

- Test Case ID: `SECT41`
- Input value: data-`Mock[]`,row-`1`
- Relevant Condition(s): `E1; E2; NOM`
- Expected Output: `15.0`

calculateRowTotalShouldBeZeroForRowNegativeOne:

- Test Case ID: `SECT42`
- Input value: data-`Mock[]`,row-`-1`
- Relevant Condition(s): `U1`
- Expected Output: `0.0`

calculateRowTotalShouldBeZeroForRowThree:

- Test Case ID: `SECT43`
- Input value: data-`Mock[]`,row-`3`
- Relevant Condition(s): `U2`
- Expected Output: `0.0`

calculateRowTotalShouldRaiseInvalidParameterException:

- Test Case ID: `SECT44`
- Input value: data-`null Mock[]`,row-`0`
- Relevant Condition(s): `U3`
- Expected Output: `InvalidParameterException`

createNumberArray

Based on the test strategy section above, we decided to include the following 4 tests:

Using example double[] = {1,2,3}

createNumberArrayShouldHaveLengthThree:

- Test Case ID: `SECT45`
- Input value: data-`double[]`
- Relevant Condition(s): `E1`
- Expected Output: `3.0`

createNumberArrayShouldHaveValueTwoInSecondIndex:

- Test Case ID: `SECT46`
- Input value: data-`double[]`
- Relevant Condition(s): `E1`
- Expected Output: `2.0`

createNumberArrayShouldHaveLengthZero:

- Test Case ID: `SECT47`
- Input value: data-`Empty double[]`
- Relevant Condition(s): `E1;`
- Expected Output: `0.0`

createNumberArrayShouldRaiseInvalidParameterException:

- Test Case ID: `SECT48`
- Input value: data-`null double[]`
- Relevant Condition(s): `U1`
- Expected Output: `InvalidParameterException`

createNumberArray2D

Based on the test strategy section above, we decided to include the following 6 tests:

Using example double[] = {1,2,3 4,5,6}

createNumberArray2DShouldHaveLengthTwoRows:

- Test Case ID: `SECT49`
- Input value: data-`double[][]`
- Relevant Condition(s): `E1`
- Expected Output: `2.0`

createNumberArray2DShouldHaveLengthThreeColumns:

- Test Case ID: `SECT50`
- Input value: data-`double[][]`
- Relevant Condition(s): `E1`
- Expected Output: `3.0`

createNumberArray2DShouldHaveValueFive:

- Test Case ID: `SECT51`
- Input value: data-`double[][]`
- Relevant Condition(s): `E1`
- Expected Output: `5.0`

createNumberArray2DShouldHaveLengthZeroRows:

- Test Case ID: `SECT52`
- Input value: data-`Empty double[][]`
- Relevant Condition(s): `E1`
- Expected Output: `0.0`

createNumberArray2DShouldHaveLengthZeroColumns:

- Test Case ID: `SECT53`
- Input value: data-`Empty double[][]`
- Relevant Condition(s): `E1`
- Expected Output: `0.0`

createNumberArray2DShouldRaiseInvalidParameterException:

- Test Case ID: `SECT54`
- Input value: data-`null double[][]`
- Relevant Condition(s): `U1`
- Expected Output: `InvalidParameterException`

getCumulativePercentages

Based on the test strategy section above, we decided to include the following 4 tests:

Using Mock KeyedValues(key,value) = {0, 1 1, 5 2, 9 }

getCumulativePercentagesShouldHaveThreeKeys:

- Test Case ID: `SECT55`
- Input value: data-`Mock KeyedValues`
- Relevant Condition(s): `E1`
- Expected Output: `3.0`

getCumulativePercentagesSameLengthShouldBeTrue:

- Test Case ID: `SECT56`
- Input value: data-`Mock KeyedValues`
- Relevant Condition(s): `E1`
- Expected Output: `true`

getCumulativePercentagesShouldHaveLastValueOne:

- Test Case ID: `SECT57`
- Input value: data-`Mock KeyedValues`
- Relevant Condition(s): `E1`
- Expected Output: `1.0`

getCumulativePercentagesShouldHaveZeroKeys:

- Test Case ID: `SECT58`
- Input value: data-`Empty Mock KeyedValues`
- Relevant Condition(s): `E1;`
- Expected Output: `0.0`

getCumulativePercentagesShouldRaiseInvalidParameterException:

- Test Case ID: `SECT59`
- Input value: data-`null Mock KeyedValues`

- Relevant Condition(s): `U1`
- Expected Output: `InvalidParameterException`

4 How the team work/effort was divided and managed

The test plan and strategy was developed together from all team members. Afterwards, we split into two groups to carry out all the tests. The first group focused on the Range class while the second group focused on the DataUtilities class. Once both groups have completed their testing, we scheduled a peer review meeting to go over the issues found from each group. The meeting was used to discuss consistent severity and priority ratings, reproducibility procedure formating and removed any duplicate issues that were found by both teams.

Group 1: Bogdan and Andy

Group 2: Juan, Billy and Moaz

5 Difficulties encountered, challenges overcome, and lessons learned

The mocking portion of the assignment was difficult and a challenge we had to overcome. Mock frameworks are often used in scenarios where we want to capture parameters and verify them against pre-determined expectations. For all the methods we have used mocks, our mocks showed us if the method behave as intended or not by verifying that a method was or wasn't called. However, this involved guessing what the mocked component will do even with the Javadoc specification available. In our case, some of the system matched the specifications and others didn't. The drawbacks associated with mocking include inconsistencies between the mock data and the real data, which can lead to false positives/negative in the test results. Ultimately, we overcome the challenges with mocking and learned its advantages and disadvantages as well as adaptability when conducting thorough software testing unit testing.

6 Comments/feedback on the lab itself

We all thought the lab was very helpful as it exposed us to test code using JUnit and mocking. Additionally, we also thought the instructions of the lab was structured well.