SEENG 438 - Lab Report #2

Group 11:
Ryan Khryss Obiar (30151002)
Armin Sandhu (30143482)
Parsa Kargari (30143368)
Jayden Mikulcik (30139637)

REPO LINK : https://github.com/seng438-winter-2024/seng438-a2-haywirebanana :

1 Introduction

In this lab, our group got introduced to automating tests using JUnit on the Eclipse IDE. We were tasked and given a software program ("JFreeChart") to test and write JUnit tests for, while ensuring we follow Javadoc requirements. We specifically tested two classes, being the org.jfree.data.Range class and the org.jfree.data.DataUtilities class, making sure that we are to test 5 methods from each class. The objective of this lab was to learn and get familiar with creating test cases, implementing mock objects, and executing them.

Our aim was to identify bugs throughout different versions of the JAR program. This involved us learning how to use the Eclipse IDE effectively, making sure our JUnit tests follow the Javadoc set requirements, and successfully executing our mock objects. This lab is great in simulating a real world industry example of a problem a company could have in regards to software another dev built that needs to be tested. By having to resort to the documentation to come up with the requirements needed to be satisfied in the test suite we incorporate the skills we learned from the first lab in needing to identify potential problems a code base could have.

2 Detailed description of unit test strategy

2.1 Scope of Testing

The Scope of our testing consisted of a small subsection of the greater ("JFreeChart") library the scope includes testing five methods of the classes Range and DataUtilities. The testing scope of each method in the mentioned classes consisted of each edge case we could think we needed to implement when reading each method's individual documentation.

2.1.1 Classes to be tested

2 Classes of JFreeChart will be tested:

Class Name	**Description**
Range	Represents an immutable range of values by creating a new Range
DataUtilities	Utility methods for use with some of the data classes

2.1.2 Methods to be tested

We chose 5 methods from both the Range class and DataUtilities class for testing, these methods and their descriptions are the following :

Belongs to class	Method Name	Description
Range org.jfree.data.Range	double getLowerBound()	Returns the lower bound for the range.
	double getUpperBound()	Returns the upper bound for the range.
	double getLength()	Returns the length of the range.
	boolean intersects(double lower, double upper)	Returns true if the range intersects (overlaps) with the specified range, and false otherwise.
	boolean contains(double value)	Returns true if the specified value is within the range and false otherwise.
DataUtilities org.jfree.data.DataUtilities	double calculateColumnTotal(Values2D, int)	Returns the sum of the values in one column of the supplied data table. With invalid input, a total of zero will be returned.
	double calculateRowTotal(Values2D,int)	Returns the sum of the values in one row of the supplied data table. With invalid input, a total of zero will be returned.
	Number[] createNumberArray(double[])	Constructs an array of Number objects from an array of double primitives.
	Number[][] createNumberArray2D(double[][])	Constructs an array of arrays of Number objects from a corresponding structure containing double primitives.
	KeyedValues getCumulativePercentages(KeyedValues): KeyValues	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.

2.2 Test Type

In this Lab, we are tasked with getting comfortable and exploring **unit testing**. Testing each small component and function is good as it improves the overall code quality and allows the tester to cover as many different parts of the software are possible. This practice of breaking down the code into its smallest testable parts not only enhances the reliability of the software but also facilitates easier debugging and maintenance in the long run. By systematically examining individual units of code, developers can identify and address potential bugs or errors early in the development process, reducing the likelihood of encountering more complex issues later on. This lab was also set up in such a way we had to employ black box testing. This testing is when the testers don't know the internal code, but just test the end to end functions. This way of testing is beneficial as the testers can solely focus on the inputs and outputs of the system, essentially evaluating the functionality of the system through a user's perspective. These testing methods will better equip us with real-word testing scenarios in the software industry.

2.3 Test Objective

The objective of writing these test cases is to identify and catch all potential bugs within the aforementioned methods of the classes Range and DataUtilities. Having this objective is essential in creating a quality test suite that can be used to deploy to production with full confidence in the performance of the product.

2.4 Test-Case Design Approach

Our test case design approach consisted of discussing as a group of four multiple different techniques of test case design we could implement. Overall in our discussion we were able to come up with tests using different established techniques including Decision Testing Coverage, an approach to test case design that will be used as an example. One example of implementation for each test case in our test suite is having unit tests intersectLower(), intersectUpper(), intersectAll(), and intersectNone() for the method intersections. By having all four tests for this method we ensured that all Decisions with regard to intersects had been tested leading to an error being caught in intersectsUpper() which likely would have been missed if not for Decision Testing Coverage. Overall our approach to developing our test cases consisted of discussions among our group on where to implement different test case design approaches within our test suite.

Furthermore, we made sure to follow the important techniques set by black-box testing such as Equivalence Partitioning and Boundary Value Analysis:

- **Equivalence Partitioning**: For each of the methods we tested, we attempted to input both valid and invalid inputs to ensure the correct handling of various inputs. For example in the “getLowerBound” method, we tested using negative inputs, positive inputs, and NaNs.

- **Boundary Value Analysis**: We made sure to test methods at the edges of the function. In other words, testing functions at their bounds, just below it and slightly above it. For example, in the "intersects" method, we made sure to test at its bound, slightly within its bounds and also slightly above.

In specific, here are some of the test cases classified into Equivalence and Boundary, as well as additional Worst Case and Robust:

1. **Range Class**
    - **getLowerBound()**
        - Equivalence
            - lowerBoundShouldBeOne()
        - Boundary
            - lowerBoundShouldBeNegativeOne()
        - Worst Case
            - lowerBoundShouldBeMinimumValue()
            - lowerBoundWorstCaseTesting()
        - Robust
            - lowerBoundShouldBeNaN()

    - **getUpperBound()**
        - Equivalence
            - upperBoundTestOfDistinctRange()
        - Boundary
            - upperBoundShouldBeOne()
        - Worst Case
            - upperBoundIsLargestVal()

    - **getLength()**
        - Equivalence
            - lengthShouldBeTwo()
        - Boundary
            - lengthShouldBeZero()
        - Worst Case
            - lengthShouldBePositiveForNonEmptyRange()
        - Robust
            - lengthWithNaNLower()

    - **intersects(double lower, double upper)**
        - Equivalence
            - intersectAll()
            - intersectNone()

        - intersectSame()
    - Boundary
        - intersectLower()
        - intersectUpper()
    - Robust
        - intersectNaN()

- **contains(double value)**
    - Equivalence
        - containsValue()
    - Boundary
        - doesNotContainValue()
    - Robust
        - doesNotContainNaNValue()

2. DataUtilities Class

- **calculateColumnTotal(Values2D data, int column)**
    - Equivalence
        - testColumnOtherThanOne()
    - Boundary
        - testNegativeColumnIndex()
        - testTenRowsForColumnTotal()
    - Robust
        - testNegativeValues()
        - testNullValue()

- **calculateRowTotal(Values2D data, int row)**
    - Equivalence
        - testIntAsValuesInData()
        - testDoubleAsValuesInDate()
    - Robust
        - testInvalidDataObject()
        - testNegativeRowIndex()

- **createNumberArray(double[] data)**
    - Equivalence
        - testCreateNumberArrayWithValidInput()
    - Boundary
        - testCreateNumberArrayWithEmptyInput()
    - Robust
        - testNullDataValue()

- **createNumberArray2D(double[][] data)**
    - Equivalence
        - testCreateNumberArray2DWithValidInput()
    - Boundary
        - testCreateNumberArray2DWithEmptyInput()
        - testCreateNumberArray2DWithOneValue()
    - Robust
        - testNullDataValueCreateNumberArray2D()

- **getCumulativePercentages(KeyedValues data)**
    - Equivalence
        - calculateCumulativePercentageForThreeValues()
        - calculateCumulativePercentageWithFloatingPointValues()
    - Boundary
        - calculateCumulativePercentageForEmpty()
        - calculateCumulativePercentageWithNegativeValues()
        - calculateCumulativePercentageForOneValue()

Lastly, we wanted to mention our use of Mocking in our test cases, primarily in the DataUtilities class. Specifically, JMock is a library we used that mocks objects to help design and execute tests for components that need objects. The library helped us define our mock objects so we could test our given methods. We found this method of testing very intuitive and easy to implement, allowing us to isolate the components we needed to. The set up was not too complex, and it served us well for the functionality we needed to test. On the other hand, some drawbacks we see that can arise with using mocking is that it may be hard to emulate the real objects that would be used with our method being tested. We are building the objects from scratch, thus can see how easily things could be overlooked for complex objects. Additionally, because we have to manually specify the characteristics of the objects, as the code gets updated when using mocking you would need to ensure that you are always updating the mock objects, which can get complicated.

3 Test cases developed

Range Class					
Method	Test Name	Test Description + Why	Test Argument Input	Expected Outcome	Actual Outcome

double getLowerBound()	lowerBoundShouldBeNegativeOne()	Test that checks that a range of 2 distinct values returns the correct lower bound	exampleRange = new Range(-1, 1);
			
			exampleRange.getLowerBound()
				-1	-1

	lowerBoundShouldBeOne()	Test that checks that a range of 2 of the same values returns the correct lower bound	exampleRange1 = new Range(1, 1);
			
			exampleRange1.getLowerBound()
				1	1

	lowerBoundShouldBeMinimumValue()	Test that checks that a range of 2 distinct values with the lower bound being the integer min value returns the min value	range = new Range(Integer.MIN_VALUE, 100);
			
			range.getLowerBound()
				Integer.MIN_VALUE	Integer.MIN_VALUE

	lowerBoundShouldBeNaN()	Test that checks that a range of 2 distinct values with the lower bound being NaN returns NaN	range = new Range(Double.NaN, 100);
			
			range.getLowerBound()
				Double.NaN	Double.NaN

	lowerBoundWorstCaseTesting()	Test that checks that a range of the max representable value returns the max representable value	range = new Range(Double.MAX_VALUE, Double.MAX_VALUE);
			
			range.getLowerBound()
				Double.MAX_VALUE	Double.MAX_VALUE

double g my etUpperBound()	upperBoundTestOfDistinctRange()	Test that checks that a range of 2 distinct values returns the correct upper bound	exampleRange = new Range(-1, 1);
			
			exampleRange.getUpperBound()
				1	-1

	upperBoundShouldBeOne()	Test that checks that a range of 2 of the same values returns the correct upper bound	exampleRange1 = new Range(1, 1);
			
			exampleRange1.getUpperBound()
				1	1

	upperBoundIsLargestVal()	Test that checks that a range with an upper bound of the max value returns the max value	range = new Range(0, Double.MAX_VALUE);
			
			range.getUpperBound()
				Double.MAX_VALUE	0

double getLength()	lengthShouldBeTwo()	Test that checks that a range of 2 distinct values returns the correct length of the interval	exampleRange = new Range(-1, 1);
			
			exampleRange.getLength()
				2	2

	lengthShouldBeZero()	Test that checks that a range of 2 of the same values returns the correct length of the interval	exampleRange1 = new Range(1, 1);
			
			exampleRange.getLength()
				0	0

	lengthShouldBePositiveForNonEmptyRange()	Test that checks that a range of 2 distinct values returns a positive length	Range range = new Range(-1, 1); range.getLength() > 0	Length > 0	Length > 0

	lengthWithNaNLower()	Test that checks that a range with a NaN lower bound returns NaN	range = new Range(Double.NaN, 10);
			
			range.getLength()
				NaN	NaN

boolean intersects(double lower, double upper)	intersectLower()	Test that returns true when a given range intersects the lower bound of the class member range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(-2,1)
				true	true

	intersectUpper()	Test that returns true when a given range intersects the upper bound of the class member range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(0, 2);
				true	false

	intersectAll()	Test that returns true when a given range encapsulates the both bounds of the class member range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(-2,2)
				true	true

	intersectNone()	Test that returns false when a given range is distinct from both bounds of the class member range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(-4,-2)
				false	false

	intersectSame()	Test that returns true is a range intersects the same range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(-1,1)
				true	true

	intersectNaN()	Test that returns False when checking if NaN bounds intersect a range	exampleRange = new Range(-1, 1);
			
			exampleRange.intersects(Double.NaN, Double.NaN));
				false	false

boolean contains(double value)	containsValue()	Test that returns true when an inputted value is within the range	exampleRange = new Range(-1, 1);
			exampleRange.contains(0)
				true	true

	doesNotContainNaNValue()	Test that returns false when checking that a range contains NaN	exampleRange = new Range(-1, 1);
			
			exampleRange.contains(Double.NaN));
				false	false

	doesNotContainValue()	Test that returns false when an inputted value is not within the range	exampleRange = new Range(-1, 1);
			
			exampleRange.contains(5)
				false	false

DataUtilities Class

Method	Test Name	Test Description + Why	Test Argument Input	Expected Outcome	Actual Outcome
double **calculateColumnTotal**(Values2D data, int column)	testColumnOtherThanOne()	Test to make sure positive parameters inputs are accepted	Data: 1, 2, 3, 4 5, 5, 5, 5 column: 2	Returns 8	Returns 8
	testNegativeValues()	Test to make sure null values in data table are not permitted	data: -10 5 Column: 0	Returns 5	Returns 5
	testNullValue()	Test to check that method returns 0 due to invalid value in data	data: null column: 0	Returns 0	Returns 1
	testNegativeColumnIndex()	Test to make sure negative integers are not permitted for column number	Data: 10 Column: -1	InvalidParameterException	Expected Error Occurs
	testTenRowsForColumnTotal()	Test that functions calculates a high number of elements correctly	Data: 12 13 55 123 23 21313 123 11	Returns 21676	Returns 21676

			1 2.1 Column: 0		
double **calculateRowTotal**(Values2D data ,int row)	testInvalidDataObject()	Test to make sure invalid data object is not permitted	Data: “Hi” 5 Row: 0	InvalidParameterException	Expected Error Occurs
	testIntAsValuesInData()	Test to check Ints as a value in data work as intended	Data: 5 Row: 0	Returns 5	Returns 0
	testDoubleAsValuesInData()	Test to check Doubles as value in data work as intended	Data: 0.5 0.2 Row: 0	Returns 0.7	Returns 0.5
	testNegativeRowIndex()	Tests that negative row index is not permitted	Data: 1 1 2 2 Row: -1	InvalidParameterException	ExpectationError
Number[] **createNumberArray**(double[] data)	testNullDataValue()	Test to make sure null values in data table are not permitted	null	InvalidParameterException	IllegalParameterException
	testCreateNumberArrayWithValidInput()	Test to make sure the method is able to create a number array with valid double inputs	{2.0,3.4,5.6}	{2.0,3.4,5.6}	{2.0,3.4,null}
	testCreateNumberArrayWithOneValue()	Test to make sure the method is able to create a number array with one decimal input	{2.0}	{2.0}	{null}
	testCreateNumberArrayWithEmptyInput()	Test to make sure the method is able to create an empty array.	{}	{}	{}

Number[] **createNumberArray2D**(double[][] data) [])	testNullDataValueCreateNumberArray2D()	Test to make sure null values in data table are not permitted	null	InvalidParameterException	IllegalParameterException
	testCreateNumberArray2DWithValidInput()	Test to make sure the successful creation of a 2D array	{{2.0, 3.4, 5.6}, {1.0, 2.0, 3.0}}	{{2.0, 3.4, 5.6}, {1.0, 2.0, 3.0}}	{{2.0, 3.4, null}, {1.0, 2.0, null}}
	testCreateNumberArray2DWithOneValue()	Test to make sure the successful creating of a 2D Array that's 1x1	{{2.0}, {1.0}}	{{2.0}, {1.0}}	{{null}, {null}}
	testCreateNumberArray2DWithEmptyInput()	Test to make sure the method is able to create an empty 2D array.	{},{}	{},{}	{},{}
KeyedValues **getCumulativePercentages**(KeyedValues data)	calculateCumulativePercentageForThreeValues()	Test to make sure method returns correct cumulative percentage for a keyedvalue structure of 3 values	Key Value 0 5 1 9 2 2	Key Value 0 0.3125 1 0.875 2 1.0	Key Value 0 0.45454 1 2
	calculateCumulativePercentageForOneValue()	Test to make sure method returns correct cumulative percentage for a keyedvalue structure of 1 value	Key Value 0 5	Key Value 0 1.0	Key Value 0 inf
	calculateCumulativePercentageForEmpty()	Test to make sure method returns empty keyedvalue structure	Key Value	Key Value	Key Value
	calculateCumulativePercentageWithNegativeValues()	Test to make sure method returns correct cumulative percentage for a keyedvalue structure of 2 negative values and 1 positive value	Key Value 0 -5 1 9 2 -2	Key Value 0 0.3125 1 0.875 2 1.0	Key Value 0 -0.71428 1 2
	calculateCumulativePercentageWithFloatingPointValues()	Test to make sure method returns correct cumulative percentage for a keyedvalue structure of 3 floating point values	Key Value 0 0.5 1 0.25 2 0.25	Key Value 0 0.5 1 0.75 2 1.0	Key Value 0 1.0 1 2

4 How the teamwork/effort was divided and managed

When devising our testing plan we came together as a group and discussed the different strategies we wanted to use when tackling this assignment. We discussed and came up with the plan to continuously swap our peer review partners to ensure that each test was crafted to perfection and meet all the standards that each test needed to uphold .This approach was aimed at maximizing the quality of our test cases by leveraging multiple perspectives. By rotating peer review partners, we ensured that each test was crafted to perfection and met all the necessary standards. It also promoted a collaborative environment where everyone's input and expertise could contribute to the refinement of the testing process. This was decided before we started and each pair worked in conjunction with the other pair to write the test cases at the same time for each method then compare and discuss with the other group. Not only did this allow for every member to be engaged and broaden our knowledge about unit testing and black box testing, this was also beneficial for our code quality. Ultimately, this method allowed for a coherent development experience between the four of us, effective learning, and quality work to be produced.

5 Difficulties encountered, challenges overcome, and lessons learned

Some difficulties we encountered were as a whole using the JUnit framework as most of us had limited experience developing JUnit test suites and executing these test cases. Overall the way to overcome this challenge was to meticulously read and understand the online documentation as this allowed our test cases to have all the functionality that we had planned to implement. Similarly with JMock, there was a good amount of documentation the group had to familiarize ourselves with, but in the end, this did help us overcome this difficulty. The greatest lesson taken away as a group we came away with was learning to come up with an understanding of the technology at hand and using resources to make progress on the assignment. This assignment really emphasized the importance of getting familiarized with documentation, and how useful this can be when developing code. In the end our difficulties directly lead to a better understanding of JUnit/JMock allowing us to overcome the errors in our test cases and expand on our knowledge of JUnit/JMock we learn in class.

6 Comments/feedback on the lab itself

Overall feedback on this lab was very positive as we got to use and understand a technology used in the field. This coupled with the critical thinking it took to come up with each method we used meant to our group a lab that is essential to complete in order to round out our skills as software engineers. The general consensus of our group was that the lab was very well designed from the content we had to implement to the workflow of the readme being laid out very well allowing for a great lab experience.