ENSF 438 Lab 2 - Introduction to Unit Testing

Group Number: 6 - Neha Parmar, Shahed Issa, Jaisumer Sandhu, Jaival Patel

February 26, 2025

Introduction

In the previous lab, we familiarized ourselves with testing and tracking software bugs through exploratory and manual functional testing. Using Jira, we were able to document and track bugs along with detailed descriptions. In this lab, we shift our focus to unit testing, implementing code, and creating test cases for specific methods within the software. With JUnit, we will test JFreeChart, a Java library used for generating various types of charts, while also utilizing mock objects to simulate dependencies and analyze test results. Although we have had prior exposure to unit testing and JUnit in previous classes, our experience has been mostly theoretical with limited hands-on practice. This lab provides an opportunity to deepen our understanding of JUnit and gain practical experience in writing and executing test code using software such as Eclipse.

Detailed Description of unit test strategy

The unit test strategy we chose follows a structured approach to ensure comprehensive testing of the JFreeChart library. The strategy focuses on validating individual methods and their expected behavior under different conditions. It involves defining test cases using JUnit and leveraging mock objects to isolate dependencies.

Input Partitions

1. **Valid Inputs—**This partition includes expected values that conform to the function's requirements. These test cases verify the method's correctness under normal operating conditions.

2. **Boundary Values—**We test edge cases, such as the minimum and maximum allowable values, to ensure the software correctly handles extreme conditions.

3. **Invalid Inputs** – Inputs that deviate from expected formats, such as null values, empty datasets, or negative numbers where positive values are required, are included to assess error handling and robustness.

Test Case Developed

The Design Component:

Test Plan:

Range methods being tested:

1. **equals()**
    a. Test equality for identical ranges.
    b. Test equality with null (should return false).
    c. Test equality with a different object type.
2. **expand()**
    a. Expand a range with positive margins.
    b. Expand a range with negative margins (shrinking).
    c. Expand with zero margins (should remain unchanged).
3. **expandToInclude()**

a. Expand when the value is inside the range (should remain unchanged).
b. Expand when the value is below the lower bound.
c. Expand when the value is above the upper bound.

4. **getLength()**
    a. Test a range with positive bounds.
    b. Test a range with negative bounds.
    c. Test a single-point range (lower == upper).
5. **getLowerBound()**
    a. Test for positive range between (10, 20) should return 2
    b. Test for negative range between (-15, -5) should return -15
    c. Test for a zero-bounded range between (0, 10) should return 0

DataUtilities methods being tested:

1. **calculateColumnTotal()**
    a. Compute total for a valid column with positive values.
    b. Compute total for a column containing null values.
    c. Compute total for an empty dataset.
    d. Handle an invalid column index (should return 0).
2. **calculateRowTotal()**
    a. Compute total for a valid row with positive values.
    b. Compute total for a row containing null values.
    c. Compute total for an empty dataset.
    d. Handle an invalid row index (should return 0).
3. **createNumberArray()**
    a. Test with normal array of positive values
    b. Test with a null array
    c. Test with a empty array
4. **createNumberArray2D()**
    a. Test with normal array of positive values
    b. Test with a null array
    c. Test with a empty array
5. **getCumulativePercentages()**
    a. Test with a normal array of positive values
    b. Test with a null array
    c. Test with a empty array

Test-case design:

Range:

For the Range class, we're focusing on equivalence partitioning and boundary value analysis to make sure we test all possible inputs effectively. We're dividing inputs into two categories: Expected (E), which includes normal numerical values, positive margins for expansion, and values inside or just outside the range, and Unexpected (U), which covers NaN, null, negative margins, and cases where the upper bound is smaller than the lower bound. By testing both valid and invalid inputs, we can confirm that the class handles errors gracefully while functioning correctly in normal scenarios.

For catching edge cases, we're using boundary value analysis by testing things like zero-length ranges, extreme expansions, and adjustments when a new value is just outside the range. For example, in getLength(), we're checking how it handles positive, negative, and identical bounds. In expandToInclude(), we'll see if the lower or upper bound shifts correctly when a new value is introduced. This structured approach helps us catch any inconsistencies and verify that Range behaves appropriately.

Input Range definitions:

- Lower Bound (lower) – The starting value of the range
- Upper Bound (upper) – The ending value of the range
- Margins (lowerMargin, upperMargin) – Used in expand(), defining percentage-based adjustments
- Value (value) – Used in expandToInclude(), representing a number that may extend the range

E:

- lower: Any finite number
- upper: Any finite number greater than lower
- lowerMargin, upperMargin: ≥ 0
- value: A number inside the range, below the lower bound, or above the upper bound

U:

- lower: NaN, null
- upper: NaN, null, upper < lower
- lowerMargin, upperMargin: < 0
- value: NaN, null

Boundary Value Analysis Approach:

getLength(), we test:

- Range(10, 20), expected length = 10
- Range(-10, 5), expected length = 15
- Range(5, 5), expected length = 0

expandToInclude(), we test:

- A value inside the range (no change)
- A value below the lower bound (shifts lower bound)
- A value above the upper bound (shifts upper bound)

DataUtilities:

For DataUtilities, we're applying the same structured testing approach to make sure the methods work reliably across different types of datasets. The key inputs include 2D data tables (Values2D), numerical arrays (double[], double[][]), and column/row indices. Expected inputs (E) include properly formatted datasets with valid numerical values, while unexpected inputs (U) cover things like null datasets, empty arrays, and out-of-bounds indices.

We're also using boundary value analysis to check how the methods react to edge cases like empty datasets returning 0, handling null values within row and column totals, and creating arrays with various data sizes. For getCumulativePercentages(), we'll confirm that it properly calculates percentages that sum to 1 and correctly deals with lists containing null or negative values.

Input Range Definitions:

- 2D Data (Values2D) – Represents tabular data
- Column/Row Index (column, row) – Specifies which column or row is being computed
- 1D/2D Number Arrays (double[], double[][]) – Represents numerical datasets
- Keyed Values (KeyedValues) – Used in cumulative percentage calculations

E:

- Values2D: A non-empty dataset
- column/row: 0 ≤ index < N (within dataset bounds)
- double[] / double[][]: A non-empty array
- KeyedValues: A non-empty list with valid percentages

U:

- Values2D: null, empty dataset
- column/row: Negative index, out-of-bounds index
- double[] / double[][]: null, empty array
- KeyedValues: null, empty list, negative values

Boundary Value Analysis Approach:

calculateColumnTotal() and calculateRowTotal(), we test:

- A valid row/column with all positive numbers
- A valid row/column with a mix of positive, negative, and zero values
- A row/column with null values (ensuring they are ignored)
- An out-of-bounds column/row (expecting a return value of 0)
- An empty dataset (expecting a return value of 0)

createNumberArray() and createNumberArray2D(), we test:

- A valid array with positive numbers
- A valid array with negative numbers
- A valid array with mixed numbers
- A null array (expecting an InvalidParameterException)
- An empty array (expecting an empty Number[])

getCumulativePercentages(), we test:

- A valid KeyedValues list (ensuring percentages sum to 1)
- A list containing null values (verifying correct handling)
- A list with all zero values (ensuring correct behavior)
- A negative value in the dataset (ensuring error handling)

All appropriate values using assert method are listed below in our actual test cases.

How the Teamwork/Effort Was Divided and managed

Our team has four members, and as stated in the instructions, this lab was performed as a team. To start, we had a session to familiarize ourselves with Eclipse, as none of us had prior experience using this software. Furthermore, during this meeting, we performed two unit tests together as a team—one from the *Range* class and one from the *DataUtilities* class. Through this session, we were able to understand the software we are using, which allowed us to begin implementation on our own as well. After gaining some experience with Eclipse, each team member completed two unit tests, ensuring an equal distribution of work. To maintain accuracy and consistency, we reviewed each other's tests to verify correctness. Through this approach, we built confidence in writing the tests independently while also ensuring that everyone on the team had a clear understanding of the testing process.

Difficulties Encountered, Challenges Overcome, and Lessons Learned

Difficulties Encountered:

- The first and biggest difficulty that we faced as a team was setting up Eclipse as the software would work on some team members' laptops but not others
- Understanding the methods within each of the classes. For example, When we did choose the 10 methods that we would test, trying to understand each method was difficult because it was not formatted in the conventional Java style and we had only ever done whitebox testing in the past.
- Writing and implementing the tests for both the Range and DataUtilities. For example, getCentralValue() was challenging because of the need for precise behavior understanding, and also for mocking interfaces like Values2D was difficult to set the mocks and verify interactions correctly

Challenges Overcome:

- To overcome the issue with the Eclipse software we decided to work on the laptops of the team members who were able to use Eclipse, and the people who could not set up Eclipse had to work in pairs to get their tests done as well.
- To understand the methods within each class we worked together as a team for the first two unit tests, through which we were able to help each other find the documentation for the methods, we did one test from the Range class and another from the DataUtilities class
- The challenge of writing and implementing the test was overcome by breaking down complex methods into smaller units and collaborating to ensure we got the best result possible

Lessons Learned

- We were able to gain experience working with the Eclipse software and being able to implement or test any methods necessary
- Being able to understand the methods without actually seeing the code being tested along with how they are tested as well through teamwork

Comments/Feedback on the Lab and Lab Document Itself

Overall, this lab provided a valuable hands-on introduction to requirements based testing. It was useful to have both classes to test and understand how to implement the JUnit tests and document what was tested and how. The instructions were comprehensible, however it would be helpful if they were more concise and easy to follow. It would be helpful to include additional guidance on how to set up Eclipse based on operating systems as well.

Range Test cases:

Method	Test(s)

equals(Object obj)	@Test
	public void testEqualsSameRange() {
	Range range1 = new Range(0, 10);
	Range range2 = new Range(0, 10);
	assertTrue("Identical ranges should be equal", range1.equals(range2));
	}
	
	@Test
	public void testEqualsDifferentRange() {
	Range range1 = new Range(0, 10);

	Range range2 = new Range(5, 15);
	assertFalse("Different ranges should not be equal",
	range1.equals(range2));
	}
	
	@Test
	public void testEqualsNullObject() {
	Range range = new Range(0, 10);
	assertFalse("Range should not be equal to null", range.equals(null));
	}
	
	@Test
	public void testEqualsDifferentObjectType() {
	Range range = new Range(0, 10);
	String differentObject = "Not a Range";
	assertFalse("Range should not be equal to a different object type",
	range.equals(differentObject));
	}

expand(Range range,
double lowerMargin,
double upperMargin)
	@Test
	public void testExpandPositiveRange() {
	Range range = new Range(10, 20);
	Range expanded = Range.expand(range, 0.1, 0.2); // Expands 10%
	lower, 20% upper
	assertEquals("Lower bound should be 9", 9,
	expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should be 22", 22,
	expanded.getUpperBound(), 0.0001);
	}
	
	@Test
	public void testExpandNegativeRange() {
	Range range = new Range(-20, -10);
	Range expanded = Range.expand(range, 0.5, 0.5); // Expands 50%
	lower, 50% upper
	assertEquals("Lower bound should be -25", -25,
	expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should be -5", -5,
	expanded.getUpperBound(), 0.0001);
	}
	
	@Test
	public void testExpandWithZeroMargins() {
	Range range = new Range(10, 20);
	Range expanded = Range.expand(range, 0, 0); // No change
	assertEquals("Lower bound should remain 10", 10,
	expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should remain 20", 20,
	expanded.getUpperBound(), 0.0001);
	}

expandToInclude(Ra nge range, double	@Test public void testExpandToIncludeInsideRange() {

value)	Range range = new Range(5, 15);
	Range expanded = Range.expandToInclude(range, 10); // Inside range
	assertEquals("Lower bound should remain 5", 5, expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should remain 15", 15, expanded.getUpperBound(), 0.0001);
	}
	
	@Test
	public void testExpandToIncludeLower() {
	Range range = new Range(5, 15);
	Range expanded = Range.expandToInclude(range, 2); // Below lower bound
	assertEquals("Lower bound should become 2", 2, expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should remain 15", 15, expanded.getUpperBound(), 0.0001);
	}
	
	@Test
	public void testExpandToIncludeUpper() {
	Range range = new Range(5, 15);
	Range expanded = Range.expandToInclude(range, 20); // Above upper bound
	assertEquals("Lower bound should remain 5", 5, expanded.getLowerBound(), 0.0001);
	assertEquals("Upper bound should become 20", 20, expanded.getUpperBound(), 0.0001);
	}

getLength()	@Test
	public void testGetLengthPositiveRange() {
	Range range = new Range(10, 20);
	assertEquals("Length should be 10", 10, range.getLength(), 0.0001);
	}
	
	@Test
	public void testGetLengthNegativeRange() {
	Range range = new Range(-10, 5);
	assertEquals("Length should be 15", 15, range.getLength(), 0.0001);
	}
	
	@Test
	public void testGetLengthZeroRange() {
	Range range = new Range(5, 5);
	assertEquals("Length should be 0", 0, range.getLength(), 0.0001);
	}

getLowerBound()	@Test
	public void testGetLowerBoundPositiveRange() {
	Range range = new Range(10, 20);
	assertEquals("Lower bound should be 10", 10, range.getLowerBound(), 0.0001);
	}

	@Test
	public void testGetLowerBoundNegativeRange() {
	Range range = new Range(-15, -5);
	assertEquals("Lower bound should be -15", -15, range.getLowerBound(), 0.0001);
	}
	
	@Test
	public void testGetLowerBoundZero() {
	Range range = new Range(0, 10);
	assertEquals("Lower bound should be 0", 0, range.getLowerBound(), 0.0001);
	}

DataUtilities Test cases:

Method	Test(s)

calculateColumnTotal	@Test
	public void testCalculateColumnTotal_ValidData() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getRowCount(); will(returnValue(3));
	allowing(mockData).getValue(0, 1); will(returnValue(5.0));
	allowing(mockData).getValue(1, 1); will(returnValue(3.0));
	allowing(mockData).getValue(2, 1); will(returnValue(2.0));
	}});
	
	double result = DataUtilities.calculateColumnTotal(mockData, 1);
	assertEquals("Sum of column 1 should be 10.0", 10.0, result, 0.0001);
	}
	
	@Test
	public void testCalculateColumnTotal_EmptyData() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getRowCount(); will(returnValue(0));
	}});
	
	double result = DataUtilities.calculateColumnTotal(mockData, 1);
	assertEquals("Sum should be 0 for an empty table", 0.0, result, 0.0001);
	}
	
	@Test
	public void testCalculateColumnTotal_NullValues() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getRowCount(); will(returnValue(3));
	allowing(mockData).getValue(0, 1); will(returnValue(null));
	allowing(mockData).getValue(1, 1); will(returnValue(1.0));
	allowing(mockData).getValue(2, 1); will(returnValue(5.0));
	}});
	
	double result = DataUtilities.calculateColumnTotal(mockData, 1);
	assertEquals("Sum should only include non-null values", 6.0, result, 0.0001);
	}
	@Test(expected = InvalidParameterException.class)
	public void testCalculateColumnTotal_NullData() {
	DataUtilities.calculateColumnTotal(null, 1);

	}

calculateRowTotal	@Test
	public void testCalculateRowTotal_ValidData() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getColumnCount(); will(returnValue(3)); // 3 columns
	allowing(mockData).getValue(1, 0); will(returnValue(4.0)); // Value at row 1, column 0
	allowing(mockData).getValue(1, 1); will(returnValue(2.0)); // Value at row 1, column 1
	allowing(mockData).getValue(1, 2); will(returnValue(3.0)); // Value at row 1, column 2
	}});
	
	double result = DataUtilities.calculateRowTotal(mockData, 1);
	assertEquals("Sum of row 1 should be 9.0", 9.0, result, 0.0001); // 4.0 + 2.0 + 3.0 = 9.0
	}
	
	@Test
	public void testCalculateRowTotal_EmptyData() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getColumnCount(); will(returnValue(0));
	}});
	
	double result = DataUtilities.calculateRowTotal(mockData, 1);
	assertEquals("Sum should be 0 for an empty table", 0.0, result, 0.0001);
	}
	
	@Test
	public void testCalculateRowTotal_NullValues() {
	Mockery context = new Mockery();
	Values2D mockData = context.mock(Values2D.class);
	
	context.checking(new Expectations() {{
	allowing(mockData).getColumnCount(); will(returnValue(3));
	allowing(mockData).getValue(1, 0); will(returnValue(null));
	allowing(mockData).getValue(1, 1); will(returnValue(2.0));
	allowing(mockData).getValue(1, 2); will(returnValue(null));
	}});
	
	double result = DataUtilities.calculateRowTotal(mockData, 1);
	assertEquals("Sum should only include non-null values", 2.0, result, 0.0001);
	}

	@Test(expected = InvalidParameterException.class)
	public void testCalculateRowTotal_NullData() {
	DataUtilities.calculateRowTotal(null, 1);
	}

createNumberArray	@Test
	public void testCreateNumberArray_ValidData() {
	double[] input = {1.0, 2.5, 3.8};
	Number[] expected = {1.0, 2.5, 3.8};
	
	Number[] result = DataUtilities.createNumberArray(input);
	
	assertArrayEquals("The created array should match the expected values", expected, result);
	}
	
	@Test(expected = InvalidParameterException.class)
	public void testCreateNumberArray_NullData() {
	DataUtilities.createNumberArray(null);
	}
	
	@Test
	public void testCreateNumberArray_EmptyData() {
	double[] input = {};
	Number[] expected = {};
	
	Number[] result = DataUtilities.createNumberArray(input);
	
	assertArrayEquals("The created array should be empty when input is empty", expected, result);
	}

createNumberArray2D	@Test
	public void testCreateNumberArray2D_ValidData() {
	double[][] input = {
	{1.0, 2.5, 3.8},
	{4.0, 5.5, 6.1}
	};
	Number[][] expected = {
	{1.0, 2.5, 3.8},
	{4.0, 5.5, 6.1}
	};
	
	Number[][] result = DataUtilities.createNumberArray2D(input);
	
	assertArrayEquals("The created 2D array should match the expected values", expected, result);
	}
	
	@Test(expected = InvalidParameterException.class)
	public void testCreateNumberArray2D_NullData() {

	DataUtilities.createNumberArray2D(null);
	}
	
	@Test
	public void testCreateNumberArray2D_EmptyData() {
	double[][] input = {};
	Number[][] expected = {};
	
	Number[][] result = DataUtilities.createNumberArray2D(input);
	
	assertArrayEquals("The created 2D array should be empty when input is empty", expected, result);
	}

getCumulativePercentages	@Test
	public void testGetCumulativePercentages_ValidData() {
	DefaultKeyedValues input = new DefaultKeyedValues();
	input.addValue(0, (Number) (5.0)); // Casting to Number
	input.addValue(1, (Number) (9.0));
	input.addValue(2, (Number) (2.0));
	
	KeyedValues result = DataUtilities.getCumulativePercentages(input);
	
	// Expected cumulative percentages
	DefaultKeyedValues expected = new DefaultKeyedValues();
	expected.addValue(0, (Number) (0.3125)); // 5 / 16
	expected.addValue(1, (Number) (0.875)); // (5 + 9) / 16
	expected.addValue(2, (Number) (1.0)); // (5 + 9 + 2) / 16
	
	// Compare result with expected
	assertEquals("The cumulative percentages should match the expected values", expected, result);
	}
	
	@Test(expected = InvalidParameterException.class)
	public void testGetCumulativePercentages_NullData() {
	DataUtilities.getCumulativePercentages(null);
	}
	
	@Test
	public void testGetCumulativePercentages_EmptyData() {
	DefaultKeyedValues input = new DefaultKeyedValues();
	
	KeyedValues result = DataUtilities.getCumulativePercentages(input);
	
	// Check if the result is empty using getItemCount()
	assertEquals("The result should be empty for empty input data", 0, result.getItemCount());
	}
	}