**Course: Software Testing**

**Lab. Report #2 – Automated Requirements-Based API Unit Testing using JUnit**

|  |  |
| --- | --- |
| Group #: | 100 |
| Student Names: | Cerys Lundy |
| Mark Woods |

# Unit testing plan

## List of the methods under test

Range class

* combine
* constrain
* expandToInclude
* getLength
* toString

DataUtilities

* calculateColumnTotal
* calculateRowTotal
* createNumberArray
* createNumberArray2D
* getCumulativePercentages

## The chosen black-box test technique(s) and why you have decided to choose them

Equivalence-class partitioning, Boundary Value Analysis and Decision Tables are techniques chosen for this black-box testing project. The rationale for choosing these particular techniques is outlined below.

**Equivalence-Class Partitioning**

Equivalence partitioning divides the input domain into partitions such that each class is expected to exhibit similar behavior. It was chosen because the technique is very powerful in reducing the number of test cases required while ensuring high coverage of various types of input data. With respect to this particular testing challenge, functions within the Range class such as ‘combine’, ‘getLength’ and ‘toString’ lend themselves well to this technique because their response to input data is largely uniform aside from small variations on the input class such as integer or double, positive or negative values.

**Boundary Value Analysis**

Boundary value analysis focuses on testing the boundaries of the equivalence-classes discussed in the previous section. It was chosen because it is understood that these boundaries between classes is where the errors often occur meaning, for certain applications, it is more effective than equivalence class partitioning. It is particularly useful for functions that can take continuous forms of input e.g. integers or doubles but whose output varies by category. The ‘constrain’ function within the Range class, for example, provides a perfect example of this technique to be applied with effect.

**Decision Tables**

Decision tables are a tabular representation of various input and output combinations. It can borrow input selection from the previous two methods, and it is of particular value when testing more complex functions that depend on multiple variables to effect various actions or outputs. This should lead to the test plan being more comprehensive and effective. An example function in the Range class where this technique proves useful is the ‘expandToInclude’ function. The multiple inputs and various outputs depending on the order of the inputs could prove hard to track without the use of a table.

## How the team plans to organize their JUnit test suites (based on Appendix C in lab doc)

The team will structure their test suite such that it is comprised of several JUnit4 files contained withing ‘test’ directory. The team has defined a series of rules, outlined below.

* Each file within will concern the testing of one corresponding class in the SUT e.g. ‘KeyedObjectTest.java’ will test the ‘KeyedObject.java class’.
* Each test file will contain at least one function testing a corresponding method in the class under test (CUT). These will be prefixed with the ‘@Test’ tag, provided by JUnit4, and will describe the CUT and the test being performed e.g. equalsReturnTrueWhenInputsAreEqual(). It was decided that the location and name of the test file and the ‘@Test’ prefix to the function make it sufficiently obvious the method’s purpose is to execute a test.
* Each test function will make use of the various assertions provided within the JUnit 4 framework. An exception will, however, be made for testing if exceptions are thrown, in which case, the use of the ‘expected’ variable in the ‘@Test’ tag will be exercised, as seen in the exception example below.

Exception Example

@Test (expected=NullPointerException.class)

Void myMethodThrowsExceptionWhenInputIsNull(){

Utils.myMethod(null);

}

# Designing the unit test-cases using black-box test-case design techniques

* **Make sure to include, in your report: HOW you used the black-box test-case design techniques (such as equivalence classes, and boundary value analysis)**
* When applying these techniques, make sure to explicitly follow the steps discussed in the class, e.g., first derive the domain for each input variable, then the equivalence classes, etc. just like how we learned and practiced with them in the lectures.
* You should not include any test code in this section, but only the design of the test cases using the above methods, before coding them.

## SUT Class DataUtilities

#### calculateColumnTotal(Values2D data, int column)

Details steps of the test-case design method:

**Note:** if you apply boundary-value analysis / testing (BVA) and other test-case design techniques, you should also add their details below

1-Identify the input domain (of method/function parameters):

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

#### calculateRowTotal(Values2D data, int row)

1-Identify the input domain (of method/function parameters):

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

#### createNumberArray(double[] data)

#### createNumberArray2D(double[][] data)

#### getCumulativePercentages(KeyedValues data)

## SUT Class Range

#### Range combine(Range range1, Range range2)

For the testing of this function, equivalence-class partitioning will be employed.

1-Identify the input domain (of method/function parameters):

The input domain is two Range objects which in essence is two sets of upper and lower bound double values.

A number line with numbers and lines

Description automatically generated with medium confidence2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

Figure 2.2.1.1

There are 7 different test cases for this function, with respect to how the two ranges are arranged relative to each other, they can be seen visually in figure 2.2.1.1. They are also listed below.

1. Range A is lower and distinct than range B.
2. Range A shares an upper boundary with Range B’s lower boundary.
3. Range A’s upper boundary is contained within Range B but it’s lower boundary lies outside Range B.
4. Range A is wholly contained within Range B.
5. Range A’s lower boundary is contained within Range B bit its upper boundary lies outside Range B.
6. Range A shares a lower boundary with Range B’s upper boundary.
7. Range A is greater and distinct than range B.

The domain in which the input data also must flex incorporating the following domains.

* Positive value
* Negative value
* Zero Value
* Double Value
* Integer Value
* Null Value

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

The combination of these two areas of test, relative range positions and input domain, have been combined to make a series of weak equivalence class tests. Weak equivalence class was selected because these two areas of test are likely to be independent of each other, meaning both can be tested at the same time. As this is the case, weak equivalence class testing limits the number of test cases, thus increasing the code base efficiency and maintainability with minimal reduction to test coverage.

The specific test cases are outlined below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Range A | | Range B | | Expected Output |
| Lower Limit | Upper Limit | Lower Limit | Upper Limit |
| 1 | -10 | -5 | 0 | 5 | Range(-10, 5) |
| 2 | -11.5 | -5.2 | -5.2 | 0.2 | Range(-11.5, 0.2) |
| 3 | -20.2 | -10 | -12 | -7.6 | Range(-20.2, -7.6) |
| 4 | 10 | 20 | 0 | 30 | Range(0, 30) |
| 5 | 7.5 | 12.6 | 5.2 | 10.2 | Range(5.2, 12.6) |
| 6 | 10 | 20.5 | 0.4 | 10 | Range(0.4, 20.5) |
| 7 | 10 | 20 | -10 | 0 | Range(-10, 20) |
| 8 | Null | Null | 12.2 | 50.9 | Range(12.2, 50.9) |
| 9 | -7.5 | 2.2 | Null | Null | Range(-7.5, 2.2) |

#### double constrain(double value)

1-Identify the input domain (of method/function parameters):

The input domain of this function is a double and is called upon a Range object that had double upper and lower bound values.

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

There are 5 different test cases for this function, with respect to how the value is arranged relative to the range object, they can be seen visually in figure 2.2.1.2. They are also listed below.

1. Value is greater than range upper boundary.
2. Value is on range upper boundary.
3. Value is contained within the range.
4. Value is on the range lower boundary.
5. A line of numbers and lines with dots and lines

   Description automatically generated with medium confidenceValue is lesser than the range lower boundary.

Figure 2.2.1.2

In addition, there exists some edge cases that must also be tested. These are outlined below.

* Range of 0 length (lower and upper bound are the same value)
* Positive double value
* Negative double value

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

The combination of these two areas of test, relative range and value position and input domain, have been combined to make a series of weak equivalence class tests. Weak equivalence class was selected because these two areas of test are likely to be independent of each other, meaning both can be tested at the same time. As this is the case, weak equivalence class testing limits the number of test cases, thus increasing the code base efficiency and maintainability with minimal reduction to test coverage.

The specific test cases are outlined below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Range | | Value | Expected Outcome |
| Lower Value | Upper Value |
| 1 | 10.5 | 20.2 | 30.6 | 20.2 |
| 2 | -10.4 | -5.6 | -5.6 | -5.6 |
| 3 | 5 | 10 | 7 | 7 |
| 4 | -10 | -5 | -10 | -10 |
| 5 | 2 | 5 | 0 | 2 |
| 6 | 2 | 2 | 6 | 2 |

#### Range expandToInclude(Range range, double value)

1-Identify the input domain (of method/function parameters):

The input domain of this function is a double and a Range object that has double upper and lower bound values.

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

There are 5 different test cases for this function, with respect to how the value is arranged relative to the range object, they can be seen visually in figure 2.2.1.3. They are also listed below.

1. Value is greater than range upper boundary.
2. Value is on range upper boundary.
3. Value is contained within the range.
4. Value is on the range lower boundary.
5. A line of numbers and lines with dots and lines

   Description automatically generated with medium confidenceValue is lesser than the range lower boundary.

Figure 2.2.1.3

In addition, there exists some edge cases that must also be tested. These are outlined below.

* Range of 0 length (lower and upper bound are the same value)
* Positive double value
* Negative double value

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

The combination of these two areas of test, relative range and value position and input domain, have been combined to make a series of weak equivalence class tests. Weak equivalence class was selected because these two areas of test are likely to be independent of each other, meaning both can be tested at the same time. As this is the case, weak equivalence class testing limits the number of test cases, thus increasing the code base efficiency and maintainability with minimal reduction to test coverage.

The specific test cases are outlined below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Range | | Value | Expected Outcome |
| Lower Value | Upper Value |
| 1 | 1.0 | 3.0 | 2.0 | Range(1.0, 3.0) |
| 2 | 1.0 | 3.0 | 0.5 | Range(0.5, 3.0) |
| 3 | -3.0 | -1.0 | 4.0 | Range(-3.0, 4.0) |
| 4 | -1.0 | 3.0 | -1.0 | Range(-1.0, 3.0) |
| 5 | -1.5 | 0 | 0 | Range(-1.5, 0) |
| 6 | 5.0 | 5.0 | 3.0 | Range(3.0, 5.0) |
| 7 | Null | | 5.0 | Range(5.0, 5.0) |

#### double getLength()

1-Identify the input domain (of method/function parameters):

The domain of this input is a Range object which is, in essence, double upper and lower bound values.

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

There are 4 classes that need to be tested in this function. They are positive values, negative values, zero values and null values. Any other form of values are validated in the Range class constructor.

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

A strong equivalence-class testing approach was taken to test this function due to the low number of test cases needed to achieve this, due to the small number of equivalence classes. See the test cases outlined in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Range | | Expected Output |
| Lower Bound | Upper Bound |
| 1 | 5.0 | 10.0 | 5.0 |
| 2 | -5.0 | -10.0 | 5.0 |
| 3 | 0.0 | 0.0 | 0.0 |
| 4 | Null | | NullPointerException |

#### toString()

1-Identify the input domain (of method/function parameters):

The domain of this input is a Range object which is, in essence, double upper and lower bound values.

2-Equivalence classing of method input(s): shall be presented "visually", like done in the lectures

There are 4 classes that need to be tested in this function. They are positive values, negative values, zero values and null values. Any other form of values are validated in the Range class constructor.

3-Combining equivalence classes of the different inputs, using the multi-dimensional approach, i.e., Strong Equivalence-Class Testing (SECT)

A strong equivalence-class testing approach was taken to test this function due to the low number of test cases needed to achieve this, due to the small number of equivalence classes. See the test cases outlined in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Range | | Expected Output |
| Lower Bound | Upper Bound |
| 1 | 5.0 | 10.0 | 5.0 |
| 2 | -10.0 | -5.0 | 5.0 |
| 3 | 0.0 | 0.0 | 0.0 |
| 4 | Null | | NullPointerException |

# Output of test suite execution: Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status (showing the names of test methods)

Include a screenshot of ALL test suite execution in JUnit showing their Pass/Fail/Error status, such as:

A screenshot of a computer

Description automatically generated

**(Note: This is just an example. We are NOT providing the number of test cases for you.)**

# Based on the list of failed test cases (failures) in the previous section above, discuss the possible faults leading to those failures

Hint: Use the chain of software dependability threats: errors, fault, failure, as learned in the lectures

# How the team work/effort was divided and managed

## How the team work/effort of the lab was managed and divided

* You can say for example discuss which parts of the lab-work (e.g., classes under test, etc.) was done by who…
* And also discuss the meetings that you had to plan and run the lab work
* Etc.

## Writing the lab report

Fill up the following table to specify who wrote what part of the lab document:

|  |  |
| --- | --- |
| **Lab-report section** | **Written by** |
| 1- Introduction | Student A |
| 2-.. |  |
| … |  |

## Lessons learned from your teamwork in this lab

Only include lessons learned from **your teamwork in this section**. **“Technical”** lessons learned **shall be discussed in another section below.**

# Difficulties/ challenges encountered, overcoming them, and lessons learned

## Difficulties/ challenges encountered

Text…

## How did you overcome the above difficulties/ challenges?

Text…

## “Technical” Lessons learned

Only include **“technical”** lessons learned from **in this section**. Lessons learned **your teamwork shall be discussed in another section above.**

# Comments/feedback on the lab and lab document itself

This section has the following sub-sections.

## Did you find the lab a useful learning experience? How it helped you learn the new testing topics

Text…

## Was the lab document easy to follow?

Text…

## About time budget? (Was there too much/too little time for this lab?)

Text…

## Please provide your comments on how to improve the lab work and lab document

Text…