**COMP 3505 - Software Testing**

**Assignment. Report #4 – Code Coverage & White-box testing**

|  |  |
| --- | --- |
| Group #: | 4 |
| Student Names: | Kevin Ruiz |
| Ryan Wong |
| Charlemagne Palisoc |

# **A detailed report of the coverage achieved of each class and method**

org.jfree.data.DataUtilities

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Counter** | **Coverage %** | **Covered** | **Missed** | **Total** |
| *Instructions* | 95.6 | 174 | 8 | 182 |
| *Branches* | 88.5 | 23 | 3 | 26 |
| *Lines* | 95.3 | 41 | 2 | 43 |
| *Methods* | 83.3 | 5 | 1 | 6 |
| *Types* | 100.0 | 1 | 0 | 1 |
| *Complexity* | 78.9 | 15 | 4 | 19 |

Notes:

All individual methods in DataUtilities class have 100% coverage in all counters except for GetCumulativePercentages(KeyedValues) testing. This brings down the overall coverage percentages for the class below minimum coverage. At the time of completion, our team was unable to correct the testing methods to pass a null value and increase the coverage to minimums.

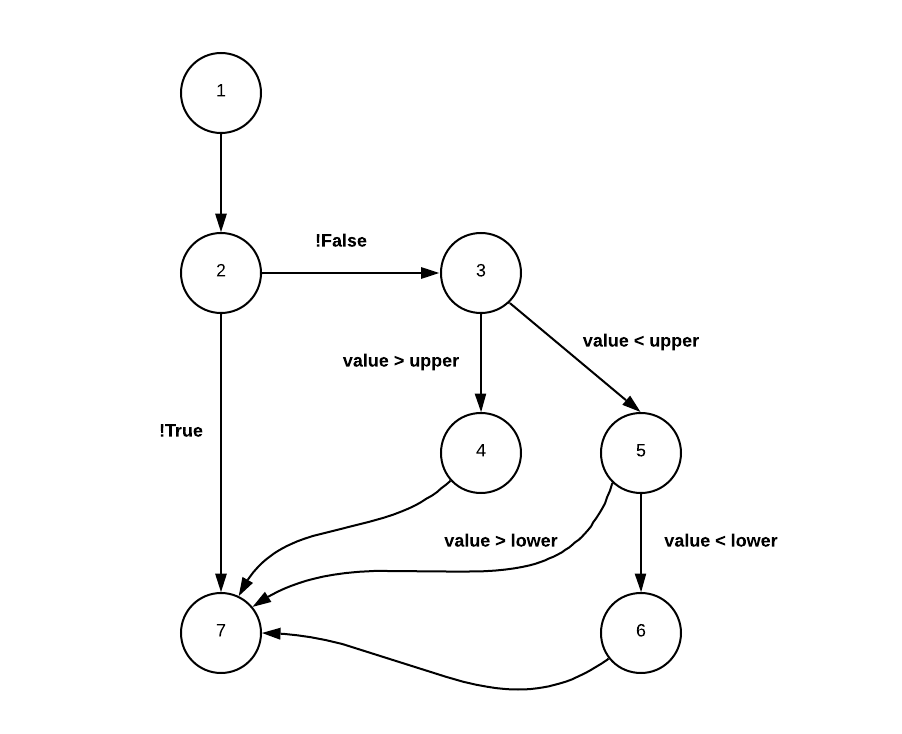
org.jfree.data.Range

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Counter | Coverage % | Covered | Missed | Total |
| Instructions | 99.4 | 333 | 2 | 335 |
| Branches | 93.2 | 41 | 3 | 44 |
| Lines | 100.0 | 73 | 0 | 73 |
| Methods | 100.0 | 17 | 0 | 17 |
| Types | 100.0 | 1 | 0 | 1 |
| Complexity | 92.3 | 36 | 3 | 39 |

org.jfree.data.DefaultKeyedValues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Counter | Coverage % | Covered | Missed | Total |
| Instructions | 98.3 | 285 | 5 | 290 |
| Branches | 80.6 | 29 | 7 | 36 |
| Lines | 97.7 | 85 | 2 | 87 |
| Methods | 100.0 | 18 | 0 | 18 |
| Types | 100.0 | 1 | 0 | 1 |
| Complexity | 83.3 | 30 | 6 | 36 |

# **Manual data-flow coverage calculations for Range.constrain(double) method**



|  |  |  |  |
| --- | --- | --- | --- |
| Control-Flow Graph | | | |
| **Node** | **Defines** | **c-uses** | **p-uses** |
| 1 | value, result, upper, lower | value |  |
| 2 |  |  | value |
| 3 |  |  | value, upper |
| 4 | result | upper |  |
| 5 |  |  | value, lower |
| 6 | result | lower |  |
| 7 |  | result |  |

|  |  |  |
| --- | --- | --- |
| Definition-clear- use Paths | | |
| **Node** | **dcu(v,i)** | **dpu(v,i)** |
| 1 | dcu(result, 1) = { 7 }  dcu(upper, 1) = { 4 }  dcu(lower, 1) = { 6 } |  |
| 2 |  | dpu(value,1) = { (2,3), (2,7) } |
| 3 |  | dpu(upper, 1) = { (3,4), (3,5) }  dpu(value, 1) = { (3,4), (3,5) } |
| 4 | dcu(result, 4) = { 7 } |  |
| 5 |  | dpu(value, 1) = { (5,6), (5,7) } |
| 6 | dcu(result, 6) = { 7 } |  |
| 7 |  |  |
| **Total** | **5** | **8** |

# **If you have added any new test case in this step of the assignment, include one of them in your assignment report and describe how this test was not possible to be developed based on Black-box testing**

*@Timeout*(5)

*@Tag*("unitTest")

*@Test*

void testSortByKeysAscending() {

DefaultKeyedValues expected = new DefaultKeyedValues();

expected.addValue("1", 1);

expected.addValue("2", 2);

expected.addValue("3", 3);

value = new DefaultKeyedValues();

value.addValue("1", 1);

value.addValue("3", 2);

value.addValue("2", 3);

SortOrder asc = SortOrder.***ASCENDING***;

value.sortByKeys(asc);

*assertTrue*(expected.equals(value));

}

Testing the sortByKeys method in org.jfree.data.DefaultKeyedValues class would not have been testable in our team’s opinion with just black-box testing. Without actual white box source code information on the use of the SortOrder object parameter and the ability to search that java.util’s function, we would not have known what the expected output should be. By analyzing the source code we were able to understand that 3 tests would be needed to check the possible expected outcomes of each passed SortOrder object- an ascending, descending and unsorted instance.

# **A comparison on the advantages and disadvantages of requirements-based test generation and coverage-based test generation.**

Requirements-based testing (RBT) is a testing approach in which test cases, conditions, and data derived from requirements, while coverage-based testing (CBT) is determining how much is being tested. The advantage that coverage-based testing has is the ability to check a part of the code that could contain a bug. CBT also helps measure how much unit testing has been done and how much is left. In contrast, RBT has certain cases where whatever test case one writes, certain faults always go undetected.

# **Any difficulties encountered, challenges overcome, and lessons learned from performing the assignment**

We had some real difficulties with setting up our environment in the beginning but that was resolved quickly enough. What was challenging is making sure that our test cases were covering the methods and classes. When we began we had less coverage for multiple classes and methods and missed a lot of instructions/branches in our initial test cases. Refactoring the test code to include the missing branches was sometimes hard because we couldn’t figure out what was wrong.

Another challenge was creating the control flow graph and definition clear graph tables, we had some issues trying to understand what was shown in the assignment appendix, and it took sometime before we can wrap our heads in what we had to do, after that it became much easier to apply the steps in other methods.

The assignment required some extra time as we had to also work on our older test code and update them as well as include some more tests to meet and exceed the coverage test. The lesson is that modifying older code often takes more time than writing a new one, especially if the code lacks some documentation. At the same time, we often have some reluctance in modifying the code as we believe that we have written them correctly.

# **Comments/feedback on the assignment itself (Optional)**

* The assignment was challenging in terms of analysis and implementation but a good way to revisit and improve unit testing.
* More time may have allowed our team to improve more faults in our original tests, but we finished fairly satisfied with the outcomes for our coverage and general improvements.