Washington State University

422 Deliverable 2

Test Report

Team members

Yuer Shen (SID: 11421959)

Courtney Snyder (SID: 11463078)

Ran Tao (SID: 11488080)

CptS 422 Software Engineering Principle II

Spring 2017

Instructor: Venera Arnaoudova

Black Box Testing

For our black box testing technique, we decided to use the equivalence class partitioning method. Originally, we wanted to use category partition testing, as it is arguably the strongest black box testing method since it includes both equivalence class partitioning and boundary value analysis. We separated our data into two classes, Operators and Operands. Since Difficulty, Length, Vocabulary, and Volume are all dependent on operators and operands. We did not deem it necessary to partition those classes since they calculate their data using the maps in Operator and Operand objects.

The elements in the operand class are types of variables and the elements for the operators class are types of operators. We determined that the possible types of operands are numerical, alphanumerical, object, or other. A numerical type is that of an integer, double, floating point, or any other non-string number. An alphanumerical type is a character; we originally wanted to add string to this definition, but we realized that strings are objects in Java. An object can be user defined, or predefined in a library. An other type is non-numerical, non-alphanumerical, and non-object, such as a boolean. In our implementation for Deliverable 1, we used a definition of operators that included 35 different operators.

After closer examining our data, we realized that there is no distinct boundary between the elements in our equivalence classes. For example, there is no way for an operand to be a “minimal” integer or a “maximal” integer; that operator can have minimal or maximal values, and those values can be tested, but its type is still an integer. Similarly, there is no way for an operator to be a “minimal” or “maximal” greater-than sign; the operator simply is a greater-than sign.

Given this lack of boundaries, we determined that the Strong Equivalence Class Testing (SECT) method was more or less equivalent to the All Combinations (AC) method used in category partition testing. We then decided to use Strong Robust Equivalence Class Testing over SECT because we wanted to be more thorough and include invalid cases, as the checkstyle plugin can run even with compiler errors.

We looked at two kinds of test cases: one operand with one operator and two operands with one operators. We did not look at multiple operators because we looked at the way the computations are made with three operands and two operators; 2 + 3 + 5 becomes 2 + 8 becomes 10. This philosophy can be applied for 4, 5, 6, and n operators; if the operator and operand combination is valid, the term will simplify.

We came up with 700 test cases total; since we have 35 operator types and 4 operand types and looked at the two kinds of test cases:

140 test cases for single operand and single operator (eg integer++)

560 test cases for double operand and single operator (eg string + char)

White Box Testing

For our white box testing technique, we decided to use a control flow graph with data flow analysis. We chose data flow analysis because it is stronger than control flow coverage since we can analyze path coverage, and within that, we can ensure that all “useful” nodes are executed by defining definition and use occurrence. Also, data flow analysis allows us to see how the data changes line by line, which makes it easier to identify where an error may occur..

We decided to use two control flow graphs; one for operator behavior and one for operand behavior. We decided to only monitor these classes because as previously mentioned, Difficulty, Length, Vocabulary, and Volume are dependent on the Operator and Operand classes and calculate their totals based off of the map in the Operator and Operand objects. For each graph, we focus on data flow analysis; specifically, definition, predicate use, computation use. This ensures that all lines are executed and that variables are being assigned and used appropriately using in each branch of the code.

Theoretically, even if we cannot cover all possible paths that data can take in our system, we can make sure that data behaves appropriately in the significant paths. We will do this by determining all definition and use occurrences, and then implementing all-use criterion to ensure that all definition-use pairs are visited. Testing these use cases and edge cases will increase our confidence in the system.

We covered all definition and use cases in the operand control flow graph with one test case that includes all types of operands. We were able to cover all definition, predicate, and computational uses in one test case because there are only three if statements in the class; one on line 57, one on line 61, and one on line 65. That means there are minimal paths for the data to take. The first if statement is the base case for recursion, so it is always reached. The second if statement looks at the type of the operand for Object, Alphanumeric, and Numeric. All operands of these acceptable types must reach this line. Finally, the third if statement adds the operand to the map if it is not already in the map so that we can keep track of unique operands. So, all if statements are covered with one test case with valid operands.

Similarly, we covered all definition and use cases in operator control flow graph with one complex test cases. Again, we were able to do this in one test case because there are minimal if statements; one on line 51, one on like 54, and one on line 77. The first if statement is the base case for recursion, so it is always reached. The second if statement determines the type of operator out of the 35 possibilities. The final if statement adds the operator to the map if it is not in it already so that we can keep track of unique operators.

Conclusion

After using both white and black box testing techniques, we feel confident in our checkstyle plugin.

With black box testing, we tested all combinations of operators and operand types and determined which ones were valid and invalid.

With white box testing, we were able to get about 100% code coverage for satisfying the all-uses criterion; this makes us feel confident in those classes. In addition, Difficulty, Length, Vocabulary, and Volume are dependent on Operator and Operand, and those classes reference the result of operators and operand classes to calculate.