Code Coverage Using EMMA

By Team Voltron

Thomas Bassa, Gregory Carkin, Umar Idris, Michael Philotoff

Introduction

The purpose of this project was to learn how to improve code coverage by building new test cases, and how to deal with potentially overlapping or interfering tests. Furthermore, this project was about learning how to write better test cases to get the maximum coverage out of a test case and how to group test cases together by what part of the of the code they are focused on. Additionally, it helped the few of us who are somewhat new to JUnit tests to understand how they work, and how to build successful JUnit test cases.

Coverage Analysis

Per EclEmma, we were able to achieve 97.9% coverage of the class under test.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TaxCalculatorCodeCoverage	95.9 %	3,158	135	3,293
src	95.9 %	3,158	135	3,293
test	95.5 %	2,589	123	2,712
application	97.9 %	569	12	581
TaxCalculator.java	97.9 %	569	12	581
TaxCalculator	97.9 %	569	12	581
TaxCalculator(String, int, int, int)	91.5 %	54	5	59
isReturnRequired()	94.4 %	85	5	90
getTaxDue()	99.1 %	209	2	211
TaxCalculator(String, int, int)	100.0 %	51	0	51
getAge()	100.0 %	3	0	3
getFilingStatus()	100.0 %	3	0	3
getGrossIncome()	100.0 %	3	0	3
getName()	100.0 %	3	0	3
getNetTaxRate()	100.0 %	8	0	8
getSpouseAge()	100.0 %	3	0	3
getStandardDeduction()	100.0 %	70	0	70
getTaxableIncome()	100.0 %	47	0	47
setGrossIncome(double)	100.0 %	26	0	26
setName(String)	100.0 %	4	0	4

Complete, 100% code coverage was not possible, as some of the else statements that were covered did not read as covered even though the switch statements within the else statements were covered. Also, within one of the methods, there was a human error the logic of an if statement; the length of a name had to be less than 0 in order to reach some code, but it is impossible to have a string length less than 0. Those statements could be covered if the check considered strings of length less than or equal to 0, so an empty string would trigger it. Furthermore, many of the methods contained switch statements with unreachable default cases. The constructors properly ensure that the filingStatus variable is only one of several constants defined in TaxCalculatorInterface, making switches based off all of those values unable to fire a default case.

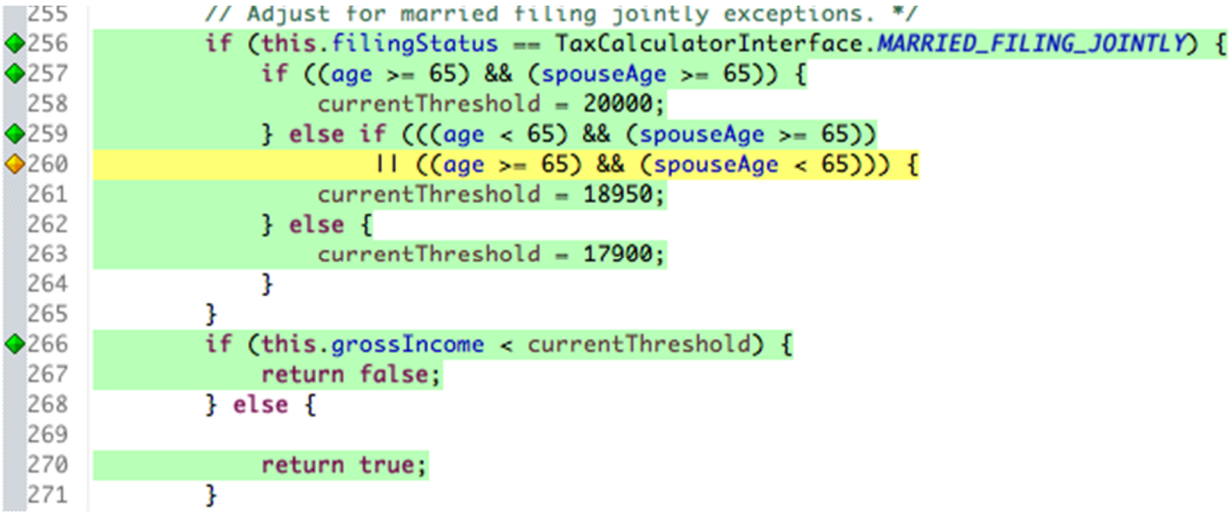
Conclusions

Umar

Methods Covered:

- getName()
- setName(String)
- getFilingStatus()
- isReturnRequired()

The isReturnRequired() method was a bit challenging to cover. It contains a few nested if statements and a switch case too. Knowing that the default case cannot be reached, I could not get a 100% coverage on that method. Everything else was covered except for a single branch of an else if statement in the method, which was impossible to reach, leaving the isReturnRequired() method with a 94.4% total coverage. (See below screenshot.)



```
255         // Adjust for married filing jointly exceptions. */
256         if (this.filingStatus == TaxCalculatorInterface.MARRIED_FILING_JOINTLY) {
257             if ((age >= 65) && (spouseAge >= 65)) {
258                 currentThreshold = 20000;
259             } else if (((age < 65) && (spouseAge >= 65))
260                     || ((age >= 65) && (spouseAge < 65))) {
261                 currentThreshold = 18950;
262             } else {
263                 currentThreshold = 17900;
264             }
265         }
266         if (this.grossIncome < currentThreshold) {
267             return false;
268         } else {
269
270             return true;
271         }
```

Covering the above method automatically covered the getFilingStatus() method because the method was being called in the isReturnRequired(). The getName() and setName(String) were very easy. Hence, being the first time I have used this tool to test a program, it was an enjoyable experience. It sure is more efficient than assuming your code works well.

Greg

Methods Covered:

- TaxCalculator(String, int, int)
- getTaxableIncome()

From creating test cases for the TaxCalculator method, I learned that in order to cover as much code as possible all branch cases must be tested. The following portion of code depicts this:

```
51          // Check that the filing status is valid for a person who does not have
52          // a spouse.
53          if ((filingStatus != SINGLE) && (filingStatus != HEAD_OF_HOUSEHOLD)
54                  && (filingStatus != QUALIFYING_WIDOWER)) {
55              throw new Exception("Invalid filing status for this constructor.");
56          }
```

As seen above, the if statement has multiple points where it can branch. One branch would be all the conditions are true. This would cause the program to go through and throw an exception. However, in order to get 100% coverage test cases must be made for when each condition is true, and each condition is false.

After having done the TaxCalculator method, the getTaxableIncome method did not present much of a problem. In order to achieve 100% coverage in it the grossIncome had to be compared against the standardDeduction in all possible ways. The test cases needed to cover if grossIncome was greater than, less than, or equal to standardDeduction. Additionally there needed to be a test case to check if an error occurred from the comparison. Overall, this was a good intro to using the tool and figuring out how to develop test cases that would cover as much code as possible.

Michael

Methods Covered:

- TaxCalculator(String, int, int, int)
- setGrossIncome()

One the most valued things learned from doing these test cases is that sometimes it is impossible to cover some of the test possibility due to a human error within the code, as shown in the method below.

```
88      public TaxCalculator(String name, int filingStatus, int age, int spouseAge)
89              throws Exception {
90          // Check the validity of all parameters.
91          if (name.length() < 0) {
92              throw new Exception(
93                      "Name must be longer than 0 characters in length. ");
94          }
```

Other than that lesson learned, I also learned that when doing code cover you must think of how the previous test affects the test case after as shown from the build in test case that was being affected by the previous test case changing the data that the next test case was going to use. Thus causing the test case to have an error when trying to execute because it would throw an exception and never execute the portion it was trying to execute.

Thomas

Methods Covered:

- getStandardDeduction()
- getTaxDue()

These final two methods were very straightforward to cover, only a matter of covering a few missed branches with new tests. My takeaways from this project are that 100% coverage is unrealistic, and that coverage does not imply quality testing.

The irony of 100% coverage is that making perfectly reachable code can make one more prone to creating faults by leaving out fallback cases. There were likely many people who wanted to strip out the default cases in the switch statements to increase coverage, but that arguably reduces the quality of the code; if the code changes elsewhere in a way that makes the defaults reachable, not having them could be catastrophic. With that said, coverage analysis is a good way to uncover logically unreachable code, such as in Michael's example.

The other thing I noticed while writing test code for this project was that achieving additional coverage was incredibly easy, yet it did not seem to add substantial value to the test code in my opinion. It is possible to achieve incredibly high coverage with a well-written set of loops inside a test that fails upon catching an exception, but passing doesn't tell one much about whether the tested code is logically correct. It is simple to cover code with a basic understanding of the language the tests are written in, but without an understanding of the requirements of the tested code, writing meaningful test cases is unlikely.