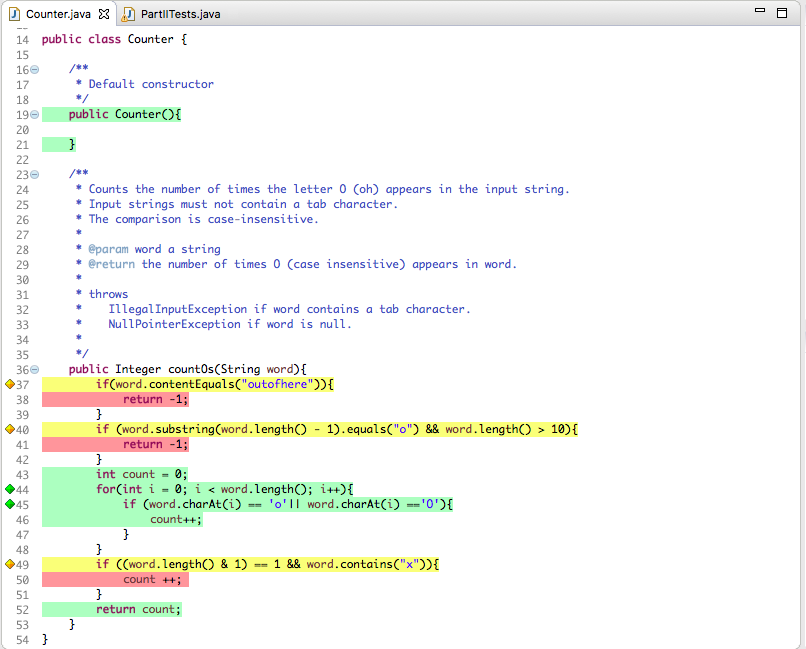
Annie Steenson

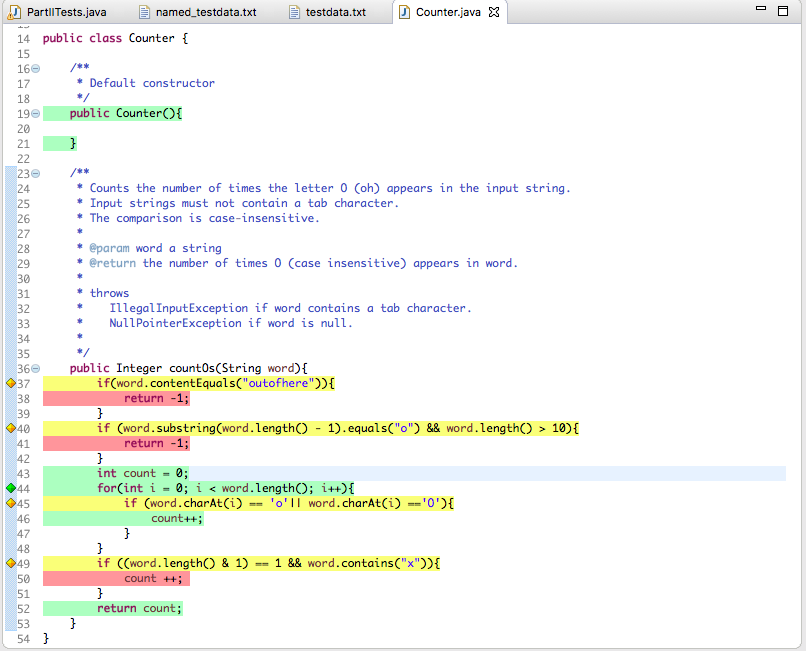
Lab 3

**Instructions Documentation**

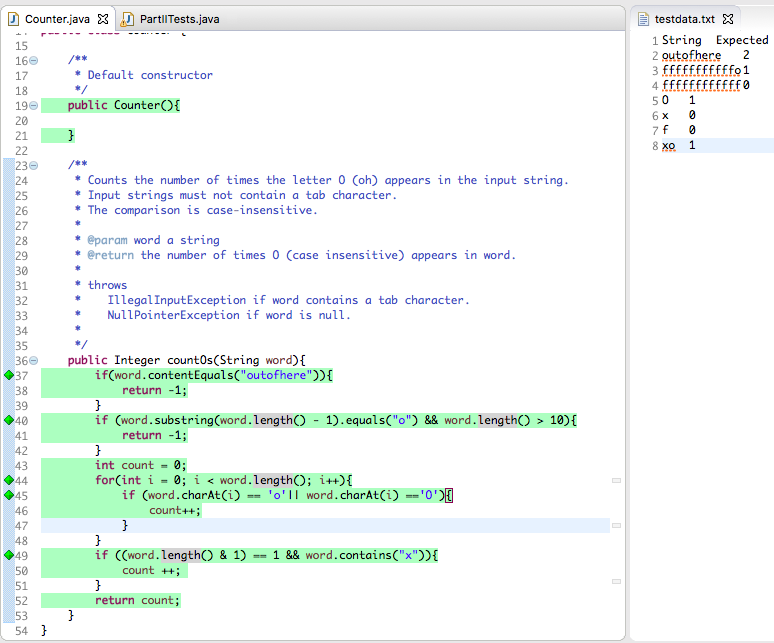
**3) Run code and conditional coverage on the Part II test suite which you turned in with lab2 (capture the results in a screen shot);**



**4) Reusing your Parameterized test class from Lab 2 (see the related instructions below), and a new test data file with only the test case “O 1”, rerun the test, paying attention only to conditional statements that do not show full coverage.**



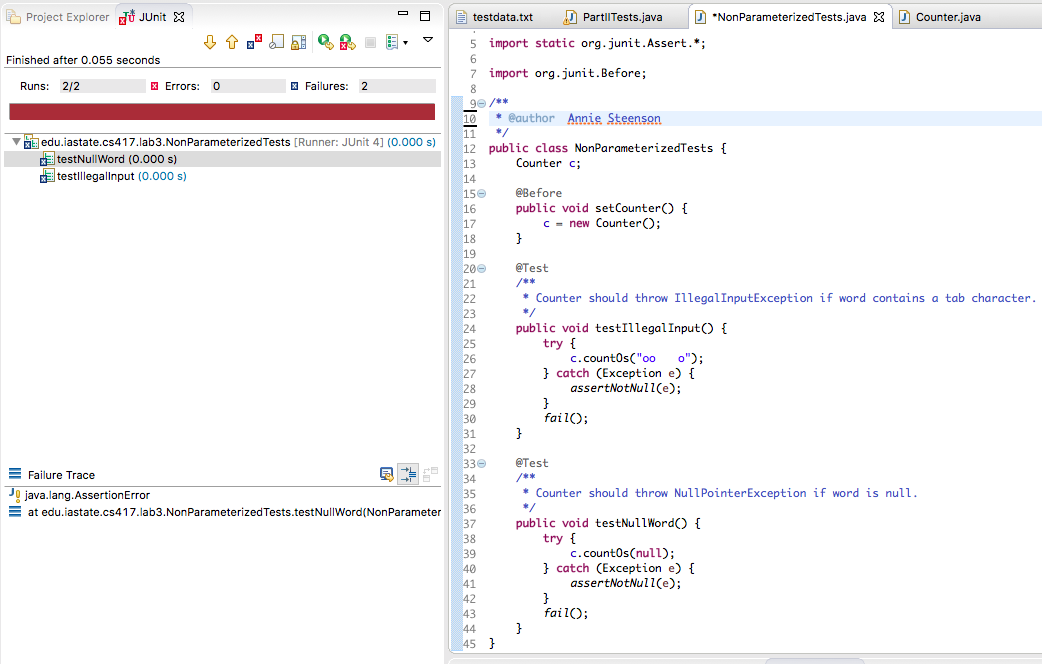
**5) Based only on the "missing" CONDITIONAL coverage in step 4, select more tests, add them to the test data file, and repeat until your test suite covers completely as many of the conditional statements as you can. When finished, record the final coverage in a second screen capture.**



**6) Now write separate non-parameterized tests (in a separate test class) for any requirements that have not been covered by your Parameterized tests. If you wrote such tests for Lab 2, Part II, you may reuse them here. Name and document these tests so that it is clear to us what fault you mean for the test to expose.**

I named this file: “NonParameterizedTests.java”

**7) Submit your documented results.**



**Report Documentation**

**Create a report that answer the following questions:**

**1. Were you able to find more bugs using the conditional coverage? Why?**

Yes, I was able to find more bugs using conditional coverage. I was able to because I could see what the program was explicitly checking for in each if statement. For example: I could see that one if statement was checking for the specific string, “outofhere.” Since the string can be infinitely long, there are an infinite amount of possible strings that could be used as input. Using conditional coverage, I was able to minimize the number of strings I needed to check for down to less than 50, which is significantly less than infinity.

**2. What kind of “conditional coverage” does EclEmma report?**

Branch coverage is defined as testing that each decision is true in at least one test case and false in at least another. EclEmma reports more than just branch coverage. EclEmma reports conditional coverage because it tests that every condition is true in at least one test case and false in another. Testing for every condition’s possibility is different from testing every decision’s possibility. The type of conditional coverage EclEmma reports is called MCDC.

**3. Would an EclEmma report showing 100% condition coverage satisfy an FAA requirement for MC/DC? Why or why not?**

**4. Were there lines you were unable to execute using only the Parameterized tests? Why?**

For step 3, I only used my original Parameterized tests and was unable to execute lines 37, 40, and in Counter.java. This is because I did not test input Strings greater than length 10 or Strings that were greater than length 10 and ended in “o”. I also did not test the String “outofhere”. I also did not test Strings of length 1 that contained “x”.   
  
For step 4, I only used the test case input, “O 1”. Similar the results and reasoning previously mentioned above about Step 3, the single test case “O 1” did not test a lot of specific cases. Those specific cases were not apparent when black box testing. However, those specific cases were very apparent when white box testing. Unlike step 3, using only the test case “O 1” did not test when the input String contained the character “o”.

For step 5, I added more test cases to more completely coverage Counter.java. After step 5, I was able to execute all lines of Counter.java. Therefore, there were no lines I was unable to execute using only the Parameterized tests from step 5.

**5. Which do they think would be a more time-consuming strategy for producing a white box test set for countOs();**

**a. using conditional coverage guidance to identify an adequate test set, or**

**b. using CFG analysis to identify an adequate test set?**

I think using CFG analysis would be more time-consuming. I think conditional coverage guidance is a lot faster because it is visual and makes specific errors (i.e. the if statement on line 37 of Counter.java) very apparent. Sometimes it is not possible to use conditional coverage. However, if possible, I think testers should use conditional coverage because it is faster to identify edge cases. I think using CFG would be more time-consuming because you would have to spend time drawing out a graph with each node and then identifying graph paths. Sure the CFG graph for Counter.java is small, but having to draw out the graph is still an extra step that isn’t completely necessary for such a small code base with simplistic branch conditionals.

**6. Which of the above two alternatives do you think would work best to select tests for a 100,000 line piece of software? Why?**

I think CFG would work best for a huge code base (i.e. a piece of software with 100,000 lines of code). I think CFG would be better because it would allow developers to use various graph path algorithms to identify specific test cases with the largest amount of code coverage. If a developer wrote a naïve test case for 100,000 lines of code, they would probably spend more time trying to cover edge cases than they would spend identifying/utilizing a CFG graph.

**7. What are the limitations or weaknesses of test sets developed using only branch coverage? (think about which bugs you didn't find using conditional coverage and about what conditionals might look like in more complicated software.)**

I did not find bugs about invalid input. Branch coverage only shows developers the branches not being coverage. However, if they didn’t write a branch to check for something (i.e. invalid input) in the first place, then branch coverage won’t show the developer that they are missing some test case. I didn’t find bugs about exceptions being thrown using conditional coverage, because there is no code to throw an exception in the SUT. Conditionals in more complicated software is probably very ambiguous and requires contextual knowledge in order to understand what is being tested in the branch.