Nhan Nguyen
CPTS 422
Del3 Report
12/8/2024

First, I had to have Pitclipse work, so I must fix certain things to get that to happen. During that time I took feedback from deliverable 1 & 2 to perform some fixes or improvements. Which is to expand the number of halstead operators & operands, and over have the tests to actually pass so Pitclipse may actually operate.

Once Pitclipse is properly set up, here are the Pit mutation results before I officially start del3.

Pit Test Coverage Report

Package Summary

My422Project

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
15	100%	576/578	86%	171/199	87%	171/197

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadDifficultyCheck.java	100%	69/69	82%	14/17	82%	14/17
HalsteadEffortCheck.java	100%	79/79	73%	29/40	73%	29/40
HalsteadLengthCheck.java	100%	71/71	100%	21/21	100%	21/21
HalsteadVocabularyCheck.java	100%	61/61	44%	4/9	44%	4/9
HalsteadVolumeCheck.java	97%	59/61	33%	4/12	40%	4/10
MethodLimitCheck.java	100%	12/12	83%	5/6	83%	5/6
NumOfCommentsCheck.java	100%	19/19	100%	13/13	100%	13/13
NumOfExpressionsCheck.java	100%	33/33	100%	6/6	100%	6/6
NumOfExternalMethodReferencesCheck.java	100%	31/31	100%	18/18	100%	18/18
NumOfLinesOfCommentsCheck.java	100%	20/20	100%	12/12	100%	12/12
NumOfLocalMethodReferencesCheck.java	100%	32/32	100%	19/19	100%	19/19
NumOfLoopingStatementsCheck.java	100%	14/14	100%	6/6	100%	6/6
NumOfOperandsCheck.java	100%	26/26	100%	7/7	100%	7/7
NumOfOperatorsCheck.java	100%	37/37	100%	7/7	100%	7/7
NumOfVariableDeclarationsCheck.java	100%	13/13	100%	6/6	100%	6/6

After making fixes and covering missing branches, here is the Pit mutation result:

Package Summary

My422Project

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
15	100%	578/578	92%	183/199	92%	183/199

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadDifficultyCheck.java	100%	69/69	82%	14/17	82%	14/17
HalsteadEffortCheck.java	100%	79/79	78%	31/40	78%	31/40
HalsteadLengthCheck.java	100%	71/71	100%	21/21	100%	21/21
HalsteadVocabularyCheck.java	100%	61/61	89%	8/9	89%	8/9
HalsteadVolumeCheck.java	100%	61/61	83%	10/12	83%	10/12
MethodLimitCheck.java	100%	12/12	83%	5/6	83%	5/6
NumOfCommentsCheck.java	100%	19/19	100%	13/13	100%	13/13
NumOfExpressionsCheck.java	100%	33/33	100%	6/6	100%	6/6
NumOfExternalMethodReferencesCheck.java	100%	31/31	100%	18/18	100%	18/18
NumOfLinesOfCommentsCheck.java	100%	20/20	100%	12/12	100%	12/12
NumOfLocalMethodReferencesCheck.java	100%	32/32	100%	19/19	100%	19/19
NumOfLoopingStatementsCheck.java	100%	14/14	100%	6/6	100%	6/6
NumOfOperandsCheck.java	100%	26/26	100%	7/7	100%	7/7
NumOfOperatorsCheck.java	100%	37/37	100%	7/7	100%	7/7
NumOfVariableDeclarationsCheck.java	100%	13/13	100%	6/6	100%	6/6

Overall, the whitebox part of this project is solid. Or at the very least, it is as solid as I could make it, considering the Mutation Coverage and Test Strength is not perfect, but close enough.

Group 1: Halstead Metrics

Halstead metrics can face issues with miscounting operators and operands, especially in complex or less obvious scenarios like ternary expressions, lambdas, or constants. Unique operators and operands may be double-counted due to case sensitivity or scoping. Difficulty and Effort calculations can also go wrong if intermediate values like fractions are mishandled or propagated incorrectly. Errors in Length or Vocabulary often carry over into other metrics like Volume and Effort.

Group 2: Comment Metrics

Comment metrics may fail to account for all inline comments, especially when mixed with code on the same line. Lines within multiline comments can also be miscounted, such as treating an entire block comment as a single line instead of counting each line it spans. Properly parsing and distinguishing comments from code is critical.

Group 3: Looping and Structural Metrics

Looping and structural metrics often encounter challenges in identifying and counting nested loops or recursive structures. Special operators like method references (::) or overloaded operators in C++ are sometimes overlooked. Expressions within control structures like if or while blocks can also be missed if parsing isn't thorough, leading to undercounted metrics.

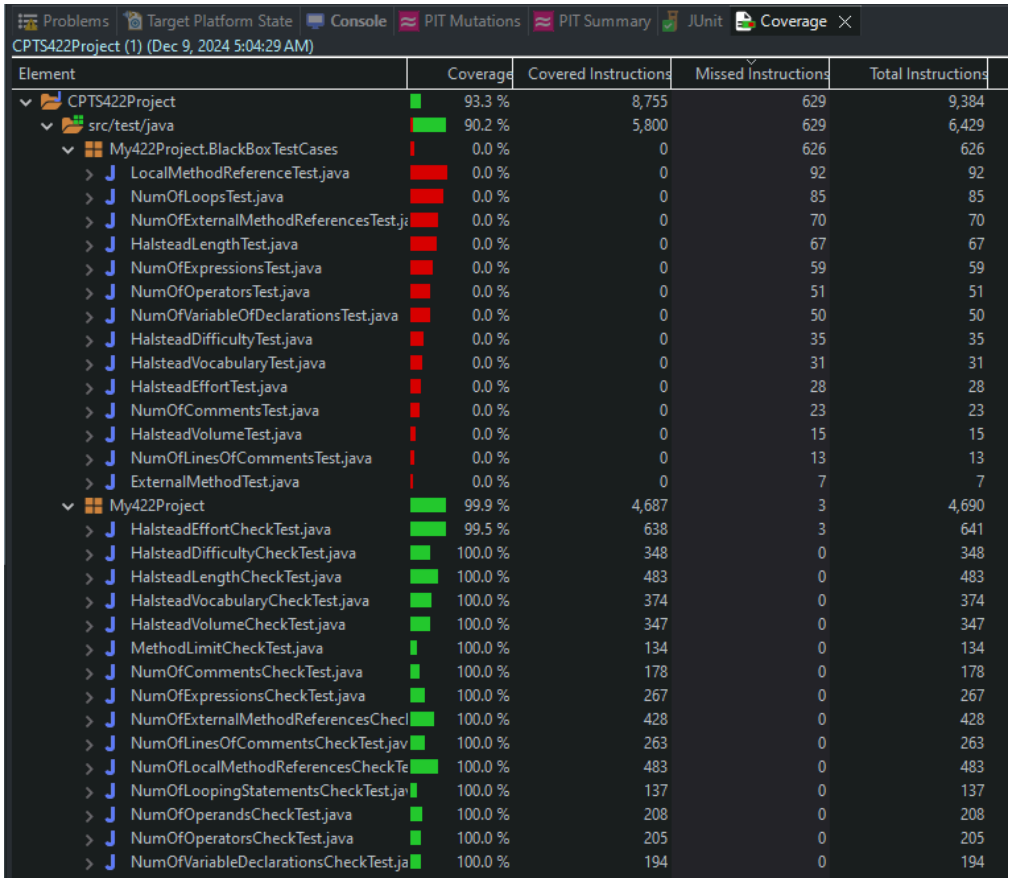
Group 4: Program References and Declarations

Counting variable declarations can be tricky if implicit declarations or shadowed variables are ignored, especially in dynamically typed languages. External method calls resolved at runtime or through reflection might be missed. Similarly, local method references invoked indirectly, such as through lambdas or callbacks, are prone to being undercounted. Properly distinguishing these elements is key to accurate metrics.

Black Box Reporting

The black box portion is definitely the most difficult part, as things get more complicated, especially with much more added files. My black box coverage is nonexistent, I believe this is because the way I set up the black box test engine led it to not communicate well with the test files. I previously thought this is due to using the same JUnit class as white box testing, but when I attempted to create a separate file for black box JUnit test, it still resulted in the same no coverage issue. After seeing no change, I did not see it was worth the time to fully make a folder of black box JUnit class testing. Oddly enough when a black box test is run, almost all of its lines run green, same with the lines in the class file as well, except for the test file, which I mentioned was the issue earlier. The part where the black box tests fail is the assertions, where all of which the actual results return zero, which indicates some odd miscommunication that I was not able to figure out. This experience has been very unfortunate, but I hope my efforts were enough.

Overall JUnit Test Results



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CPTS422Project	93.3 %	8,755	629	9,384
src/test/java	90.2 %	5,800	629	6,429
My422Project.BlackBoxTestCases	0.0 %	0	626	626
LocalMethodReferenceTest.java	0.0 %	0	92	92
NumOfLoopsTest.java	0.0 %	0	85	85
NumOfExternalMethodReferencesTest.ja	0.0 %	0	70	70
HalsteadLengthTest.java	0.0 %	0	67	67
NumOfExpressionsTest.java	0.0 %	0	59	59
NumOfOperatorsTest.java	0.0 %	0	51	51
NumOfVariableOfDeclarationsTest.java	0.0 %	0	50	50
HalsteadDifficultyTest.java	0.0 %	0	35	35
HalsteadVocabularyTest.java	0.0 %	0	31	31
HalsteadEffortTest.java	0.0 %	0	28	28
NumOfCommentsTest.java	0.0 %	0	23	23
HalsteadVolumeTest.java	0.0 %	0	15	15
NumOfLinesOfCommentsTest.java	0.0 %	0	13	13
ExternalMethodTest.java	0.0 %	0	7	7
My422Project	99.9 %	4,687	3	4,690
HalsteadEffortCheckTest.java	99.5 %	638	3	641
HalsteadDifficultyCheckTest.java	100.0 %	348	0	348
HalsteadLengthCheckTest.java	100.0 %	483	0	483
HalsteadVocabularyCheckTest.java	100.0 %	374	0	374
HalsteadVolumeCheckTest.java	100.0 %	347	0	347
MethodLimitCheckTest.java	100.0 %	134	0	134
NumOfCommentsCheckTest.java	100.0 %	178	0	178
NumOfExpressionsCheckTest.java	100.0 %	267	0	267
NumOfExternalMethodReferencesChecl	100.0 %	428	0	428
NumOfLinesOfCommentsCheckTest.jav	100.0 %	263	0	263
NumOfLocalMethodReferencesCheckTe	100.0 %	483	0	483
NumOfLoopingStatementsCheckTest.ja	100.0 %	137	0	137
NumOfOperandsCheckTest.java	100.0 %	208	0	208
NumOfOperatorsCheckTest.java	100.0 %	205	0	205
NumOfVariableDeclarationsCheckTest.ja	100.0 %	194	0	194
My422Project.BlackBoxTests	100.0 %	1,113	0	1,113
HalsteadDifficultyBlackBoxTest.java	100.0 %	128	0	128
HalsteadEffortBlackBoxTest.java	100.0 %	80	0	80
HalsteadLengthBlackBoxTest.java	100.0 %	92	0	92
HalsteadVocabularyBlackBoxTest.java	100.0 %	91	0	91
HalsteadVolumeBlackBoxTest.java	100.0 %	91	0	91
NumOfCommentsBlackBoxTest.java	100.0 %	79	0	79
NumOfExpressionsBlackBoxTest.java	100.0 %	79	0	79
NumOfExteralMethodReferencesBlackB	100.0 %	79	0	79
NumOfLinesOfCommentsBlackBoxTest.	100.0 %	79	0	79
NumOfLocalMethodReferencesBlackBo	100.0 %	79	0	79
NumOfLoopingStatementsBlackBoxTest	100.0 %	78	0	78
NumOfOperatorsBlackBoxTest.java	100.0 %	79	0	79
NumOfVariableDeclarationsBlackBoxTes	100.0 %	79	0	79
src/main/java	100.0 %	2,955	0	2,955
My422Project	100.0 %	2,955	0	2,955
HalsteadDifficultyCheck.java	100.0 %	408	0	408
HalsteadEffortCheck.java	100.0 %	509	0	509
HalsteadLengthCheck.java	100.0 %	415	0	415
HalsteadVocabularyCheck.java	100.0 %	375	0	375
HalsteadVolumeCheck.java	100.0 %	384	0	384
MethodLimitCheck.java	100.0 %	61	0	61
NumOfCommentsCheck.java	100.0 %	67	0	67
NumOfExpressionsCheck.java	100.0 %	125	0	125
NumOfExternalMethodReferencesChecl	100.0 %	107	0	107
NumOfLinesOfCommentsCheck.java	100.0 %	75	0	75
NumOfLocalMethodReferencesCheck.ja	100.0 %	105	0	105
NumOfLoopingStatementsCheck.java	100.0 %	49	0	49
NumOfOperandsCheck.java	100.0 %	93	0	93
NumOfOperatorsCheck.java	100.0 %	137	0	137
NumOfVariableDeclarationsCheck.java	100.0 %	45	0	45

Discussion on “Class Testing”

Since this project isn’t meant to create a usable product and is mainly focused on testing, there’s very little interaction between different parts of the software. Each fault model metric is tested on its own, which makes this a great situation for class testing. Because there aren’t many parts that need to work together, class testing keeps things simple.

Using class testing here can make the project cleaner and easier to manage. There would be fewer files and less code, so it’s easier to focus on what each part is supposed to do. Since you’re testing each metric separately, you don’t need to worry about setting up complex interactions between different parts, which saves time and effort.

This approach also matches the project’s purpose, which is all about making sure each metric works correctly. By testing each one on its own, you can be sure it’s doing what it’s supposed to without interference from other parts of the system. It also makes debugging simpler since you can easily see where a problem is coming from.

Overall, because this project has very little interaction between parts, class testing is a good fit. It keeps the project straightforward, helps avoid unnecessary complexity, and makes it easier to focus on testing each part effectively.