Cpt\_S 422 Report for Deliverable 3

This report will cover the tests used in this project. There are a lot of similarities with the previous deliverable so some of the text will be similar. This report will also cover the fault model for blackbox testing for each class that can be applied to it. Each check has a blackbox test related to it so there should be multiple tests for blackbox testing which should be all covered.

**DISCLAIMER REGARDING CODE COVERAGE! PLEASE READ BEFORE MOVING ON:**

src/main/java/blackBoxTests shows 0% code coverage. This should be the case because the code in this package only exists for blackbox testing purposes from src/test/java/blackBoxTests and it does not need to be covered.

Why this happens:

The "check" variable being made in each of these classes is using a check implemented within src/main/java/halsteadPackage OR src/main/java/numComPackage. This means that the checks are going through the code in src/main/java/blackboxTests and not the junit tests themselves. Hence why the code coverage is at 0%. Because of this reason, I’m excluding src/main/java/blackBoxTests from this report. Ignoring those files means that I 100% line coverage.

Simply put, src/main/java/blackBoxTests is the only package of classes that seems to be not covered. This disclaimer is to explain that they are indeed covered, just not by junit or any testing peripheral. All the other packages are covered. Which should lead to 100% line coverage.

**Halstead:**

Each test on each class has tested every single method within the class. Each one utilizes mock and spies to test functionality of each method even if there’s no information passed into the variables. This is an example of white box testing since we don’t know the full functionality of the Halstead checks in real time.

Classes tested:

* Length
* Vocabulary
* Volume
* Difficulty
* Effort

Graphical user interface, text

Description automatically generated

Pitclipse Result:

Timeline

Description automatically generated with medium confidence

**Others:**

Each test for each of the classes within this category covers every function/method. This makes for 100% coverage. The tests here utilize mock and spy because we are using a testing method known as white box where we don’t know the full functionality of the code when we run it beforehand.

Classes tested:

* NumOperators
* NumOperands
* NumExpression
* NumLoops
* NumComments
* NumLineComments

There is a screenshot on the next page covering all the methods that were tested.

Graphical user interface, text

Description automatically generated  
**Pitclipse:**

**Timeline

Description automatically generated with medium confidence**

Fault Model

Here is where we will talk about the blackbox tests used for this project. However before the tests are implemented, there needs to be clear examples of how the classes in each package can fail. So this is where the fault model will be introduced. The fault model displays how each class can have a possible compromise. This way blackbox testing can have a more clear direction as to what to test and where do they need to be tested. The fault model will be modeled as such:

Name of category:

* Class/Check name
  + Possible faults/errors

The fault model will be provided on the next page.

Halstead:

* HalsteadLength
  + No length to calculate = case for when Halstead is not used
  + Longer length = case for when Halstead is used extensively
* HalsteadDifficulty
  + No difficulty = case for when Halstead is not used
  + High Difficulty = case for when Halstead is used extensively
* HalsteadEffort
  + No effort = case for when Halstead is not used
  + High effort = case for when Halstead is used extensively
* HalsteadVocabulary
  + No vocabulary = case for when Halstead is not used
  + High vocabulary = case for when Halstead is used extensively
* HalsteadVolume
  + No volume = case for when Halstead is not used
  + Type mismatch = int to double mismatch because volume calculation requires division
  + Divide by 0 = Self explanatory

Others

* NumOperators
  + No operators = case where there are no operators within the code
  + Commented operators = case where the operator is commented
  + Every Operator = case where every operator exists in an instance of a class
* NumOperands
  + No operands = case where there are no operands
  + Commented operands = case where the operand is commented
  + Every Operand = case where every operand exists in an instance of a class
* NumExpression
  + No expressions = case where there are no expressions in the code
  + Every expression = case where every expression exists in an instance of a class
* NumLoops
  + No loops = case where there are no loops within a class
  + Nested loops = case where there is a nested loop within a class
  + Multiple types of loops = case where every type of loop exists within a class
* NumComments
  + No comments = case where there are no comments within a class
  + Block comment = case where there is a block comment within a class
  + Adjacent comments = case where there are 2 line comments right next to each other
  + Summary comments = case for a summary comment instance
* NumLineComments
  + No comments = case for when there are no comments within a class
  + Block comment = case for when there is a block comment within a class
  + Adjacent comments = case for when there are 2 line comments next to each other
  + Summary comments = case for a summary block within a class

Fault model continued…

As seen in the example on the previous page, the tests in each area mainly touch upon border cases within the project. Borders such as “no \_\_\_\_” or Every \_\_\_\_\_\_”. For example, we need to test instances of null or noComments. This way whenever there are no comments within a class, the test will account for that. Testing border cases is a great way to touch upon most blackbox scenarios because border cases are often the most common cases that cause a fault in the code. The same can be said for testing every outcome. This means that we should leave no stone unturned. If the number of operators is being tested we would want to make sure that every operator is taken into account and we shouldn’t miss a single operator. The same can be said for the other tests.

Specific types of tests can also be proven useful. For example, having the NumLineComments test for a block comment would be useful because we would want to include block comments within our tests as they are counted differently than a normal comment.

Test Engine

There is a test engine package and class that was created in src/test/java/testEngine directory. This engine is used to run black box tests for the checks. Each check is passed into the testEngine and along with the check, an example of the code that the check should be checking would be tied along with the check. That example code is an instance of a class that would have code provided (but is never called in the main project so it’s never truly utilized) and observed by the check powered by the testEngine. The check will go through the example code’s scope to make sure that the check is working properly.

Pitclipse

Pitclipse is used within our tests alongside Junit. This piece of software gives the user information about test mutations that will warn the user of test faults that might occur later in development. Junit does not provide the user with this kind of information. According to the information provided in the test cases on pages 2 (halstead), and 3 (others), I have 100% line coverage but I’m close to covering all the mutations. With roughly 10 mutations missed in the code. This means that my code may be vulnerable to errors or faults later in the project’s lifespan or further development. The strength of each test reflects a similar stance as some tests might need to be strengthened to cover all the bases in the future.