422 REPORT
Julie R.

This report is a brief overview of Deliverable 3. In this phase, I have developed three new packages: BlackBoxTesting, BlackBoxTestSourceCode, and TestEngine. The BlackBoxTestSourceCode package encompasses a set of straightforward Java programs specifically designed for testing. These programs are evaluated using the BlackBoxTesting suite, which is integrated with the TestEngine.

The TestEngine plays a crucial role in automating the testing process. This component of the system is adeptly adapted from the TreeWalker, enhancing its functionality to suit our specific testing requirements. The primary focus of the BlackBoxTesting package is to execute these tests effectively, ensuring comprehensive coverage and efficiency.
This structured approach not only streamlines the testing process but also ensures a high level of accuracy and reliability in our results.

Here are the potential fault models taken into consideration before developing my black box tests.

<u>Fault models</u>

1. Halstead Length

- Miscount of Operators and Operands: The check might incorrectly count the number of operators and operands, especially in complex expressions or nested structures.
- Overlooking Certain Types: Certain types of operators or operands (like unary operators, assignment operators, or literals) might be overlooked or misclassified.

2. Halstead Vocabulary

- Incorrect Distinction Between Unique and Total Elements: The check could fail to correctly differentiate between unique and total operators and operands.
- Handling of Similar but Different Operators: Operators that are similar in appearance but different in functionality (like == and =) might be incorrectly classified as the same.

3. Halstead Volume

- Calculation Errors: Since volume is a function of length and vocabulary, errors in calculating either of these could lead to incorrect volume calculations.
- Handling of Large Numbers: The check might not accurately calculate the volume in scenarios with a large number of operators and operands.

4. Halstead Difficulty

- Ratio Calculation Errors: Difficulty is based on the ratio of different operator and operand types. Misidentification or miscounting of these elements can lead to incorrect difficulty values.
- Handling of Zero Operands: Special cases, such as methods with no operands, might lead to division by zero or other computational errors.

5. Halstead Effort

- Compound Errors from Other Metrics: Since effort is derived from volume and difficulty, any errors in these metrics can compound in the effort calculation.
- Extreme Values Handling: The check might fail to handle extremely high or low effort values, which could occur in large or complex codebases.

<u>Fault Models for Operands and Operators</u>

Misclassification of Operators and Operands
- Fault: The tool incorrectly classifies operators as operands and vice versa, particularly in complex or unconventional code constructs.
- Example: Misinterpreting a function pointer or a lambda expression as an operand rather than an operator in languages like C++ or JavaScript.

Handling of Overloaded Operators
- Fault: Inability to correctly handle operator overloading, where the same operator symbol performs different operations based on context.
- Example: In C++, the << operator could be a bitwise shift or a stream insertion operator, depending on the context.

Operator Precedence and Associativity
- Fault: Incorrect understanding of operator precedence and associativity rules, leading to wrong interpretations.
- Example: Misinterpreting the order of execution in expressions like a + b * c.

<u>Fault models for looping:</u>

Off-by-One Errors: Perhaps the most common loop fault, this occurs when the loop iterates one time too many or one time too few. This often happens due to incorrect initialization or termination conditions, such as using <= instead of <.
Infinite Loops: This happens when the loop's termination condition is never met, causing the loop to execute indefinitely. Infinite loops can occur due to an improper update of the loop variable or a faulty conditional expression.
Uninitialized Variables: Using uninitialized variables inside a loop can lead to unpredictable behavior. This may happen when loop variables are not properly initialized before the loop begins.
Incorrect Loop Bounds: Setting incorrect boundaries for the loop, such as using the wrong variable for the loop's upper or lower limit, can lead to either missed iterations or excess iterations.

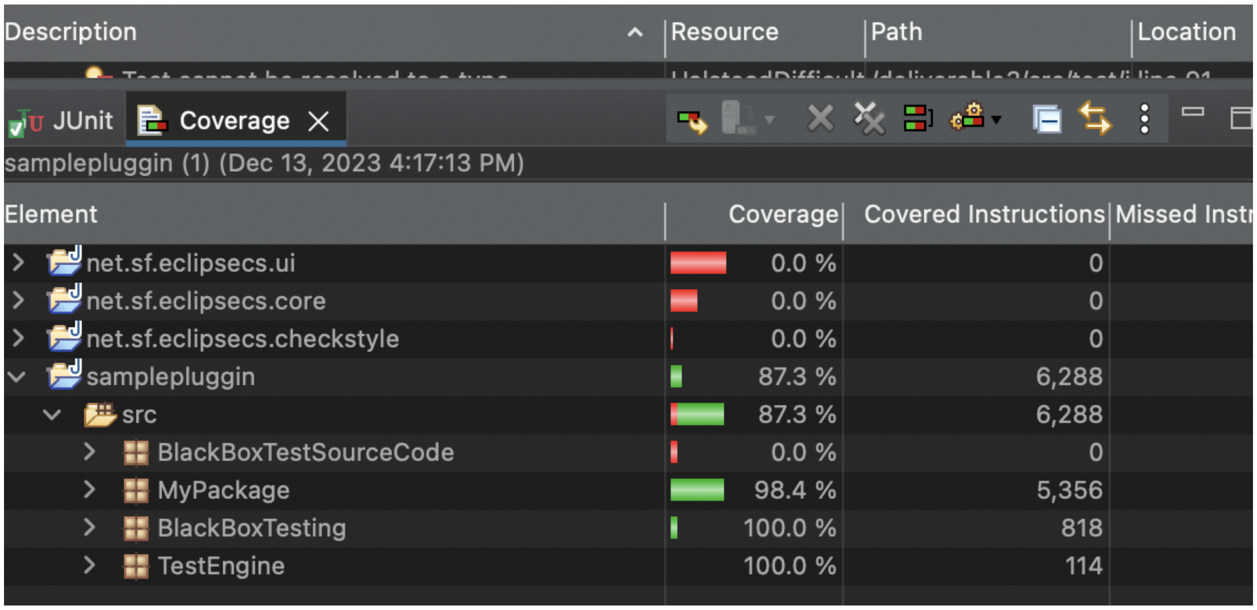
Improper Loop Variable Modification: Modifying the loop control variable within the loop body in an unintended way can disrupt the normal flow of the loop.
Nested Loop Errors: In nested loops, errors might arise from incorrect usage of loop variables, especially if the inner loop mistakenly modifies a variable that is controlling the outer loop.
Break and Continue Misuse: Incorrect use of break or continue statements can lead to premature termination of the loop or skipping necessary iterations.
Logical Errors in Loop Conditions: Using the wrong logical operators or conditions in the loop can lead to incorrect execution and unexpected results.
Concurrent Modification Error: In concurrent or multi-threaded environments, modifying a data structure while iterating over it can lead to concurrent modification errors.
Dead Code: Sometimes, due to certain conditions in loops, some code sections may never execute (dead code), which might not be an intended behavior.

General Fault Models

- Language-Specific Constructs: Certain programming constructs specific to Java (or any particular language) might not be accurately analyzed.
- Comment and String Literal Handling: The presence of operators or operands within comments or string literals should be ignored, but they might be erroneously counted.
- Empty or Minimal Code Blocks: Methods or classes with minimal code (or none at all) might cause errors in the calculation.

After going through the fault models I started to develop my blackbox source code and my tests. These black box tests uncovered some bugs within MyPackage. Since white box testing utilized the source code of how the check was made, I knew what to expect with the assert statements. With blackbox testing on my BlackBoxTestSourceCode, I didn't have access so I counted by hand and realized I was using the wrong approach to Halstead Volume as well as Halstead Effort. Since I found this out, I adapted my check to use the correct calculation and redid the blackbox checks as well.

Here is the final results of my tests and my coverage:



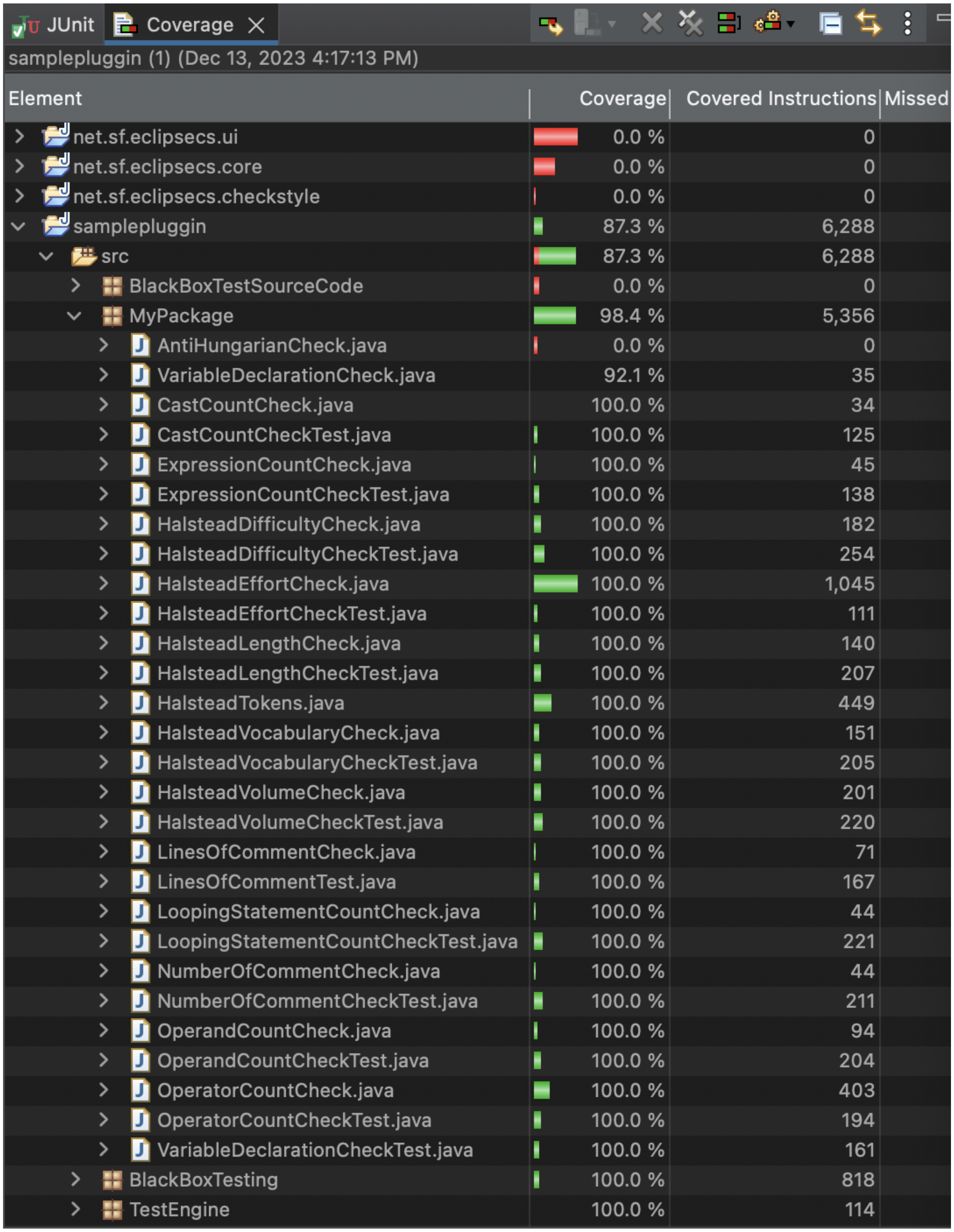
Description	Resource	Path	Location

JUnit | Coverage

samplepluggin (1) (Dec 13, 2023 4:17:13 PM)

Element	Coverage	Covered Instructions	Missed Instr
net.sf.eclipsecs.ui	0.0 %	0	
net.sf.eclipsecs.core	0.0 %	0	
net.sf.eclipsecs.checkstyle	0.0 %	0	
samplepluggin	87.3 %	6,288	
src	87.3 %	6,288	
BlackBoxTestSourceCode	0.0 %	0	
MyPackage	98.4 %	5,356	
BlackBoxTesting	100.0 %	818	
TestEngine	100.0 %	114	

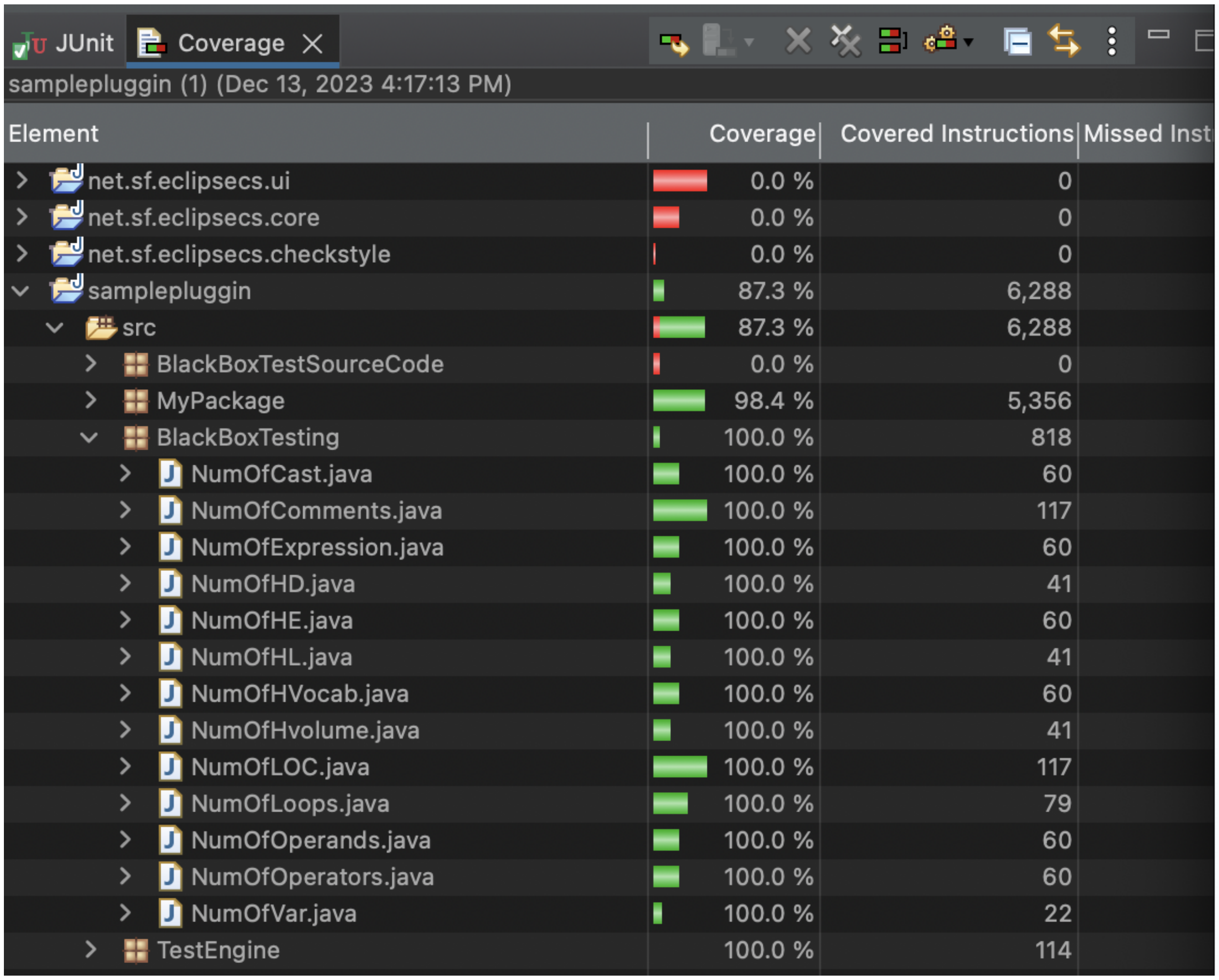
This shows that the total project has a 87% coverage.



samplepluggin (1) (Dec 13, 2023 4:17:13 PM)

Element	Coverage	Covered Instructions	Missed
net.sf.eclipsecs.ui	0.0 %	0	
net.sf.eclipsecs.core	0.0 %	0	
net.sf.eclipsecs.checkstyle	0.0 %	0	
samplepluggin	87.3 %	6,288	
src	87.3 %	6,288	
BlackBoxTestSourceCode	0.0 %	0	
MyPackage	98.4 %	5,356	
AntiHungarianCheck.java	0.0 %	0	
VariableDeclarationCheck.java	92.1 %	35	
CastCountCheck.java	100.0 %	34	
CastCountCheckTest.java	100.0 %	125	
ExpressionCountCheck.java	100.0 %	45	
ExpressionCountCheckTest.java	100.0 %	138	
HalsteadDifficultyCheck.java	100.0 %	182	
HalsteadDifficultyCheckTest.java	100.0 %	254	
HalsteadEffortCheck.java	100.0 %	1,045	
HalsteadEffortCheckTest.java	100.0 %	111	
HalsteadLengthCheck.java	100.0 %	140	
HalsteadLengthCheckTest.java	100.0 %	207	
HalsteadTokens.java	100.0 %	449	
HalsteadVocabularyCheck.java	100.0 %	151	
HalsteadVocabularyCheckTest.java	100.0 %	205	
HalsteadVolumeCheck.java	100.0 %	201	
HalsteadVolumeCheckTest.java	100.0 %	220	
LinesOfCommentCheck.java	100.0 %	71	
LinesOfCommentTest.java	100.0 %	167	
LoopingStatementCountCheck.java	100.0 %	44	
LoopingStatementCountCheckTest.java	100.0 %	221	
NumberOfCommentCheck.java	100.0 %	44	
NumberOfCommentCheckTest.java	100.0 %	211	
OperandCountCheck.java	100.0 %	94	
OperandCountCheckTest.java	100.0 %	204	
OperatorCountCheck.java	100.0 %	403	
OperatorCountCheckTest.java	100.0 %	194	
VariableDeclarationCheckTest.java	100.0 %	161	
BlackBoxTesting	100.0 %	818	
TestEngine	100.0 %	114	

This shows coverage for my checks - as I ran unit test against every function in that check.



samplepluggin (1) (Dec 13, 2023 4:17:13 PM)

Element	Coverage	Covered Instructions	Missed Inst
net.sf.eclipsecs.ui	0.0 %	0	
net.sf.eclipsecs.core	0.0 %	0	
net.sf.eclipsecs.checkstyle	0.0 %	0	
samplepluggin	87.3 %	6,288	
src	87.3 %	6,288	
BlackBoxTestSourceCode	0.0 %	0	
MyPackage	98.4 %	5,356	
BlackBoxTesting	100.0 %	818	
NumOfCast.java	100.0 %	60	
NumOfComments.java	100.0 %	117	
NumOfExpression.java	100.0 %	60	
NumOfHD.java	100.0 %	41	
NumOfHE.java	100.0 %	60	
NumOfHL.java	100.0 %	41	
NumOfHVocab.java	100.0 %	60	
NumOfHvolume.java	100.0 %	41	
NumOfLOC.java	100.0 %	117	
NumOfLoops.java	100.0 %	79	
NumOfOperands.java	100.0 %	60	
NumOfOperators.java	100.0 %	60	
NumOfVar.java	100.0 %	22	
TestEngine	100.0 %	114	

This is the coverage results of my tests being run against my blackboxTestSourceCode. BlackboxTesting is the package that utilized the code and to attain full coverage, I individually wrote every check code and wrote a test for it.