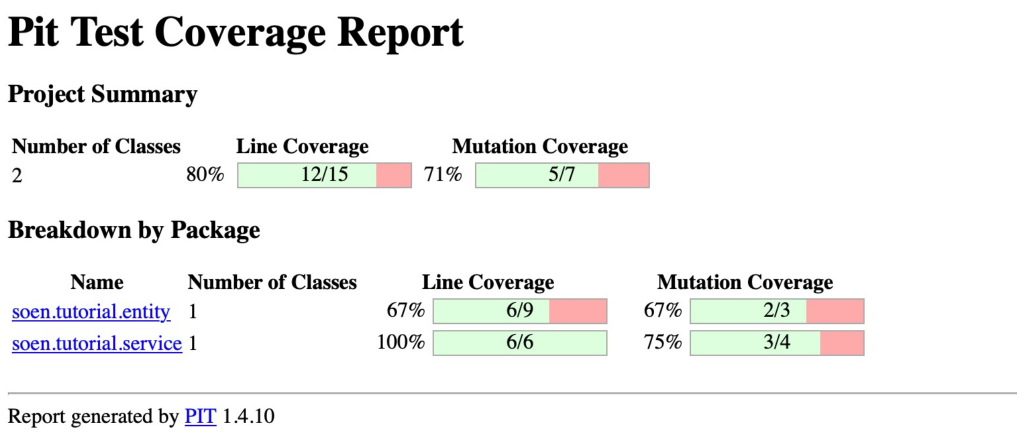
SOEN 345 - Software Testing, Verification, and Quality Assurance
Assignment 2

Ali Hanni - 40157164

Gina Cody School of Computer Science and Software Engineering
Concordia University, Montreal, QC, Canada

Winter 2022

Question 1



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
2	80%	12/15	71%	5/7

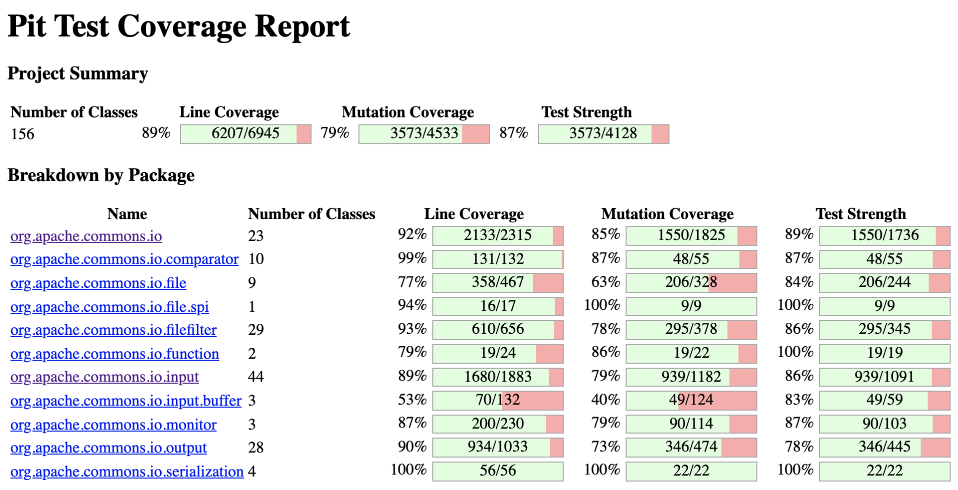
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
soen.tutorial.entity	1	67%	6/9	67%	2/3
soen.tutorial.service	1	100%	6/6	75%	3/4

Report generated by PIT 1.4.10

The line coverage for the Mutation tutorial project is of 80% while the mutation coverage is at 71%.

Question 2



Pit Test Coverage Report

Project Summary

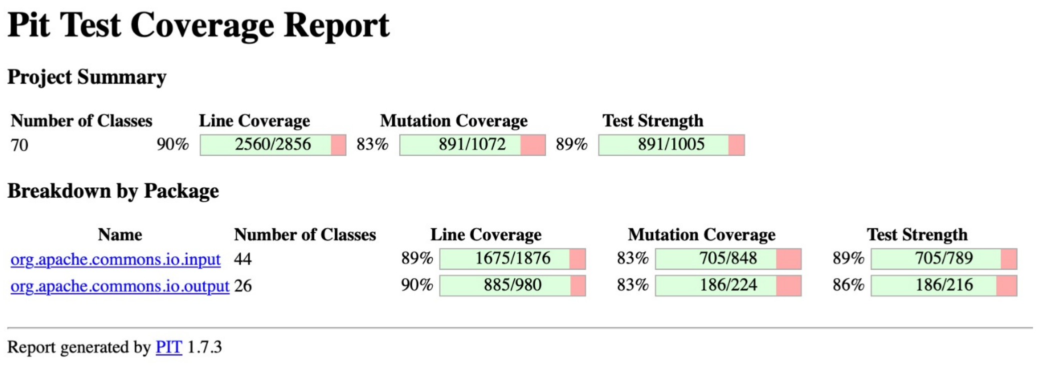
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
156	89%	6207/6945	79%	3573/4533	87%	3573/4128

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.io	23	92%	2133/2315	85%	1550/1825	89%	1550/1736
org.apache.commons.io.comparator	10	99%	131/132	87%	48/55	87%	48/55
org.apache.commons.io.file	9	77%	358/467	63%	206/328	84%	206/244
org.apache.commons.io.file.spi	1	94%	16/17	100%	9/9	100%	9/9
org.apache.commons.io.filefilter	29	93%	610/656	78%	295/378	86%	295/345
org.apache.commons.io.function	2	79%	19/24	86%	19/22	100%	19/19
org.apache.commons.io.input	44	89%	1680/1883	79%	939/1182	86%	939/1091
org.apache.commons.io.input.buffer	3	53%	70/132	40%	49/124	83%	49/59
org.apache.commons.io.monitor	3	87%	200/230	79%	90/114	87%	90/103
org.apache.commons.io.output	28	90%	934/1033	73%	346/474	78%	346/445
org.apache.commons.io.serialization	4	100%	56/56	100%	22/22	100%	22/22

The line coverage for the *apache-commons-io* project is at 89% while the mutation coverage is at 79%. Test strength is evaluated at 87%.

Question 3



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
70	90% 2560/2856	83% 891/1072	89% 891/1005

Breakdown by Package

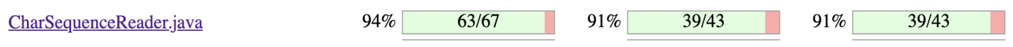
Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.apache.commons.io.input	44	89% 1675/1876	83% 705/848	89% 705/789
org.apache.commons.io.output	26	90% 885/980	83% 186/224	86% 186/216

Report generated by PIT 1.7.3

After modifying the *pom.xml* file for *apache-commons-io* project so it only tests *input* and *output* packages and kept only the conditional boundary, negate conditional, and return values mutators, the line coverage was at 90%, the mutation coverage at 83% and test strength at 89%. The modified *pom.xml* file is joined with this submission as *q3.xml*.

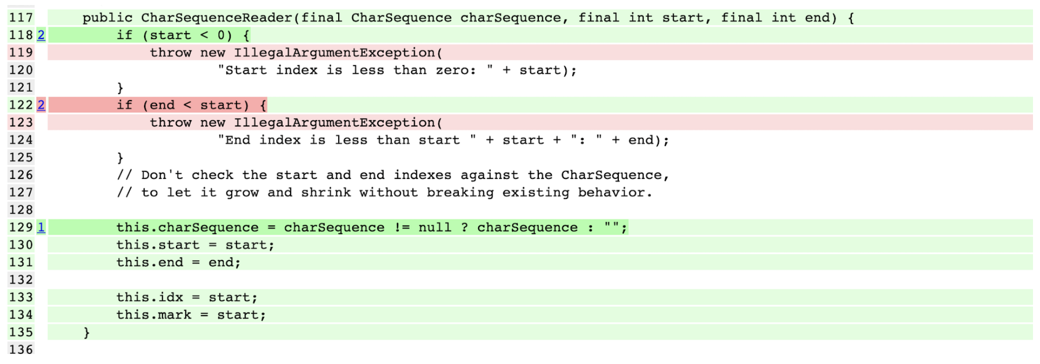
Question 4

After running the tests on the selected packages with the selected mutations activated, we search for a class that has a lower mutation coverage. Since I was looking for mutants, I also searched for a class that has a high line coverage so I know that the mutants that survived were not simply not covered in order to learn more about mutants and how they work.
I found the class `CharSequenceReader.java`



CharSequenceReader.java	94% 63/67	91% 39/43	91% 39/43

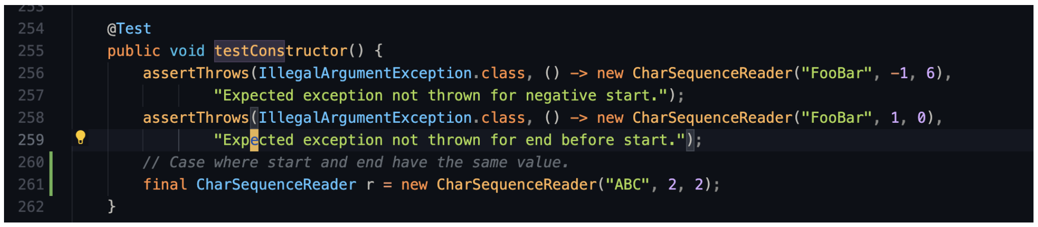
Then I searched the class looking for some mutants that were not killed.



```
    public CharSequenceReader(final CharSequence charSequence, final int start, final int end) {
        if (start < 0) {
            throw new IllegalArgumentException(
                    "Start index is less than zero: " + start);
        }
        if (end < start) {
            throw new IllegalArgumentException(
                    "End index is less than start " + start + ": " + end);
        }
        // Don't check the start and end indexes against the CharSequence,
        // to let it grow and shrink without breaking existing behavior.

        this.charSequence = charSequence != null ? charSequence : "";
        this.start = start;
        this.end = end;

        this.idx = start;
        this.mark = start;
    }

```

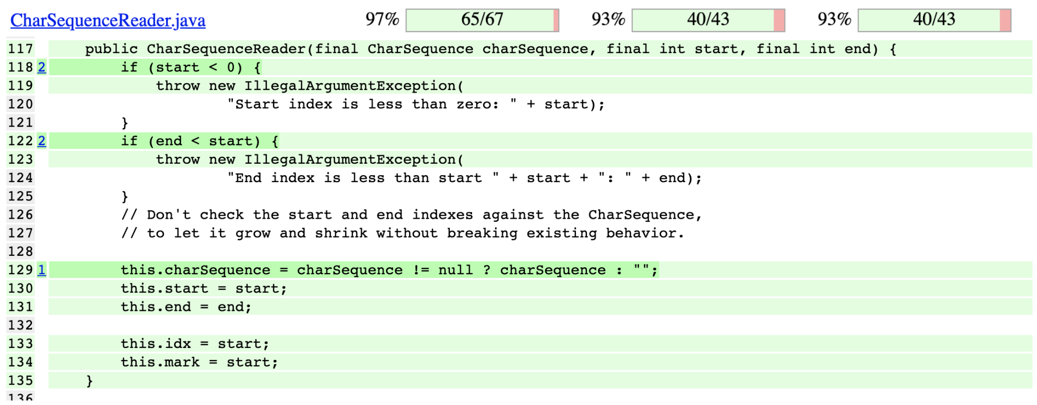
This `CharSequenceReader` constructor has a series of if statement to verify if the *start* and *end* variables passed to it that specify respectively the start and end index in the character sequence to be read are logic. The first verifies if the start is not an

negative integer. The second if is meant to verify that the end index is not smaller than the start.
The second if statement had a changed conditional boundary mutant that survived. Indeed, after taking a look at the tests that can be found in the `CharSequenceReaderTest.java` file, I realized that no test handled the case where the start and end have the same value. I added a constructor and provided a start and end value of 2:



```java
    @Test
    public void testConstructor() {
        assertThrows(IllegalArgumentException.class, () -> new CharSequenceReader("FooBar", -1, 6),
                "Expected exception not thrown for negative start.");
        assertThrows(IllegalArgumentException.class, () -> new CharSequenceReader("FooBar", 1, 0),
                "Expected exception not thrown for end before start.");
        // Case where start and end have the same value.
        final CharSequenceReader r = new CharSequenceReader("ABC", 2, 2);
    }
```

After this addition, the result of the changed conditional boundary mutant would be different that that of the normal code. Indeed, while `2 < 2` will return false, `2 <= 2` will return true. The mutant was therefor killed and the mutation coverage and test strength meliorated.



CharSequenceReader.java 97% 65/67 93% 40/43 93% 40/43

```java
    public CharSequenceReader(final CharSequence charSequence, final int start, final int end) {
        if (start < 0) {
            throw new IllegalArgumentException(
                    "Start index is less than zero: " + start);
        }
        if (end < start) {
            throw new IllegalArgumentException(
                    "End index is less than start " + start + ": " + end);
        }
        // Don't check the start and end indexes against the CharSequence,
        // to let it grow and shrink without breaking existing behavior.

        this.charSequence = charSequence != null ? charSequence : "";
        this.start = start;
        this.end = end;

        this.idx = start;
        this.mark = start;
    }
```

Question 5

1. The packages with lower test strength are `org.apache.commons.io.output` with 78% test strength and 90% line coverage and `org.apache.commons.io.input.buffer` with 83% test strength and 53% line coverage. The packages with highest test strength are `org.apache.commons.io.file.spi` with 100% test strength and 94% line coverage, `org.apache.commons.io.function` with 100% test strength and 79% line coverage, and `org.apache.commons.io.file.serialization` with 100% test strength and 100% line coverage. However, these 3 packages are the one with least number of classes and least number of line of code.
   I realized first that there is no concrete correlation between the line coverage and the test strength, quit the contrary, while overall, the tests that has the highest line coverage tend to have a lower test strength. Also, I realized that the classes with best test strength were short compared to the others. It can therefor be a good idea to keep the classes small and separate concerns and functions as much as possible in order to keep the code simple and easier to test. Here are three concrete ideas to improve test

   1 - Add more tests for 'risky' sections of the project with low test strength in order to improve the line coverage, this will make it easier to spot surviving mutants and upgrade the defaulting tests.

   2 - Focus on certain type of mutants that tend to survive more than others in order to maybe detect a flaw in the way tests are written. It can help produce a method to quickly determine what can be improved for any given test.

   3 - Produce statistics for each package, that give insight on what is the type of mutants that tend to survive the most, and go to the test code to try and understand why it is the case. From that quick analysis, interesting general rules can be derived and help upgrade the testing suite.

2. a. As previously stated, there is no solid correlation between line coverage and test strength. A low line coverage means that many tests and consequently many mutants will not be covered. The test strength only takes into account the cases where the mutants are covered. However, all mutants found in tests that are not covered are considered 'failures', as if the test was undertaken but the mutants survived. Contrary to test strength, mutation coverage takes into account tests that are not covered. Because of this, we can see a clear correlation between line coverage and mutation coverage. The lower the line coverage, the lower the mutation coverage.

   b. The package with lower test strength is the `output` package. A script was used to compute this data. A total number of 777 mutants survived in that packages. The mutator type with the highest surviving mutants ratio is the `ConditionalsBoundaryMutator` with 47% of the mutants surviving. The mutator type with the highest surviving mutants number is the `VoidMethodCallMutators` with 144 surviving mutants. A total of 262 mutants survived.

c. Adding more test can help improving line coverage, which in turns will possibly give more insight on test suite quality and reason (type of surviving mutators) why test strength is low. However, simply adding more tests does not guarantee an improved test suite. The added tests should take into account the surviving mutants in order to be pertinent and have a positive impact on the quality of the test suite.

d. Running the whole test suite every time one wants to verify the efficiency of an added or modified test test can be very time consuming. Even using surefire to reduce the scope of the testing to only the package of interest still takes a lot of time to run. However, deleting the other tests and adding them back after processing is done greatly reduces the running time and help kill all mutants and hence build a better testing suite faster.

e. Yes. Mutation testing helps to find the test cases the programmer has not thought of. It can be very helpful especially in a situation where the code to test is long, complex and handles many functionalities.