Contents

Programming Language & Development Tools

We used Java programming language version 11.0.6 for this project.

We used Maven for our build tools and we used JUnit 5 for implementing our unit tests.

GitLab Repository

The source codes have been uploaded to the GitLab and you can access it through the following link:

https://gitlab.com/sbabakmehrabi/mercel-task

We thought that you must give the task description to other candidates as well. So, maybe it would not be a wise decision to make this repository public.

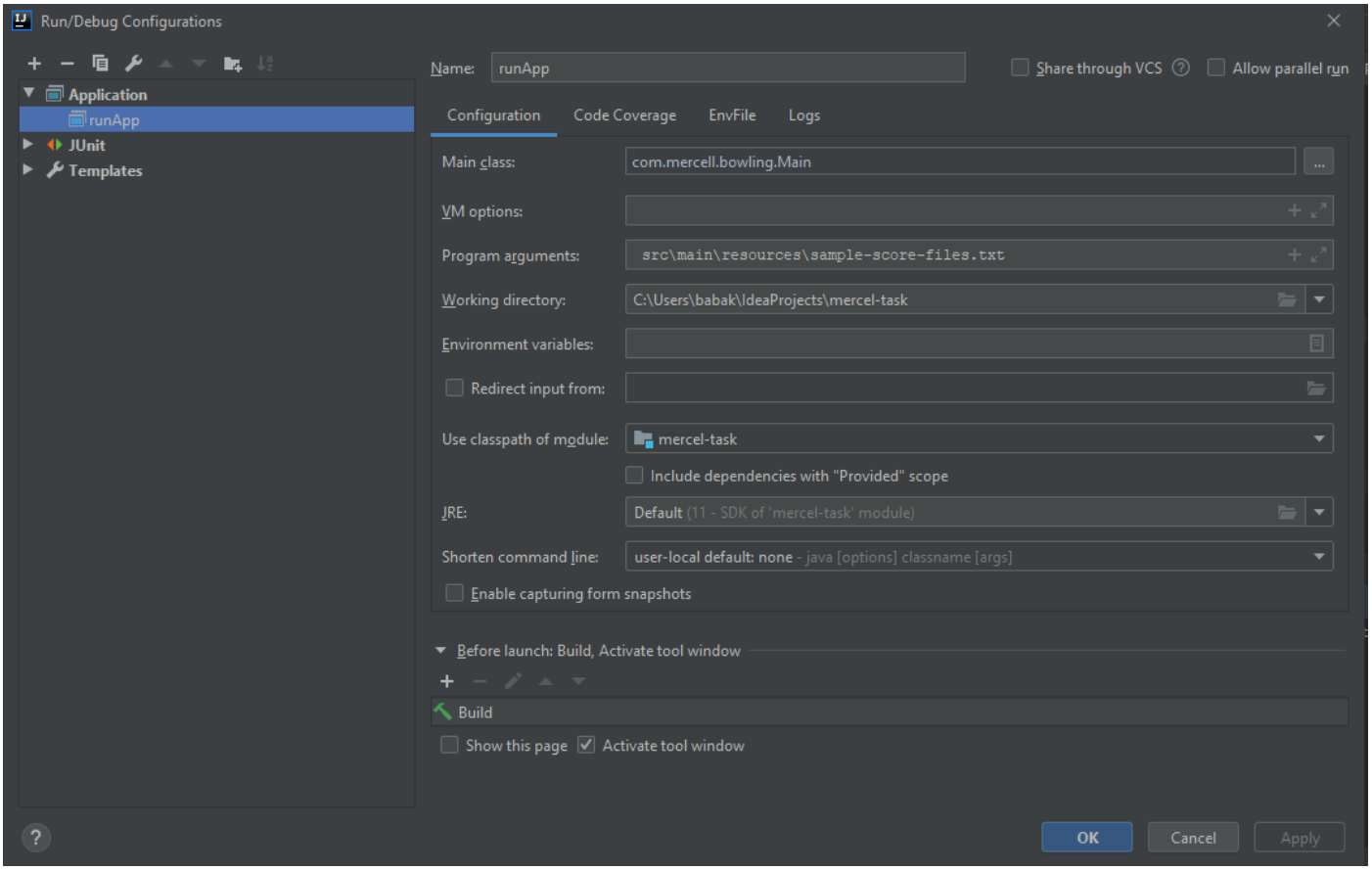
Of course, you can have access to this repository by providing us your GitLab account username.

Running the Project inside IntellijIDEA

We used IntellijIDEA as the editor for writing this program. So, you can import the source code to the IntellijIDEA.

This is a maven project. So, you will see a `pom.xml` file that shows the project properties and its dependencies. Open the pom.xml file.

The IntelliJ IDEA must show you a prompt that asks you for the installation of dependencies. Accept it and after that, you must be able to run the project with the following run configuration:



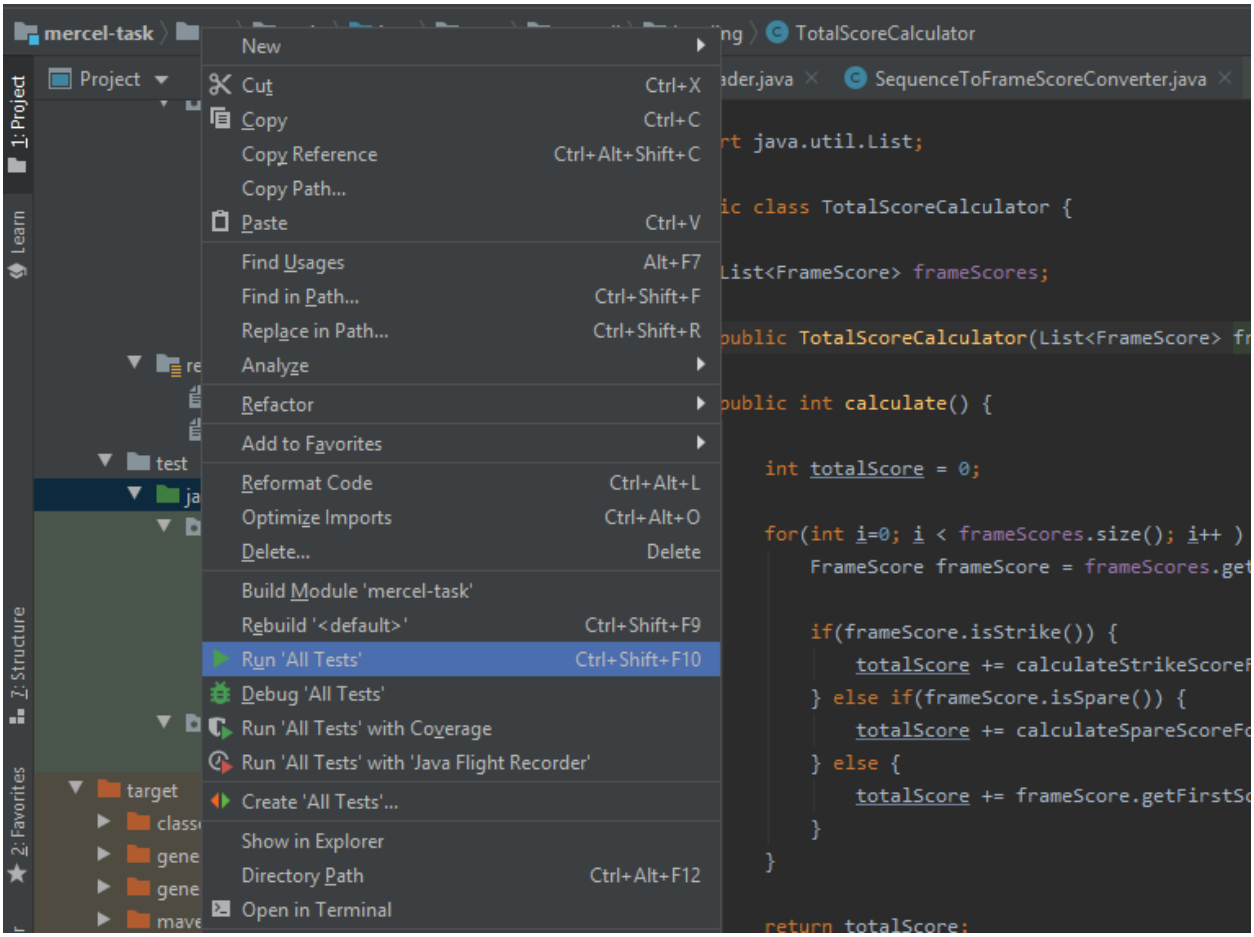
Remember that this project expects to get the file path to the game scores as the argument. So, in this run configuration, we passed the following path to it.

src\main\resources\sample-score-files.txt

This file was added to the source code. You can use this file or any desired file.

Running Tests

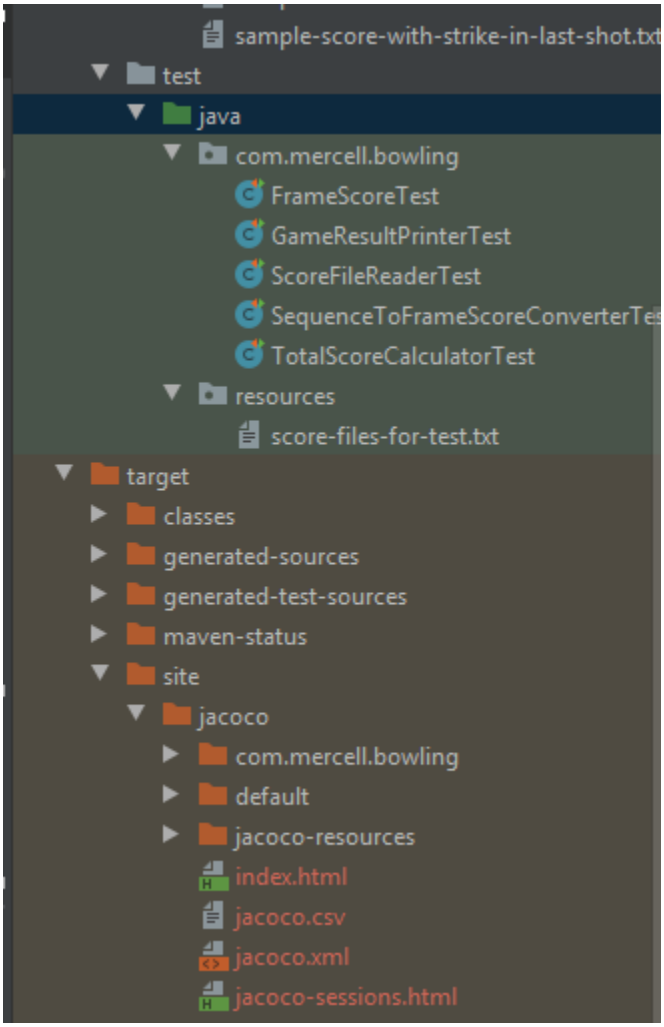
To run tests inside IntellijIDEA, right-click on the test/java folder and select the `Run All Tests` option (As shown in the following picture)



Code Coverage Report

After running tests inside the IntellijIDEA, you must see a newly created folder inside the following path:

\target\site\jacoco\



Inside this folder, you will see an index.html file. Open in with your favorite browser. Now, you can see the code coverage report.

mercel-task

mercel-task

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mercell.bowling		96%		94%	5	59	8	145	2	33	1	7
Total	25 of 703	96%	3 of 52	94%	5	59	8	145	2	33	1	7

mercel-task > com.mercell.bowling

com.mercell.bowling

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main		46%		100%	1	3	7	14	1	2	0	1
BowlingConstants		0%		n/a	1	1	1	1	1	1	1	1
GameResultPrinter		100%		100%	0	14	0	32	0	7	0	1
TotalScoreCalculator		100%		100%	0	11	0	25	0	4	0	1
SequenceToFrameScoreConverter		100%		85%	2	9	0	35	0	2	0	1
FrameScore		100%		87%	1	19	0	31	0	15	0	1
ScoreFileReader		100%		n/a	0	2	0	7	0	2	0	1
Total	25 of 703	96%	3 of 52	94%	5	59	8	145	2	33	1	7

Running the Project in the Command Line

This is a maven project. So, you must have the maven installed and configured in your path. If you have not already installed Maven, you can use the following link to install and prepare it.

https://maven.apache.org/install.html

After installing maven, run the following command inside your terminal

- *mvn install*

This command will install the dependencies of the project. Now, you must build the project by the following command:

- *mvn package*

This command must run all of the tests and combine all of the class files into a single JAR file. Now, you must be able to run the program by the following command:

- *java -cp target\mercel-task-1.0-SNAPSHOT.jar com.mercell.bowling.Main src\main\resources\sample-score-files.txt*

The program expects to get the file path as the first argument. There is a sample file inside our source code. You can use it or you can specify any desired file.

Running Tests

To run tests inside the command line, run the following command:

- *mvn test*

You must be able to see the test result report in the terminal.

Code Coverage Report

After running tests inside the command line, you must see a newly created folder inside the following path:

\target\site\jacoco\

Inside this folder, you will see an index.html file. Open in with your favorite browser. Now, you can see the code coverage report.

mercel-task

mercel-task

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mercell.bowling		96%		94%	5	59	8	145	2	33	1	7
Total	25 of 703	96%	3 of 52	94%	5	59	8	145	2	33	1	7

mercel-task > default > FrameScore

FrameScore

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
setFirstScore(int)		0%		n/a	1	1	2	2	1	1
setSecondScore(int)		0%		n/a	1	1	2	2	1	1
FrameScore(int, int)		100%		n/a	0	1	0	6	0	1
FrameScore(int)		100%		n/a	0	1	0	5	0	1
isStrike()		100%		100%	0	3	0	1	0	1
FrameScore()		100%		n/a	0	1	0	3	0	1
getFirstScore()		100%		n/a	0	1	0	1	0	1
getSecondScore()		100%		n/a	0	1	0	1	0	1
Total	8 of 62	87%	0 of 4	100%	2	10	4	17	2	8

Development Process

We used the Test Driven Development process to implement this task. In this manner, we iterate through the following cycle:

- Write the test
- Write the code
- Refactor

We first start writing a test for the feature we want to implement. In this step, we also consider our boundary situations, null-pointers, empty lists, and other specific scenarios.

Of course, this test will be failed. After that, we implement our feature in a way that it passes all of the tests. At this step, we are sure that our implementation satisfies all of the tests.

At the next step, we work on refactoring our codes. Of course, these changes must also pass all of the tests.

And then, we iterate over these 3 steps for the next feature.

Code Coverage

To develop high-quality software, we must have the mindset of 100 percent code coverages. By this mindset, we know that each line of our code may cause a problem to the final product, and because of that, we need to implement tests for each line (And remember that 100 code coverage does not mean that our software works perfectly on any situation with any data set).

But in practice, we know that there would be no need to bind ourselves to the 100 code coverage. Because there is no need to write tests for auto-generated getters and setters. We try to reach 80-90 percent code coverage for this task.

We use JaCoCO for calculation of code coverage and we add it through our pom.xml file.

First Steps of the Development

In this part, we want to explain a bit about our development process. As we mentioned before, we will use TDD for this task.

As the first entity of the project, we create the class FrameScore. The purpose of this class is to hold the score values for each frame. This class must contain two numbers for storing the score values. Because some turns may have only one score (strike shots), we will use -1 for specifying that there is no score.

Then we create a test for this class called FrameScoreTest. First, we want to test that if we create a FrameScore without any arguments, both values are initialized with -1. So, let us write our first test.

```
class FrameScoreTest {

    private FrameScore frameScore;

    @BeforeEach
    public void setUp() {
        frameScore = new FrameScore();
    }

    @Test
    public void bothScoresForAnEmptyFrameScoreObjectMustBeMinesOne() {
        assertEquals(-1, frameScore.getFirstScore());
        assertEquals(-1, frameScore.getSecondScore());
    }

}
```

Of course, this test will be failed. Even we will face the compilation error. Because we have not yet implemented the gitFristMethod and getSecondScore methods for this class.

So, Let’s start implementing our class.

```
public class FrameScore {
    private int firstScore = -1;
    private int secondScore = -1;

    public FrameScore() {}

    public FrameScore(int firstScore, int secondScore) {
        this.firstScore = firstScore;
        this.secondScore = secondScore;
    }

}
```

As mentions earlier, we used two numbers for storing score values for each shot in each frame. Now, our test will be succeeded.

There are two other important roles for the FrameScore class. We want to identify the Strike and Spare hits.

So, let us add another test to our FrameScoreTest class.

```
@Test
public void isStrikeMustOnlyReturnsTrueIfTheFirstScoreIs10() {
    FrameScore strikeFrameScore = new FrameScore(10);
    assertTrue(strikeFrameScore.isStrike(), "A shot with the first score equals to 10
is a Strike.");

    FrameScore notStrike = new FrameScore(5, 4);
    assertFalse(notStrike.isStrike(), "If the first score is not 10, then this shot
is not Strike.");

}
```

This is our test. In the first scenario, we create a Strike shot and we expect that our method returns true for this shot. In the second scenario, we created a normal shot. Of course, our test must return false for this shot.

Remember that in unit testing, we only test a single functionality. Also, there are two assertions for this test, they are very similar and they are checking the same functionality. So, it is OK for our test. We added a message to our asserts to identify the problem more easily if this test is going to fail.

We have not implemented the isStrike() method yet. So, this test can not be compiled. Let us implement it.

```
public boolean isStrike() {
    return firstScore == 10 && secondScore == -1;
}
```

Now, we can run our tests and we see the green line.

Measuring Code Coverage for this Step

Let us measure our code coverage until this step of development.

We stated that we added the JaCoCo to our pom.xml file. So, when we run our tests with the `mvn test` command, we see that a folder with the name of `site` will be created inside our target folder.

In this folder, there is another folder with the name of `JaCoCo` and inside that, we see an index.html file. Let us open this file inside our browser. We see the result in the below picture:

mercel-task > default

default

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
FrameScore		87%		100%	2	10	4	17	2	8	0	1
Main		0%		n/a	2	2	3	3	2	2	1	1
Total	15 of 69	78%	0 of 4	100%	4	12	7	20	4	10	1	2

We have not implemented our Main class yet. So the 0 value for it is OK. We will work on it further. But we see the 87% coverage for our FrameClass. Let us see the detailed report and how we can improve our code coverage.

The detailed report is shown in the following picture

mercel-task > default > FrameScore

FrameScore

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
setFirstScore(int)		0%		n/a	1	1	2	2	1	1
setSecondScore(int)		0%		n/a	1	1	2	2	1	1
FrameScore(int, int)		100%		n/a	0	1	0	6	0	1
FrameScore(int)		100%		n/a	0	1	0	5	0	1
isStrike()		100%		100%	0	3	0	1	0	1
FrameScore()		100%		n/a	0	1	0	3	0	1
getFirstScore()		100%		n/a	0	1	0	1	0	1
getSecondScore()		100%		n/a	0	1	0	1	0	1
Total	8 of 62	87%	0 of 4	100%	2	10	4	17	2	8

As you can see, the two setters have not been tested. This is one of the causes that we may not reach the 100 percent test coverage. These methods are auto-generated by the IDEA and there is no need for us to writing tests for them. So, let us continue our development.

Final Project Explanation

We have implemented the following entities. Also, you can see the main responsibility of each entity.

- BowlingConstants
  - For storing constant values related to the Bowling game. For example, the strike score, number of turns, etc
- FrameScore
  - A simple POJO object for storing scores in each frame. We also have two values for bonuses for the last turn (in case of spare or strike shot)
- ScoreFileReader
  - Reading a CSV file and returning an array of numbers.
  - This class does not know about the game logic or frame score. It only reads the file and returns a list of numbers.
- SequenceToFrameScoreConverter
  - This class is responsible for converting a list of numbers to a list of frame scores. This class knows about spare and strike shots (For example, in a strike shot, we only have one value for the frame score).
  - Also, this class knows about the bonuses that may occur in the last turn (in case of a spare or a strike shot).
- TotalScoreCalculator
  - This class receives a list of frame scores and calculates the total game score.
- GameResultPrinter
  - This class is responsible for printing the game result. It will use the TotalScoreCalculator to print the total game score.

Final Project Code Coverage

You can see the code coverage report of the final project.

mercel-task

mercel-task

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mercell.bowling		96%		94%	5	59	8	145	2	33	1	7
Total	25 of 703	96%	3 of 52	94%	5	59	8	145	2	33	1	7

As mentioned before, we are not forced to reach the 100 percent code coverage. Our final code coverage is 96 percent which is good enough.

Here comes the more detailed report of the code coverage.

mercel-task > com.mercell.bowling

com.mercell.bowling

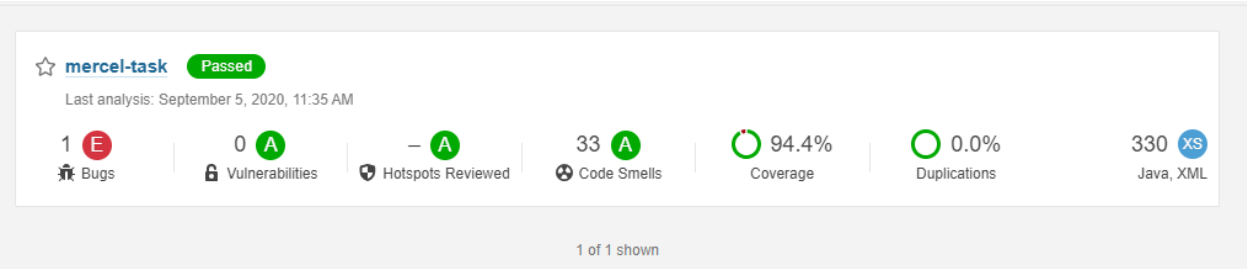
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main		46%		100%	1	3	7	14	1	2	0	1
BowlingConstants		0%		n/a	1	1	1	1	1	1	1	1
GameResultPrinter		100%		100%	0	14	0	32	0	7	0	1
TotalScoreCalculator		100%		100%	0	11	0	25	0	4	0	1
SequenceToFrameScoreConverter		100%		85%	2	9	0	35	0	2	0	1
FrameScore		100%		87%	1	19	0	31	0	15	0	1
ScoreFileReader		100%		n/a	0	2	0	7	0	2	0	1
Total	25 of 703	96%	3 of 52	94%	5	59	8	145	2	33	1	7

Analyzing the code with SonarQube

Now that we have implemented our task, let us check our code quality with SonarQube and get a report from this tool.

It is better to run SonarQube analysis in each commit and fix the code smells as soon as possible. Our task was small and simple and because of that, we run the SonarQube analysis at the end.

Here is the report



Based on this report, we have 1 Bug and 33 code smells.

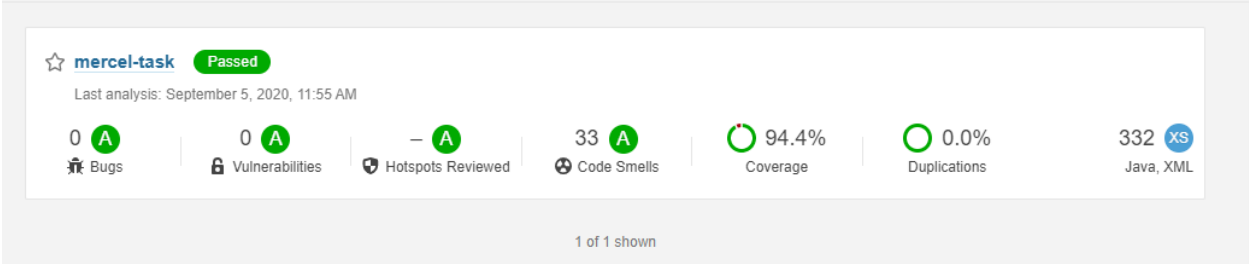
This is the detail description of our bug:

Let us fix this bug.

We change our implementation to the following

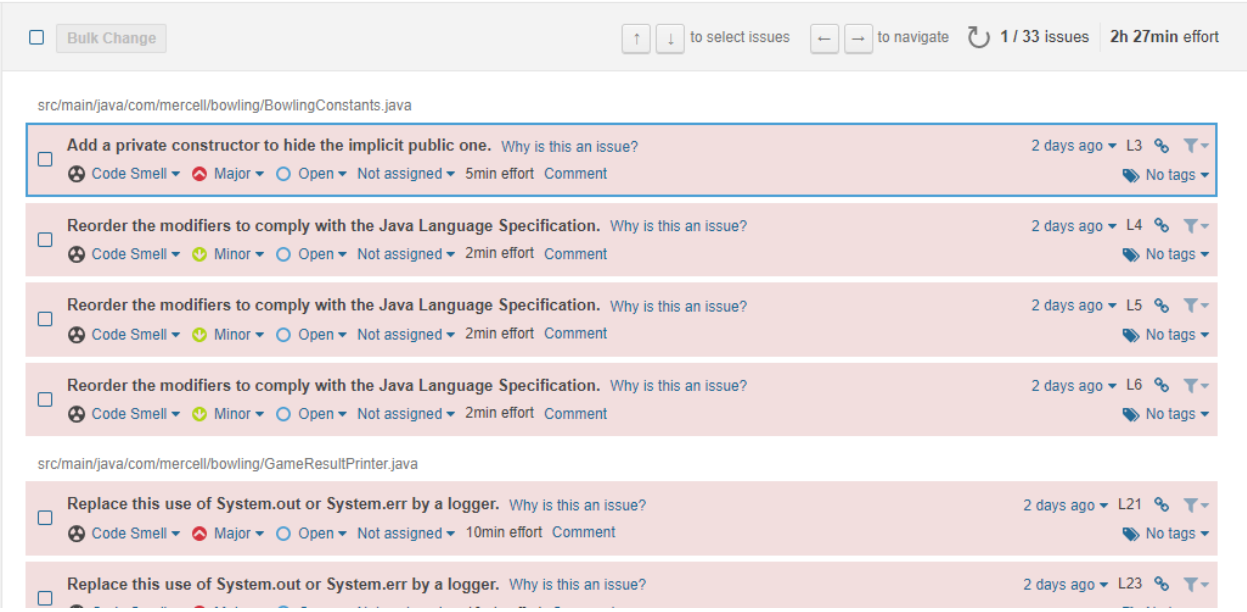
```
public List<Integer> readFileAndExtractTheScores() throws IOException {

    String line;
    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        line = br.readLine();
    }

    List<String> scoresAsStringList =
Arrays.asList(line.replaceAll("\\s+","").split(","));
    return
scoresAsStringList.stream().map(Integer::parseInt).collect(Collectors.toList());

}
```

Now, we run again the SonarQube. Our bug has been fixed 😊



Fixing the code smells

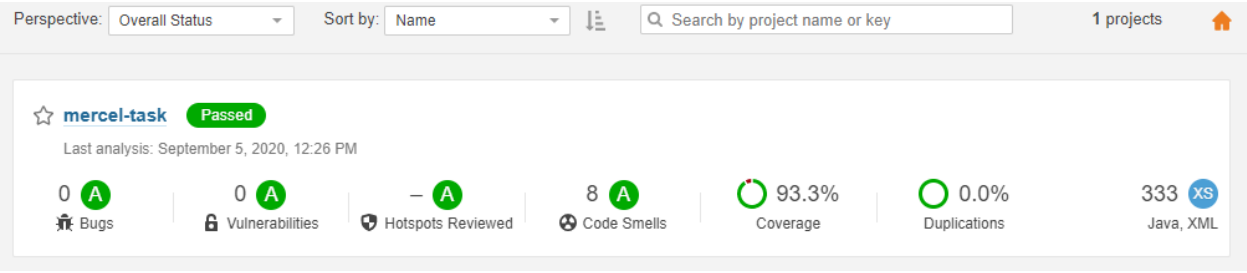
Here is the list of code smells:



We iterate over this list and fix them one by one. Most of them are simple and require a small amount of effort.

Also, some of them are not required for fixing. For example, these issues:

We improve our code by fixing these code smells. The final result is shown in the following picture:



Questions or Feedbacks

Thanks to take the time and read this report. If you have any question or want to provide some feedback, feel free to contact me through the following email address:

s.babak.mehrabi@gmail.com

With Regards,

Babak