Cody Wine

Unit Testing Report for JPacman

GitHub: https://github.com/L-u-t-o/jpacman
Branch: https://github.com/L-u-t-o/jpacman/tree/jpacman_tests

A quick analysis of unit testing prior to the work I will be conducting shows a 0% coverage for most packages present in the program. Figure 1 shows this coverage. For my testing report, I will conduct method tests for the following classes:

- Launcher.java
  - withMapFile()
- Board.java
  - squareAt()
- Pellet.java
  - Pellet()
  - getSprite

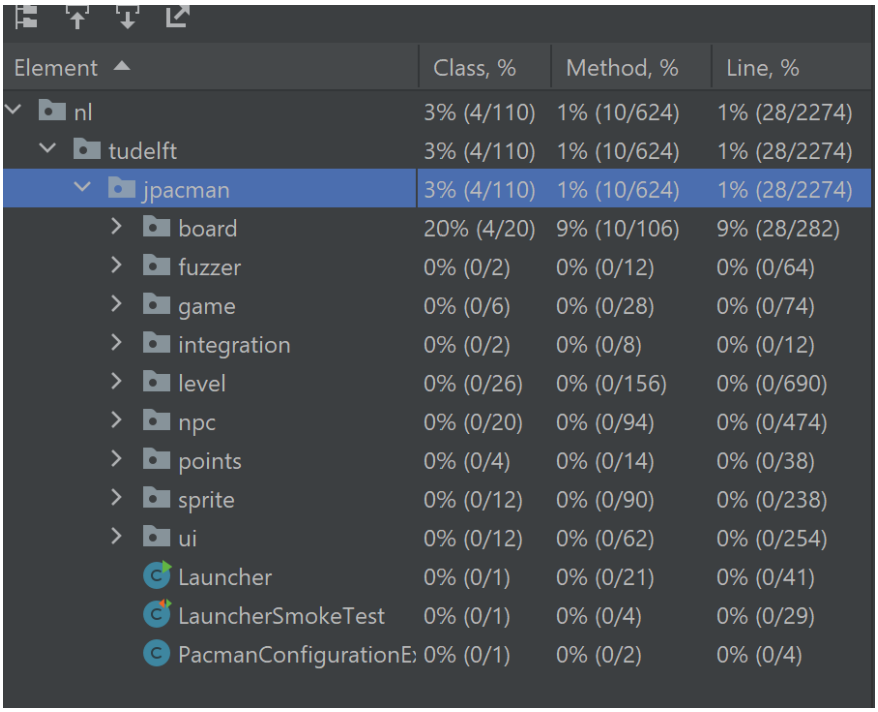

Figure 1

The process for each test involved similar steps: import the relevant libraries (including Test and AssertThat), instantiate the required objects for testing, and create a test function that will run the function from main that we are testing.

To begin, I tested the function withMapFile() from the class Launcher.java. Figure 2 shows how I instantiated the Function object and called the withMapFile() function for testing. Figure 3 shows the coverage before testing, while figure 4 shows the coverage after testing.

```java
package nl.tudelft.jpacman.ui;
import nl.tudelft.jpacman.Launcher;
import org.junit.jupiter.api.Test;


import static org.assertj.core.api.Assertions.assertThat;
no usages  Cody *
public class LauncherTest {

    1 usage
    private Launcher launchTest = new Launcher();
    1 usage
    String test = "testing123";
    no usages  Cody *
    @Test
    void testWithMapFile(){
        assertThat(launchTest.withMapFile(test));
    }
}
```

Figure 2

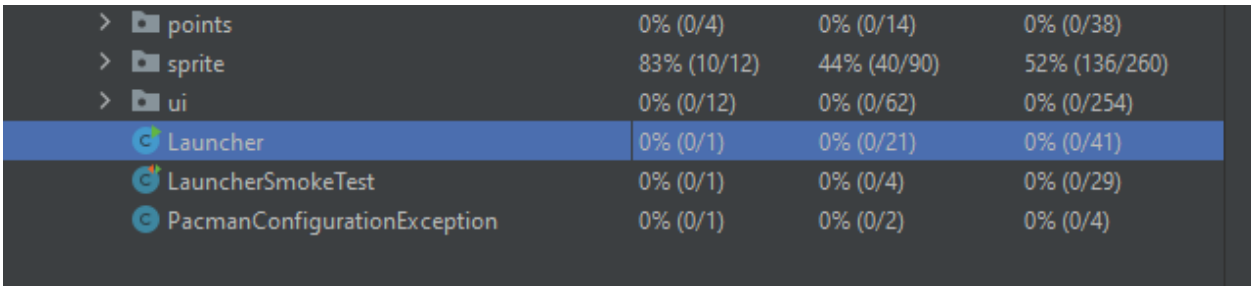

Figure 3 (BEFORE)

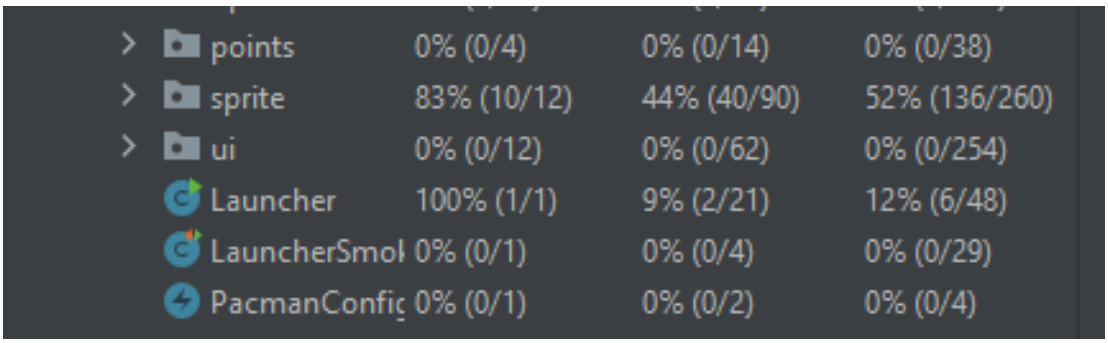

Figure 4 (AFTER)

Cody Wine

Continuing on, I ran my next method tests using the method squareAt() in class Board.java. This testing was more complicated as I needed to create a new square object and fill the square in that object as null boxes would error out the testing. The squareAt function is called as shown in figure 5. Figure 6 and Figure 7 show the coverage before and after the testing.

```java
package nl.tudelft.jpacman.board;
import nl.tudelft.jpacman.sprite.Sprite;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;
no usages  Cody *
public class BoardTest {

    no usages  Cody *
    @Test
    public void testSquareAt() {
        Square[][] grid = new Square[1][1];
        new *
        grid[0][0] = new Square() {
            new *
            @Override
            public boolean isAccessibleTo(Unit unit) {
                return false;
            }
            new *
            @Override
            public Sprite getSprite() {
                return null;
            }
        };
        Board board = new Board(grid);

        Square result = board.squareAt( x: 0,  y: 0);
        assertThat( actual: grid[0][0] == result);

    }
}
```

Figure 5

Element	Class, %	Method, %	Line, %
nl	20% (22/110)	10% (68/626)	9% (212/2322)
tudelft	20% (22/110)	10% (68/626)	9% (212/2322)
jpacman	20% (22/110)	10% (68/626)	9% (212/2322)
board	20% (4/20)	9% (10/106)	9% (28/282)
Board	0% (0/1)	0% (0/7)	0% (0/17)
BoardFactory	0% (0/3)	0% (0/11)	0% (0/27)
BoardFactoryTest	0% (0/1)	0% (0/6)	0% (0/18)
BoardTest	0% (0/1)	0% (0/3)	0% (0/3)

Figure 6 (BEFORE)

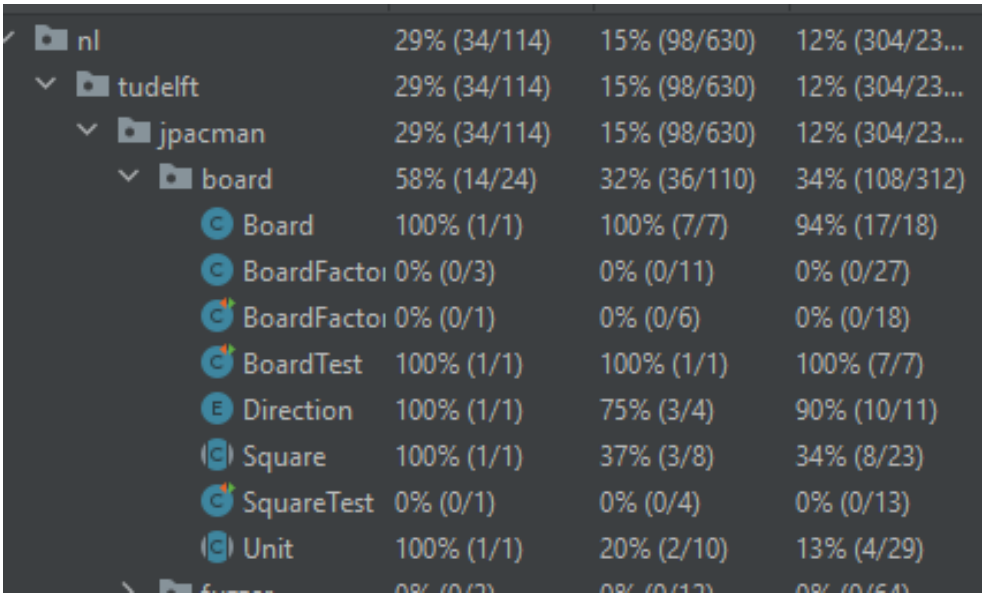

Figure 7 (AFTER)

Finally, a couple methods were tested from the class Pellet.java. This segment of testing proved to be most difficult as there were several steps involved to get a proper testing of the Pellet() constructor. Following a getSprite() method test, I used a generic value and sprite to test the new Pellet constructure mentioned. Figure 8 shows the code used and Figure 9 and Figure 10 show the before and after of testing.

```
package nl.tudelft.jpacman.level;

import nl.tudelft.jpacman.sprite.Sprite;
import org.junit.jupiter.api.Test;
import java.awt.*;
import static org.assertj.core.api.Assertions.assertThat;
no usages  Cody *
public class PelletTest {
    2 usages
    private Sprite image = null;
    1 usage
    private final int value = 0;
    no usages  Cody *
    @Test
    public void testGetSprite() {
        Cody *
        Sprite newSprite = new Sprite() {
            3 usages  Cody *
            @Override
            public void draw(Graphics graphics, int x, int y, int width, int height) {

            }
            6 usages  new *
            @Override
            public Sprite split(int x, int y, int width, int height) {
                return null;
            }
            new *
            @Override
            public int getWidth() {
                return 0;
            }
            new *
            @Override
            public int getHeight() {
                return 0;
            }
        };
        Pellet pellet = new Pellet( points: 0, newSprite);
        Sprite newerSprite = pellet.getSprite();
        assertThat(newerSprite).isEqualTo(newSprite);
        image = newSprite;
    }
    no usages  new *
    @Test
    void testPellet(){
        Pellet Pellet2 = new Pellet(value, image);
    }
}
```

Figure 8

Cody Wine

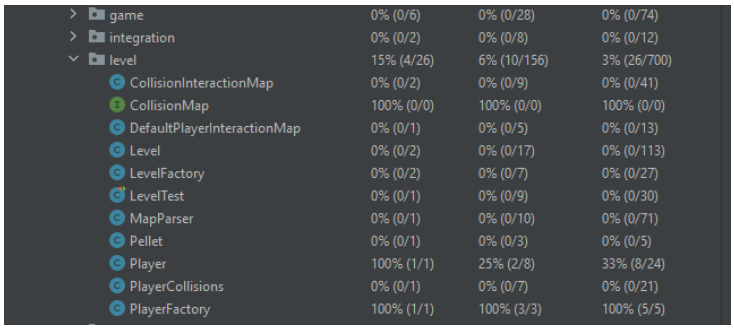

Figure 9 (BEFORE)

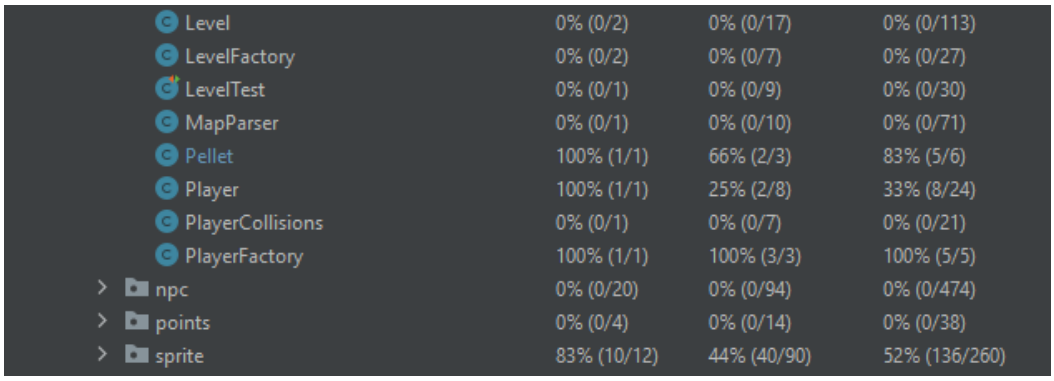

Figure 10 (AFTER)

In reference to the JaCoCo testing results shown in the following Figure, I can see several similarities between the coverage provided by IntelliJ. There is, however, an obvious discrepancy between the two in terms of results, as my own tests would have differed from the ones in Figure 11. While I like the simple coloring (red for missed, green for covered) of the JaCoCo report, I find the Element categorization and percentage monitoring more useful in the IntelliJ coverage. Although I may be used to the IntelliJ window, I do prefer it to the JaCoCo report.

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	74	155	104	344	21	69	4	12
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	25	4	46	2	15	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,213 of 4,696	74%	293 of 637	54%	293	591	229	1,040	51	269	6	47

Figure 11