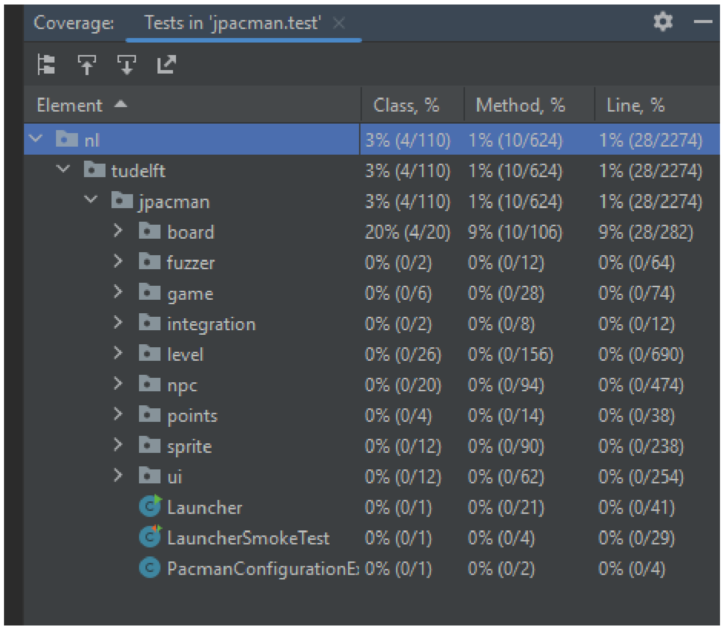
Isabella Capriotti
Dr. Businge
CS 472
28 January 2023

# Report: JPacman Dynamic Analysis

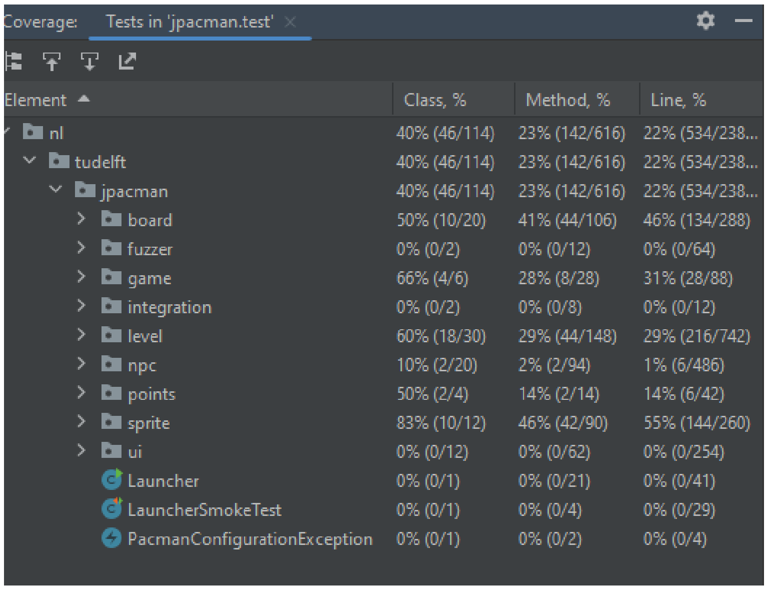
Task 2.1

Before adding additional unit test cases, this was the IntelliJ coverage report for the JPacman project:



| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| nl | 3% (4/110) | 1% (10/624) | 1% (28/2274) |
| tudelft | 3% (4/110) | 1% (10/624) | 1% (28/2274) |
| jpacman | 3% (4/110) | 1% (10/624) | 1% (28/2274) |
| board | 20% (4/20) | 9% (10/106) | 9% (28/282) |
| fuzzer | 0% (0/2) | 0% (0/12) | 0% (0/64) |
| game | 0% (0/6) | 0% (0/28) | 0% (0/74) |
| integration | 0% (0/2) | 0% (0/8) | 0% (0/12) |
| level | 0% (0/26) | 0% (0/156) | 0% (0/690) |
| npc | 0% (0/20) | 0% (0/94) | 0% (0/474) |
| points | 0% (0/4) | 0% (0/14) | 0% (0/38) |
| sprite | 0% (0/12) | 0% (0/90) | 0% (0/238) |
| ui | 0% (0/12) | 0% (0/62) | 0% (0/254) |
| Launcher | 0% (0/1) | 0% (0/21) | 0% (0/41) |
| LauncherSmokeTest | 0% (0/1) | 0% (0/4) | 0% (0/29) |
| PacmanConfigurationE: | 0% (0/1) | 0% (0/2) | 0% (0/4) |

On top of the required test for isAlive() in the Player class, the three additional methods I chose to test were src/main/java/nl/tudelft/jpacman/level/Level.java@isAnyPlayerAlive(), src/main/java/nl/tudelft/jpacman/points/DefaultPointCalculator.java@consumedAPellete(), and src/main/java/nl/tudelft/jpacman/game/SinglePlayerGame.java@getPlayers(). After adding the new unit tests, this was the new coverage report:

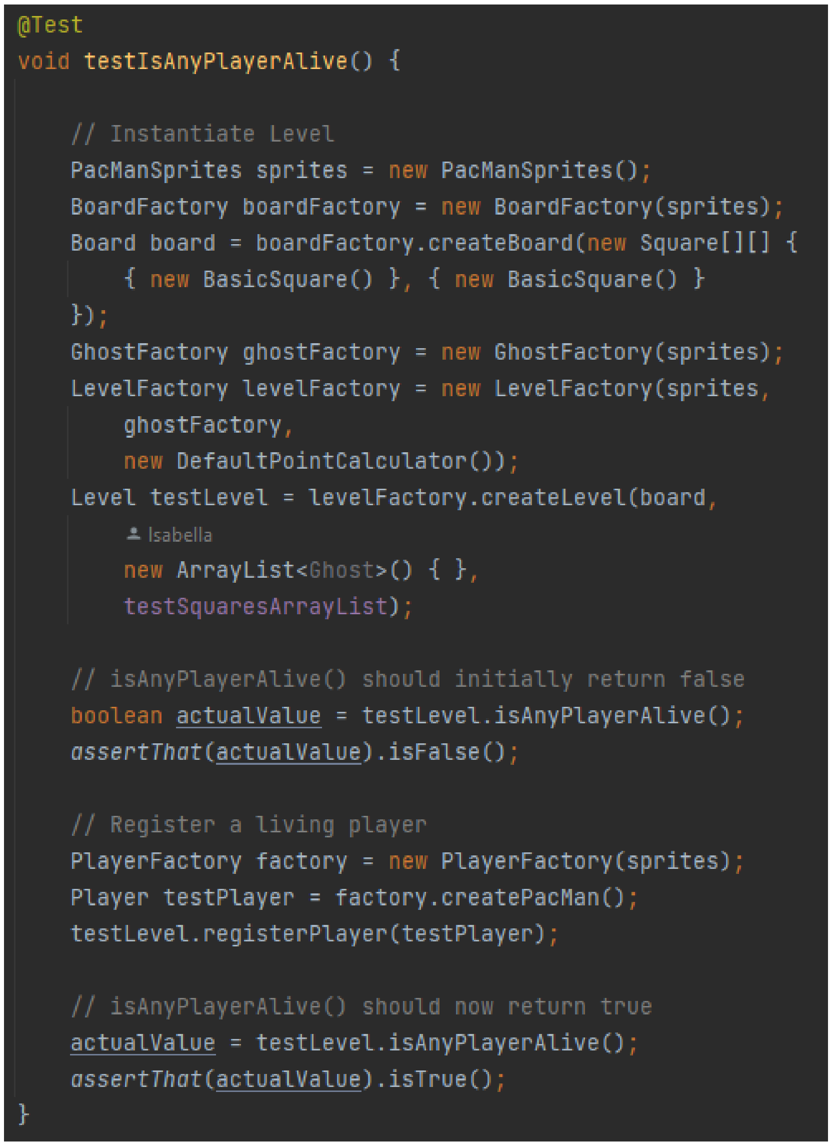


| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| nl | 40% (46/114) | 23% (142/616) | 22% (534/238... |
| tudelft | 40% (46/114) | 23% (142/616) | 22% (534/238... |
| jpacman | 40% (46/114) | 23% (142/616) | 22% (534/238... |
| board | 50% (10/20) | 41% (44/106) | 46% (134/288) |
| fuzzer | 0% (0/2) | 0% (0/12) | 0% (0/64) |
| game | 66% (4/6) | 28% (8/28) | 31% (28/88) |
| integration | 0% (0/2) | 0% (0/8) | 0% (0/12) |
| level | 60% (18/30) | 29% (44/148) | 29% (216/742) |
| npc | 10% (2/20) | 2% (2/94) | 1% (6/486) |
| points | 50% (2/4) | 14% (2/14) | 14% (6/42) |
| sprite | 83% (10/12) | 46% (42/90) | 55% (144/260) |
| ui | 0% (0/12) | 0% (0/62) | 0% (0/254) |
| Launcher | 0% (0/1) | 0% (0/21) | 0% (0/41) |
| LauncherSmokeTest | 0% (0/1) | 0% (0/4) | 0% (0/29) |
| PacmanConfigurationException | 0% (0/1) | 0% (0/2) | 0% (0/4) |

The overall coverage of all classes in the project increased from 3% to 40%. The level, points, and game packages, which previously all had 0% class coverage, are now at 60%, 50%, and 66% class coverage respectively. The method and line coverage for these packages similarly increased.

The following section will summarize the code written for each new unit test.

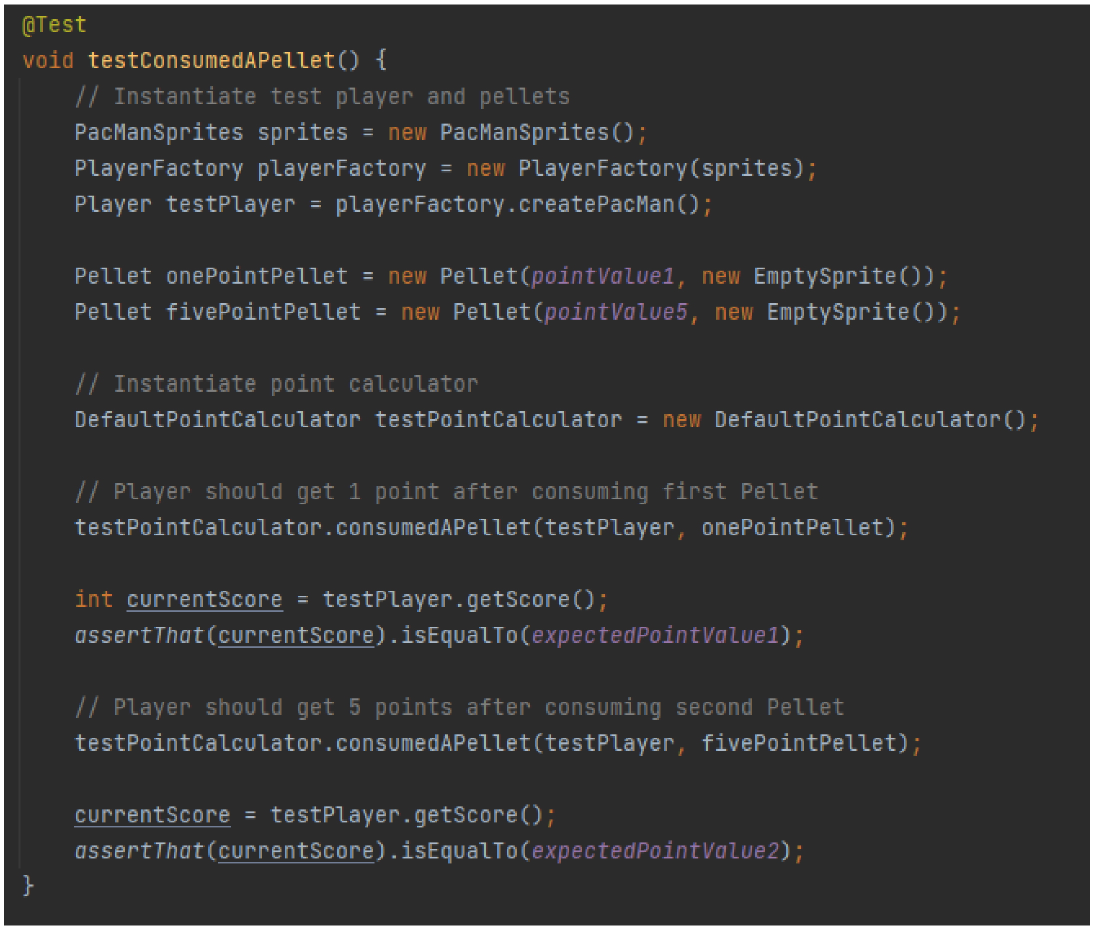
I. src/main/java/nl/tudelft/jpacman/level/Level.java@isAnyPlayerAlive()



```java
@Test
void testIsAnyPlayerAlive() {

    // Instantiate Level
    PacManSprites sprites = new PacManSprites();
    BoardFactory boardFactory = new BoardFactory(sprites);
    Board board = boardFactory.createBoard(new Square[][] {
        { new BasicSquare() }, { new BasicSquare() }
    });
    GhostFactory ghostFactory = new GhostFactory(sprites);
    LevelFactory levelFactory = new LevelFactory(sprites,
        ghostFactory,
        new DefaultPointCalculator());
    Level testLevel = levelFactory.createLevel(board,
        Isabella
        new ArrayList<Ghost>() { },
        testSquaresArrayList);

    // isAnyPlayerAlive() should initially return false
    boolean actualValue = testLevel.isAnyPlayerAlive();
    assertThat(actualValue).isFalse();

    // Register a living player
    PlayerFactory factory = new PlayerFactory(sprites);
    Player testPlayer = factory.createPacMan();
    testLevel.registerPlayer(testPlayer);

    // isAnyPlayerAlive() should now return true
    actualValue = testLevel.isAnyPlayerAlive();
    assertThat(actualValue).isTrue();
}
```

This unit test instantiates a level with no players and verifies that the isAnyPlayerAlive() method returns false initially. After adding a living player to the level, it executes isAnyPlayerAlive() again and ensures that it now returns true.

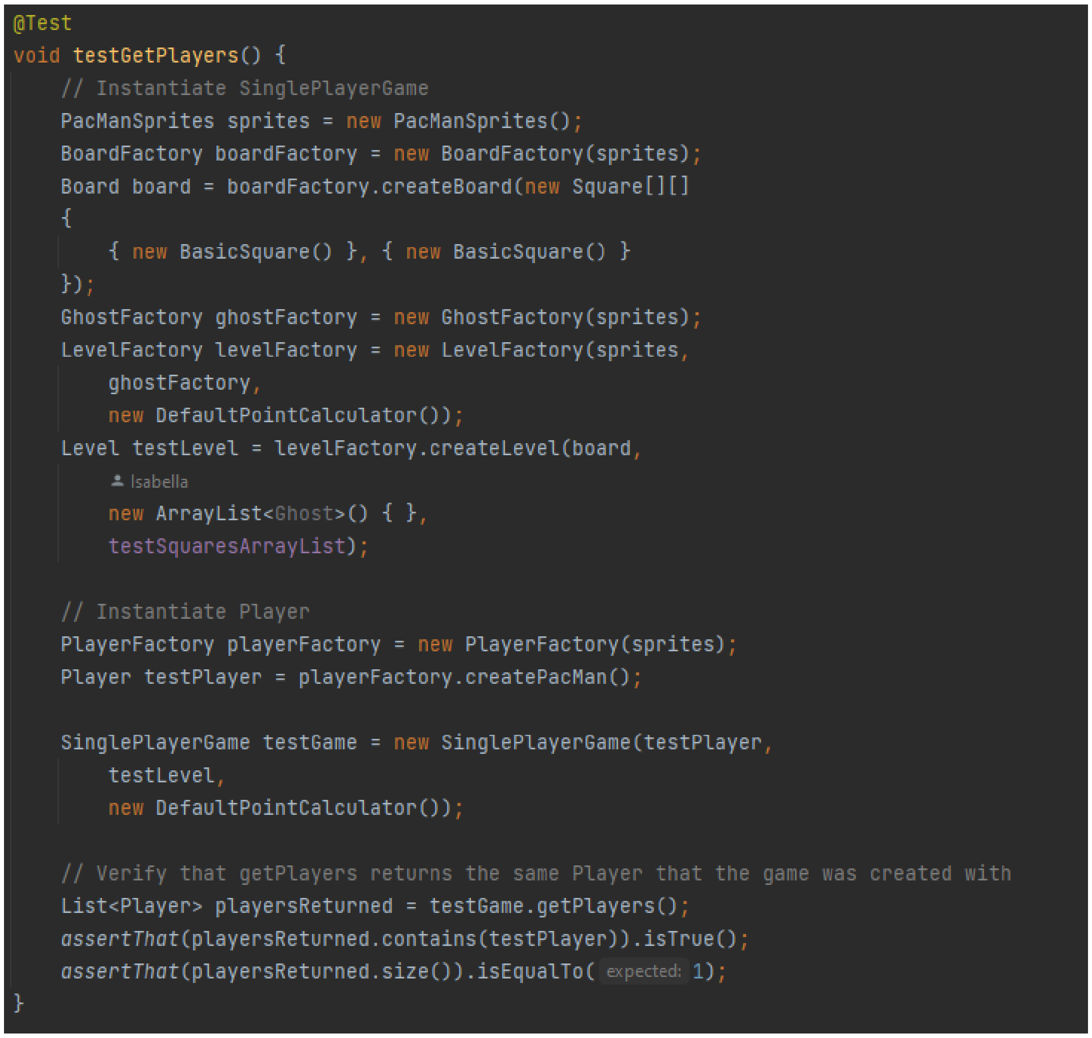
II. src/main/java/nl/tudelft/jpacman/points/DefaultPointCalculator.java@consumedAPellete()



```java
@Test
void testConsumedAPellet() {
    // Instantiate test player and pellets
    PacManSprites sprites = new PacManSprites();
    PlayerFactory playerFactory = new PlayerFactory(sprites);
    Player testPlayer = playerFactory.createPacMan();

    Pellet onePointPellet = new Pellet(pointValue1, new EmptySprite());
    Pellet fivePointPellet = new Pellet(pointValue5, new EmptySprite());

    // Instantiate point calculator
    DefaultPointCalculator testPointCalculator = new DefaultPointCalculator();

    // Player should get 1 point after consuming first Pellet
    testPointCalculator.consumedAPellet(testPlayer, onePointPellet);

    int currentScore = testPlayer.getScore();
    assertThat(currentScore).isEqualTo(expectedPointValue1);

    // Player should get 5 points after consuming second Pellet
    testPointCalculator.consumedAPellet(testPlayer, fivePointPellet);

    currentScore = testPlayer.getScore();
    assertThat(currentScore).isEqualTo(expectedPointValue2);
}
```

This unit instantiates a player and two pellets with different point values to test the DefaultPointCalculator with. For each of the two pellets, it calls the consumedAPellet() method and ensures that the player's score is updated to the expected value after consuming the pellet.

III. src/main/java/nl/tudelft/jpacman/game/SinglePlayerGame.java@getPlayers()

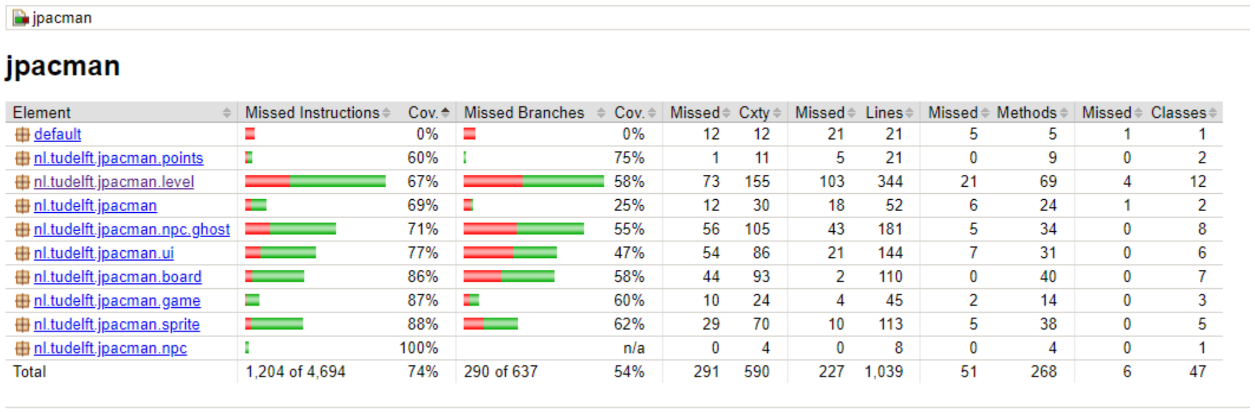


```
@Test
void testGetPlayers() {
    // Instantiate SinglePlayerGame
    PacManSprites sprites = new PacManSprites();
    BoardFactory boardFactory = new BoardFactory(sprites);
    Board board = boardFactory.createBoard(new Square[][]
    {
        { new BasicSquare() }, { new BasicSquare() }
    });
    GhostFactory ghostFactory = new GhostFactory(sprites);
    LevelFactory levelFactory = new LevelFactory(sprites,
        ghostFactory,
        new DefaultPointCalculator());
    Level testLevel = levelFactory.createLevel(board,
        Isabella
        new ArrayList<Ghost>() { },
        testSquaresArrayList);

    // Instantiate Player
    PlayerFactory playerFactory = new PlayerFactory(sprites);
    Player testPlayer = playerFactory.createPacMan();

    SinglePlayerGame testGame = new SinglePlayerGame(testPlayer,
        testLevel,
        new DefaultPointCalculator());

    // Verify that getPlayers returns the same Player that the game was created with
    List<Player> playersReturned = testGame.getPlayers();
    assertThat(playersReturned.contains(testPlayer)).isTrue();
    assertThat(playersReturned.size()).isEqualTo( expected: 1);
}
```

This unit test instantiates a level and a player, then builds a single player game with that level and player. It then calls getPlayers() on the newly created single player game and ensures that it returns the same player the level was created with.

Task 3

Below is a screenshot of the JaCoCo coverage report referenced in the following questions.



jpacman

## jpacman

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| default | | 0% | | 0% | 12 | 12 | 21 | 21 | 5 | 5 | 1 | 1 |
| nl.tudelft.jpacman.points | | 60% | | 75% | 1 | 11 | 5 | 21 | 0 | 9 | 0 | 2 |
| nl.tudelft.jpacman.level | | 67% | | 58% | 73 | 155 | 103 | 344 | 21 | 69 | 4 | 12 |
| nl.tudelft.jpacman | | 69% | | 25% | 12 | 30 | 18 | 52 | 6 | 24 | 1 | 2 |
| 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 |
| nl.tudelft.jpacman.board | | 86% | | 58% | 44 | 93 | 2 | 110 | 0 | 40 | 0 | 7 |
| nl.tudelft.jpacman.game | | 87% | | 60% | 10 | 24 | 4 | 45 | 2 | 14 | 0 | 3 |
| nl.tudelft.jpacman.sprite | | 88% | | 62% | 29 | 70 | 10 | 113 | 5 | 38 | 0 | 5 |
| nl.tudelft.jpacman.npc | | 100% | | n/a | 0 | 4 | 0 | 8 | 0 | 4 | 0 | 1 |
| Total | 1,204 of 4,694 | 74% | 290 of 637 | 54% | 291 | 590 | 227 | 1,039 | 51 | 268 | 6 | 47 |

**Question 1:** Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?

The coverage results from JaCoCo are similar, but not identical, to the values obtained with IntelliJ in the previous task. This is likely because JaCoCo uses a different formula that factors in branch coverage while calculating the overall coverage. For example, the IntelliJ coverage report puts the level package at 60% coverage, while the JaCoCo coverage report puts the level package at 67% coverage. This could be because adding unit tests for the methods isAlive() and isAnyPlayerAlive() from the level package positively impacted the coverage on a high number of code branches because of how frequently these methods are used in the source code.

**Question 2:** Did you find helpful the source code visualization from JaCoCo on uncovered branches?

Yes, I found this visualization helpful. This is because analyzing the branches that execution can take gives a more realistic view of test coverage than looking at the presence of a test for a method alone. Even if a method is covered by a test, if that test doesn't verify a key execution path or a vulnerable edge case, it is not safe to say that it is fully covered. Therefore, this visualization is very useful for identifying potential scenarios that still need to be tested.

**Question 3:** Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?

Overall, I prefer JaCoCo's visualization because it gives a more detailed breakdown of the coverage by including the branch coverage and visualization of the source code. Additionally, whereas IntelliJ's coverage window puts the number of methods/classes/lines covered in terms of how many out of the total are covered, JaCoCo puts it in terms of how many are missed. This leads to a more actionable report, since developers can target the items that are missed instead of only thinking about what is already covered. Finally, in JaCoCo, you can drill down into specific methods within classes, while IntelliJ's granularity stops at the class level. Again, this leads to a deeper understanding on the developer side of which specific workflows in the program need more thorough testing, as opposed to just having a high level overview like the one in IntelliJ. One advantage of the IntelliJ coverage window is that the UI is more intuitive; however, once I learned how to read the JaCoCo report, I found that it contains more valuable information.

Important Links
**Fork Repository:** https://github.com/IsabellaCapriotti/jpacman/tree/jpacman_tests
**Shared Team Repository:** https://github.com/justin-negron/SoftwareProductDesign