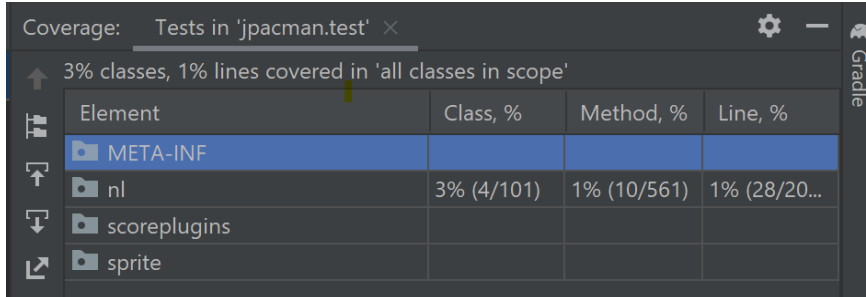


## JaCoCo Dynamic Analysis

### Task 2.1

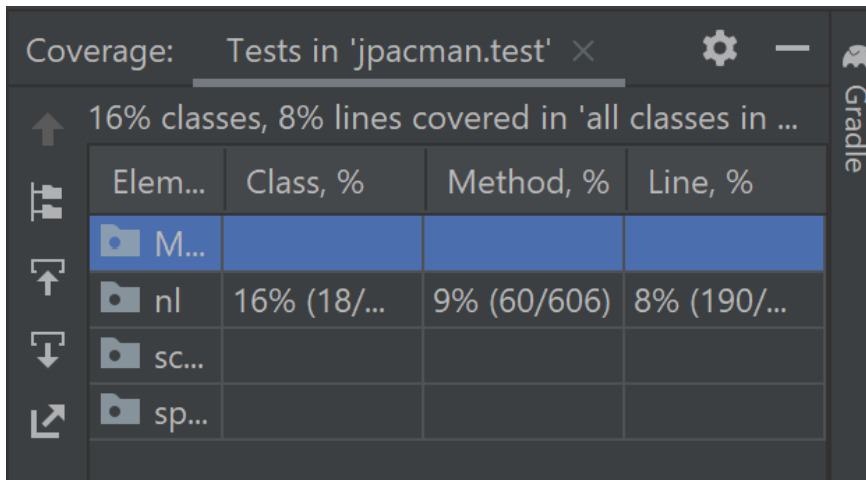
#### Base Coverage Before Any Tests



Coverage: Tests in 'jpacman.test' ×			
3% classes, 1% lines covered in 'all classes in scope'			
Element	Class, %	Method, %	Line, %
META-INF			
nl	3% (4/101)	1% (10/561)	1% (28/20...)
scoreplugins			
sprite			

#### Coverage After PlayerTest.isAlive

```
1 package nl.tudelft.jpacman.level;
2 import nl.tudelft.jpacman.sprite.PacManSprites;
3 import org.junit.jupiter.api.Test;
4
5
6 public class PlayerTest {
7     private static final PacManSprites spriteObj = new PacManSprites();
8     private final PlayerFactory pFac = new PlayerFactory(spriteObj);
9     private Player player = pFac.createPacman();
10
11
12     @Test
13     void isAliveTest() {
14         boolean test = player.isAlive();
15     }
16 }
17
```



Coverage: Tests in 'jpacman.test' ×			
16% classes, 8% lines covered in 'all classes in ...'			
Elem...	Class, %	Method, %	Line, %
M...			
nl	16% (18/...	9% (60/606)	8% (190/...
sc...			
sp...			

### Coverage After CollisionInteractionMapTest.CollisionInteractionMap

```
1 package nl.tudelft.jpacman.level;
2 import org.junit.jupiter.api.Test;
3
4 public class CollisionInteractionMapTest {
5     CollisionInteractionMap cmObj = new CollisionInteractionMap();
6
7     @Test
8     void CollisionInteractionMapTest() { new CollisionInteractionMap(); }
9
10 }
11
12
```

Coverage: Tests in 'jpacman.test' × [gear icon] [minus icon] [Gradle icon]

↑ 17% classes, 8% lines covered in 'all classes in ...'

	Elem...	Class, %	Method, %	Line, %
📁	M...			
📁	nl	17% (20/1...	10% (62/6...	8% (196/...
📁	sc...			
📁	s...			

📁 📄 🔍 ↕

### Coverage After DefaultPointCalculatorTest.consumedAPellet

```
1 package nl.tudelft.jpacman.points;
2 import nl.tudelft.jpacman.level.Pellet;
3 import nl.tudelft.jpacman.level.Player;
4 import nl.tudelft.jpacman.level.PlayerFactory;
5 import nl.tudelft.jpacman.sprite.PacManSprites;
6 import nl.tudelft.jpacman.sprite.Sprite;
7 import org.junit.jupiter.api.Test;
8
9 public class DefaultPointCalculatorTest {
10     private static final PacManSprites spriteObj = new PacManSprites();
11     private final PlayerFactory pFac = new PlayerFactory(spriteObj);
12     private Player player = pFac.createPacMan();
13     int points;
14     Sprite sprite;
15     public Pellet pellet = new Pellet(points, sprite);
16     DefaultPointCalculator dpc = new DefaultPointCalculator();
17
18     @Test
19     void consumedAPelletTest() { dpc.consumedAPellet(player, pellet); }
20
21 }
22
23
```

Coverage: Tests in 'jpacman.test' ×

21% classes, 9% lines covered in 'all classes in ...'

Elem...	Class, %	Method, %	Line, %
...			
nl	21% (24/1...)	11% (70/6...)	9% (216/2...)
S...			
S...			

Gradle

### Coverage After BoardFactoryTest.BoardFactory

```

1 package nl.tudelft.jpacman.board;
2
3 import nl.tudelft.jpacman.sprite.PacManSprites;
4 import org.junit.jupiter.api.Test;
5
6 public class BoardFactoryTest {
7     private static final PacManSprites spriteObj = new PacManSprites();
8     BoardFactory bf = new BoardFactory(spriteObj);
9
10
11     @Test
12     void BoardFactoryTest() { BoardFactory testbf = new BoardFactory(spriteObj); }
13
14
15 }
16
17

```

Coverage: Tests in 'jpacman.test' ×

26% classes, 10% lines covered in 'all classes i...'

Elem...	Class, %	Method, %	Line, %
...			
nl	26% (30/1...)	13% (82/6...)	10% (244/...)
S...			
S...			

Gradle

### Task 3

- *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 **not** similar to the ones I got from IntelliJ in the last task. The results differ because JaCoCo reports the coverage percent of Missed Instructions and Missed Branches whereas IntelliJ reports the coverage percent of Classes, Methods, and Lines. They simply differ in what is reported.

- *Did you find helpful the source code visualization from JaCoCo on uncovered branches?*

I found the source code visualization from JaCoCo on the uncovered branches helpful. It gives the user a better idea of how many files have been covered. I feel that the visual representation also allows for more detail to be shown across all files in terms of lines of code covered.

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

I preferred JaCoCo's visualization because it provided an in-depth analysis of the code coverage. The visualization also went through the code line by line to show what has been covered versus what has not. The IntelliJ coverage window provided more of a general overview rather than an in-depth analysis.