Task 1 Question(s):

- Is the coverage good enough?

The coverage report indicates that only a small portion of the code has been tested and covered. Out of the total number of Classes (110), only 4 of them are covered, which is a mere 3% coverage. In terms of Methods, only 1% (10 out of 624) have been covered. The coverage of Lines is even lower, with only 1% (28 out of 2274) being covered.

This low coverage may indicate that the existing test cases are inadequate and more tests need to be added to ensure that the code is thoroughly tested and functioning as expected.

Task 3 Question(s):

- 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 report from IntelliJ and JaCoCo are different in several ways.

IntelliJ's coverage report is integrated within the IntelliJ IDE and provides detailed information about code coverage such as the number of executed lines, missed lines, missed branches, etc. This information is easily accessible within the IDE, which makes it convenient for developers to review and analyze the results. Additionally, IntelliJ provides a visual representation of the code, which can help developers understand the structure and flow of the code.

JaCoCo, on the other hand, is a standalone tool that provides basic coverage metrics such as the percentage of code that has been executed by the test cases. JaCoCo provides a simple and straightforward representation of code coverage, but does not provide the same level of detail or accessibility as IntelliJ's coverage report.

The coverage report from IntelliJ and JaCoCo are different in terms of the level of detail they provide, the accessibility of the information, and the tool integrations they offer. These differences may influence the choice between the two, depending on the specific needs and preferences of the development team.

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

The source code visualization from JaCoCo on uncovered branches provides valuable insights into the quality of software testing. By visualizing the uncovered branches, we can easily identify areas of the code that need to be tested and focus their efforts on improving the coverage.

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

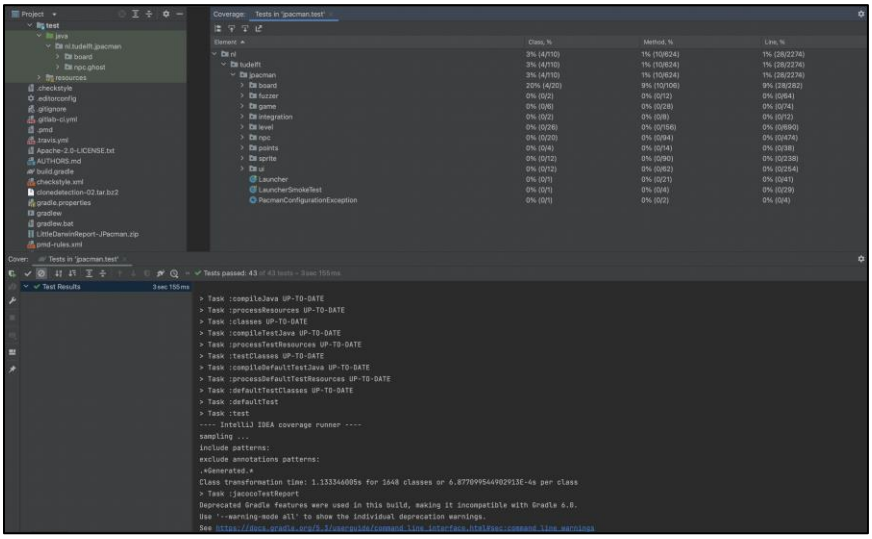
IntelliJ's coverage report provides detailed information about code coverage, including the number of executed lines, missed lines, missed branches, etc. This information is easily accessible within the IDE, which makes it convenient for developers to review and analyze the results.

In contrast, JaCoCo is a standalone tool that provides basic coverage metrics such as the percentage of code that has been executed by test cases. While JaCoCo can be integrated with other tools such as Jenkins, Gradle, etc, it does not provide the same level of detail and accessibility as IntelliJ's coverage report.

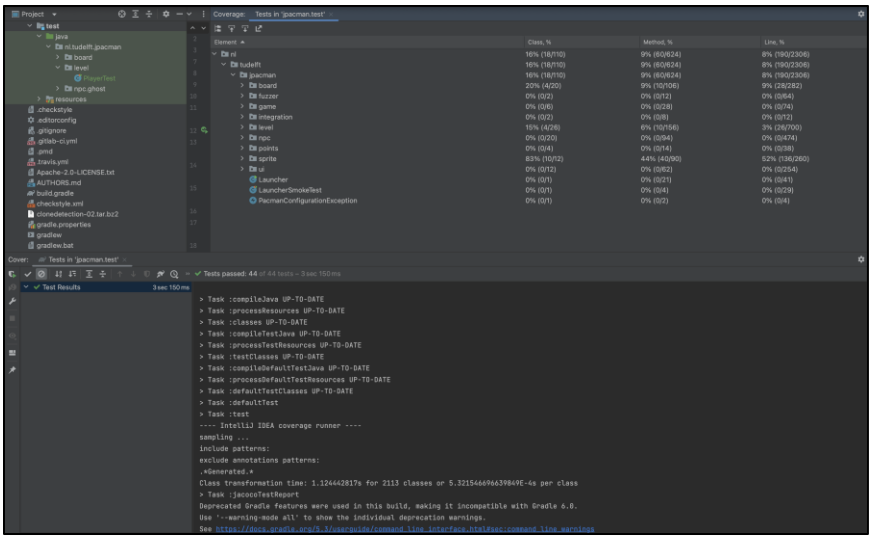
Another advantage of IntelliJ's coverage report is that it provides a visual representation of the code, which can help developers understand the structure and flow of the code. This can be particularly helpful in identifying areas that are difficult to test, or areas where tests are failing.

In summary, the coverage report from IntelliJ is considered better than JaCoCo due to its integration within the IDE, the additional metrics it provides, and the visual representation of the code. These features provide a more comprehensive and accessible view of the code coverage, which can help developers improve the quality of their testing.

IntelliJ Test Report Task 1:



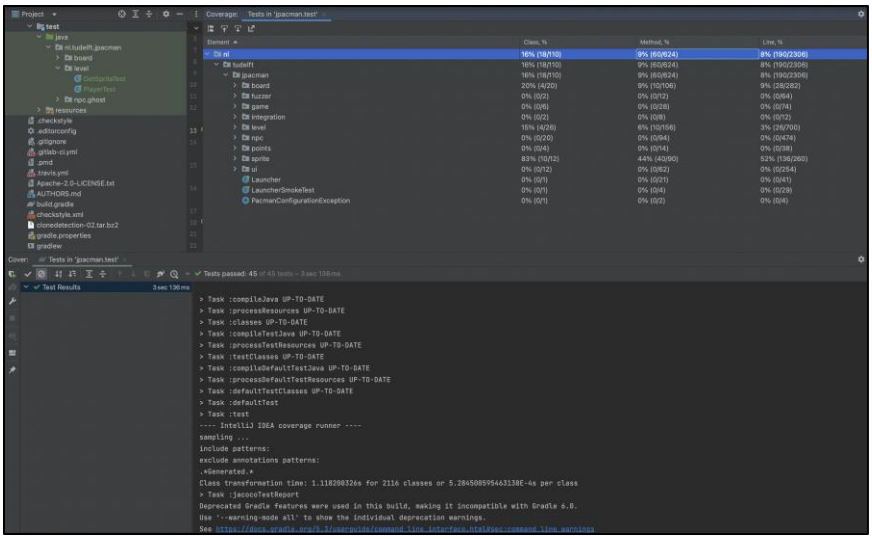
IntelliJ Test Report Task 2:



Task 2 Code Segment:

```java
package nl.tudelft.jpacman.level;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;
/**
 * New Test Case example
 * @author Devyn Gilliam
 */
no usages  devyngilliam
public class PlayerTest {
    // Instantiate the PacManPSprites object to retrieve PlayerFactory constructor
    1 usage
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    1 usage
    private PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
    1 usage
    private Player ThePlayer = Factory.createPacMan();

    no usages  devyngilliam
    @Test
    void testAlive() { assertThat(ThePlayer.isAlive()).isEqualTo( expected: true); }
}
```

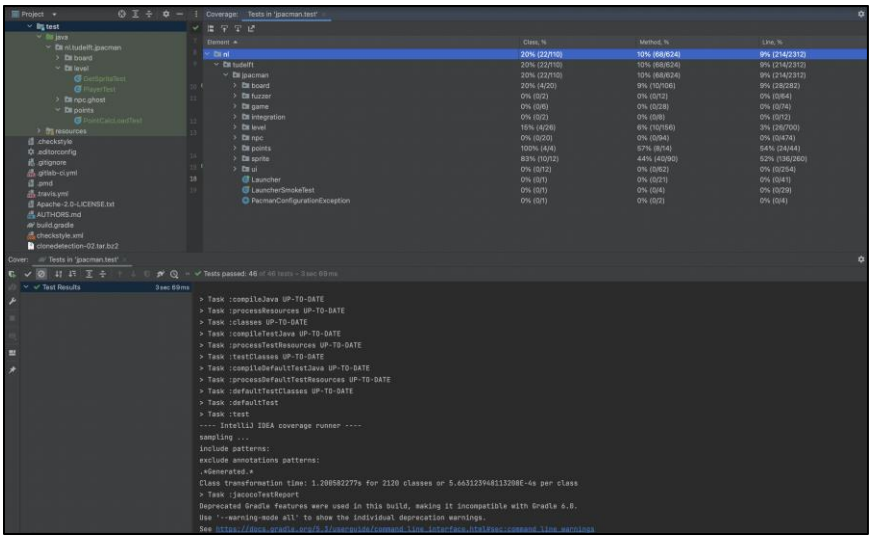
IntelliJ Test Report Task 2.1 (Method 1):



Task 2.1 Code Segment (Method 1):

```java
package nl.tudelft.jpacman.level;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;
/**
 * getSprites() test
 * @author Devyn Gilliam
 */
no usages  devyngilliam
public class GetSpriteTest {

    1 usage
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    1 usage
    private PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
    no usages  devyngilliam
    @Test
    void testGetSprite() { assertThat(Factory.getSprites()).isInstanceOf(PacManSprites.class); }
}
```

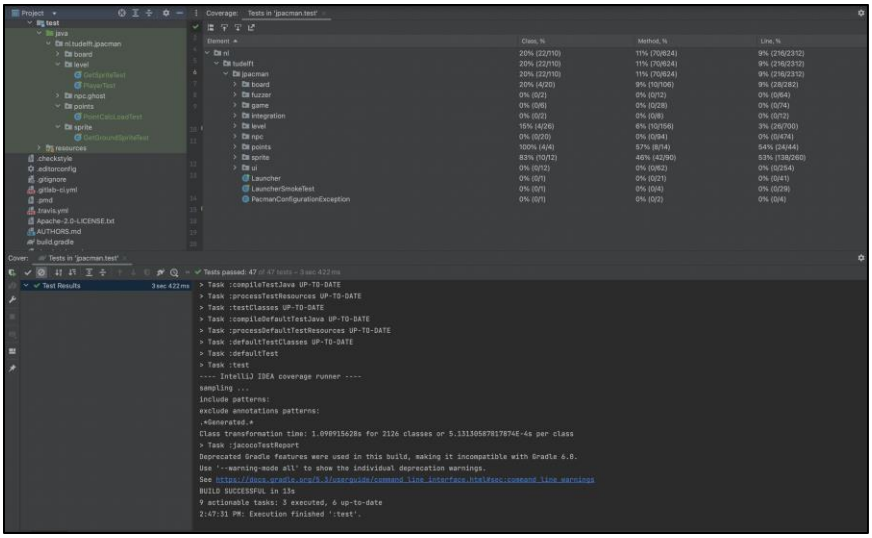
IntelliJ Test Report Task 2.1 (Method 2):



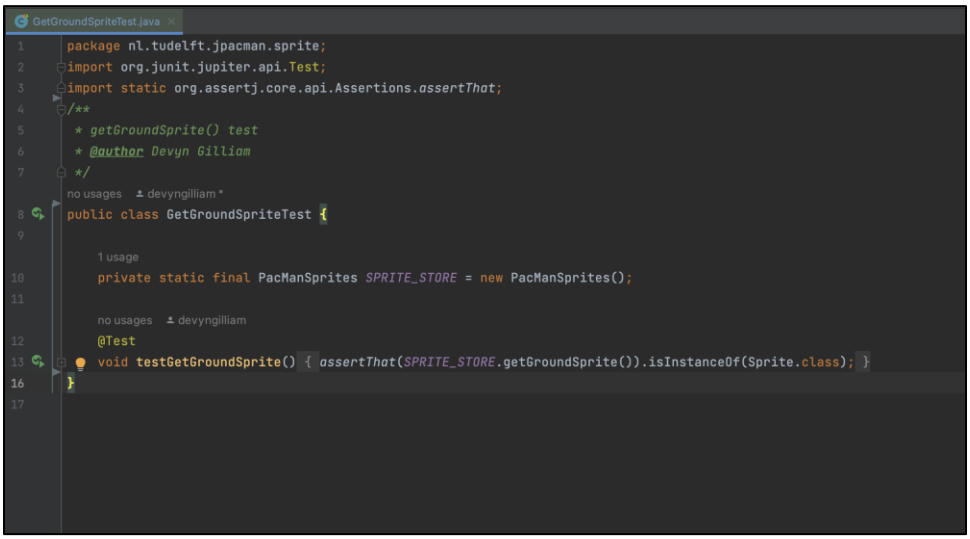
Task 2.1 Code Segment (Method 2):

```java
package nl.tudelft.jpacman.points;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;
/**
 * load() test
 * @author Devyn Gilliam
 */
no usages  devyngilliam
public class PointCalcLoadTest {

    1 usage
    public PointCalculatorLoader PCL = new PointCalculatorLoader();

    no usages  devyngilliam
    @Test
    void testPointCalcLoad() { assertThat(PCL.load()).isInstanceOf(PointCalculator.class); }
}
```

IntelliJ Test Report Task 2.1 (Method 3):



Task 2.1 Code Segment (Method 3):



```java
package nl.tudelft.jpacman.sprite;
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;
/**
 * getGroundSprite() test
 * @author Devyn Gilliam
 */
public class GetGroundSpriteTest {

    private static final PacManSprites SPRITE_STORE = new PacManSprites();

    @Test
    void testGetGroundSprite() { assertThat(SPRITE_STORE.getGroundSprite()).isInstanceOf(Sprite.class); }
}
```

Fork Repository: https://github.com/devyngilliam/jpacman