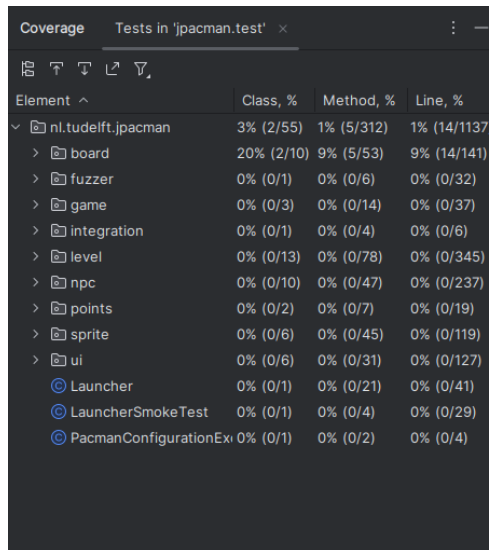


**GITHUB REPO:** <https://github.com/AlvaroH2001/cs472-group4>

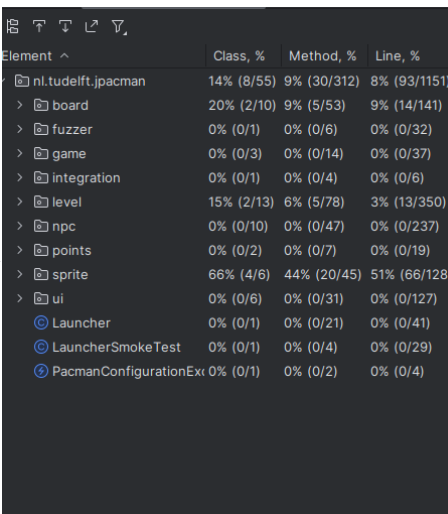
## Task 2.1 and 3

The initial coverage of this program was extremely low, with the Class, Method, and Line percentages being 3%, 1%, and 1% coverage respectively. This coverage is not enough to say with much confidence that this program will work under most conditions.



Element ^	Class, %	Method, %	Line, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	0% (0/13)	0% (0/78)	0% (0/345)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	0% (0/6)	0% (0/45)	0% (0/119)
ui	0% (0/6)	0% (0/31)	0% (0/127)
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)

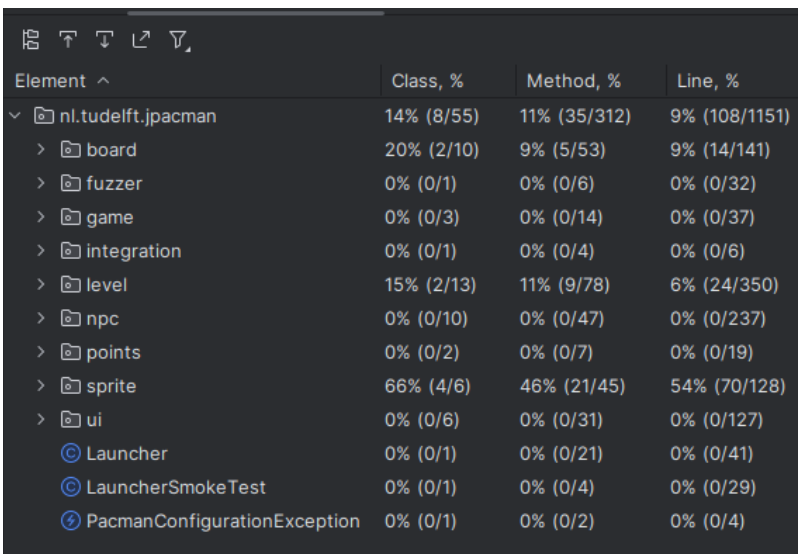
After adding in the initial test case (similar to the example that was provided by the instructor) that was used to determine if the `player.isAlive()` method works, the percentage shot up to 14%, 9% and 8% coverage. This resulted from the fact that not just was the `isAlive()` function being tested in the example but also the other functions that were used to initialize and create the objects like the factory, the sprites, and player itself.



Element ^	Class, %	Method, %	Line, %
nl.tudelft.jpacman	14% (8/55)	9% (30/312)	8% (93/1151)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	15% (2/13)	6% (5/78)	3% (13/350)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	66% (4/6)	44% (20/45)	51% (66/128)
ui	0% (0/6)	0% (0/31)	0% (0/127)
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)

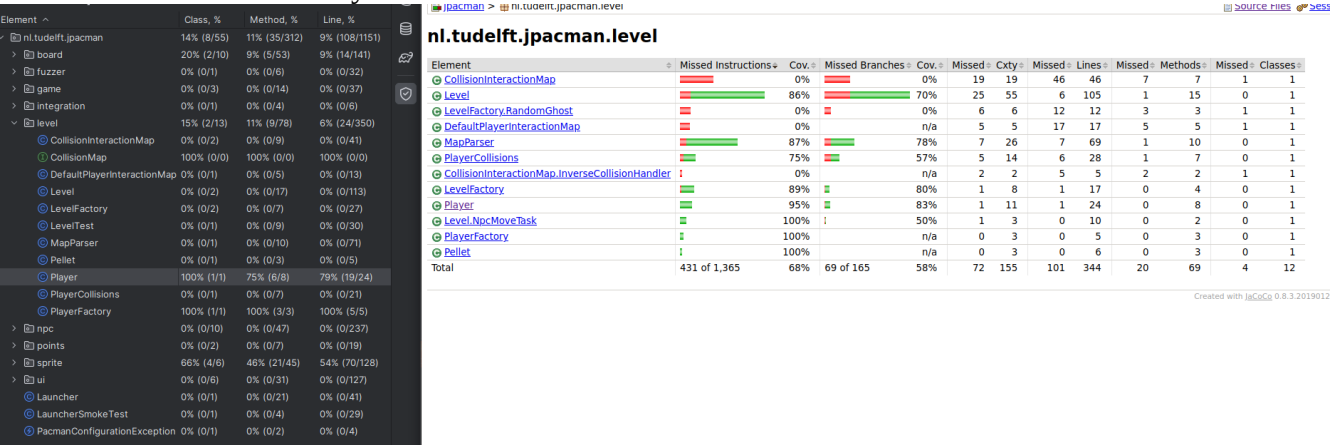
After implementing the above classes that would specifically test the Player functions of `setAlive()`, `getScore()`, and `addPoints()` in the code snippets below, the test coverage went up to 14%, 11% and 9%.

The reports that were retrieved from JaCoCo's tool revealed that while the



Element ^	Class, %	Method, %	Line, %
nl.tudelft.jpacman	14% (8/55)	11% (35/312)	9% (108/1151)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	15% (2/13)	11% (9/78)	6% (24/350)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	66% (4/6)	46% (21/45)	54% (70/128)
ui	0% (0/6)	0% (0/31)	0% (0/127)
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)

percentages matched for the level package, each individual method within Player did actually show slightly different percentages as is shown in the screenshot below, with a variety of reasons in that the tool is much more in depth than what is presented in the pop up window on the IDE. The visual representation of which methods are covered especially by their respective branches is really helpful and I think that I prefer the JaCoCo report as it is much easier to navigate and really get a sense of what's missing and where exactly I need to check, as well as being more thorough in checking several branches which is incredibly useful.



## Code Snippets:

This test suite evaluates the Player class's `setAlive` method in the JPacman game framework, ensuring the method accurately updates the player's alive status. It comprises two tests: `testPlayerSetAliveTrue` verifies that setting a player to alive correctly updates their status to alive and clears any killer information, while `testPlayerSetAliveFalse` checks that setting a player to not alive updates their status accordingly. The tests utilize a fresh Player instance for each case, established through a `BeforeEach` setup with the `PlayerFactory` and `PacManSprites`, ensuring isolation and accuracy of the test outcomes. This concise approach effectively confirms the `setAlive` method's functionality, critical for gameplay dynamics involving player life states.

```
public class PlayerScoreTest {
    2 usages
    private PlayerFactory factory;
    6 usages
    private Player thePlayer;

    @BeforeEach
    void setUp() {
        // Initialize the player factory with a sprite store
        PacManSprites spriteStore = new PacManSprites();
        factory = new PlayerFactory(spriteStore);

        // Create a new player instance before each test
        thePlayer = factory.createPacMan();
    }

    @Test
    void testInitialScoreIsZero() {
        // Test that the initial score of a new player is zero
        assertThat(thePlayer.getScore()).isEqualTo( expected: 0 );
    }

    @Test
    void testScoreAfterAddingPoints() {
        // Assuming there's a method to add points to the player's score
        int pointsToAdd = 50;
        thePlayer.addPoints(pointsToAdd);

        // Test that the player's score is correctly updated after adding points
        assertThat(thePlayer.getScore()).isEqualTo(pointsToAdd);

        // Adding more points and testing again
        int additionalPoints = 100;
        thePlayer.addPoints(additionalPoints);

        // The total score should now be the sum of the points added
        assertThat(thePlayer.getScore()).isEqualTo( expected: pointsToAdd + additionalPoints );
    }
}
```

This test suite focuses on verifying the scoring functionality within the JPacman game's Player class, specifically ensuring that the scoring system accurately tracks and updates the player's score. Initiated with a setup that creates a new player instance for each test, it first confirms that a new player starts with a score of zero. Subsequent tests then validate that the player's score correctly increases by specific amounts when points are added, both for a single addition and cumulatively for multiple additions. This series of tests, by asserting the initial score and the correct update of scores upon adding points, effectively ensures the integrity of the game's scoring mechanism, a crucial aspect of gameplay and player progress tracking.

```
1 package nl.tudelft.jpacman.level;
2
3 import nl.tudelft.jpacman.sprite.PacManSprites;
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Test;
6 import static org.assertj.core.api.Assertions.assertThat;
7
8 public class PlayerSetAliveTest {
9     1 usage
10     private static final PacManSprites SPRITE_STORE = new PacManSprites();
11     2 usages
12     private PlayerFactory factory;
13     3 usages
14     private Player thePlayer;
15
16     @BeforeEach
17     void setUp() {
18         factory = new PlayerFactory(SPRITE_STORE);
19         thePlayer = factory.createPacman();
20     }
21
22     @Test
23     void testPlayerSetAliveTrue() {
24         // Assuming the player is not alive initially for the sake of this test
25         thePlayer.setAlive(false); // Make sure the player is initially not alive
26         thePlayer.setAlive(true);
27         assertThat(thePlayer.isAlive()).isTrue();
28         assertThat(thePlayer.getKiller()).isNull();
29     }
30
31     @Test
32     void testPlayerSetAliveFalse() {
33         // Assuming the player is alive initially
34         thePlayer.setAlive(true); // Make sure the player is initially alive
35         thePlayer.setAlive(false);
36         assertThat(thePlayer.isAlive()).isFalse();
37         // Since setAlive(false) does not inherently set a killer, this remains unchanged/null if not previously set.
38         // Additional logic could be tested here if setAlive(false) had more observable effects.
39     }
40 }
```

## Task 4

Below I will have attached all of the code snippets along with a brief description of each, together they all add up to 100% coverage using nosetests.

```
new *
def test_repr(self):
    """Test the representation of an account"""
    account = Account()
    account.name = "Foo"
    self.assertEqual(str(account), "<Account 'Foo'>")
```

**test\_repr:** Verifies that the string representation (`__repr__`) of an Account object correctly formats as "`<Account 'Foo'>`" when the account's name is set to "Foo". This test ensures the representation method works as expected for display purposes.

```
def test_to_dict(self):
    """ Test account to dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    result = account.to_dict()
    self.assertEqual(account.name, result["name"])
    self.assertEqual(account.email, result["email"])
    self.assertEqual(account.phone_number, result["phone_number"])
    self.assertEqual(account.disabled, result["disabled"])
    self.assertEqual(account.date_joined, result["date_joined"])
```

**test\_to\_dict:** Tests that an Account object can be correctly converted to a dictionary (to\_dict method), with all relevant attributes (name, email, phone number, disabled status, and date joined) accurately reflected in the dictionary. It ensures data serialization integrity.

```
def test_from_dict(self):
    """Test setting account attributes from a dictionary"""
    # Prepare a dictionary with test data
    test_data = {
        "name": "Test User",
        "email": "test@example.com",
        "phone_number": "123-456-7890",
        "disabled": False,
        "date_joined": "2021-01-01"
    }

    # Create an empty Account object
    account = Account()

    # Use the from_dict method to set attributes
    account.from_dict(test_data)

    # Assert that attributes were correctly set
    self.assertEqual(account.name, test_data["name"])
    self.assertEqual(account.email, test_data["email"])
    self.assertEqual(account.phone_number, test_data["phone_number"])
    self.assertEqual(account.disabled, test_data["disabled"])
    self.assertEqual(account.date_joined, test_data["date_joined"])
```

**test\_from\_dict:** Validates the from\_dict method, which sets an Account object's attributes from a given dictionary. This test checks that all account attributes are properly updated from the dictionary values, ensuring the method accurately deserializes data into an account object.

```

def test_update_account(self):
    """Test updating an existing account in the database."""
    # Create and insert an account into the database
    data = ACCOUNT_DATA[self.rand] # Get a random account data
    account = Account(**data)
    account.create() # Save the new account to the database

    # Modify one of the account's attributes
    new_email = "updated@example.com"
    account.email = new_email

    # Update the account in the database
    account.update()

    # Retrieve the updated account from the database
    updated_account = Account.find(account.id)

    # Assert that the account was updated correctly
    self.assertEqual(updated_account.email, new_email)

```

**test\_update\_account:** Confirms that an existing account in the database can be updated correctly. It involves changing an account's attribute (email), saving the changes to the database, and then verifying that the updated attribute is correctly persisted. This test ensures the update method functions correctly for existing accounts.

```

def test_update_account_without_id_raises_error(self):
    """Test updating an account without an ID raises DataValidationError."""
    # Create an account instance without saving it to the database
    account = Account(name="Test Account", email="test@example.com")

    # Directly manipulating the id to None to simulate the scenario
    # where the account is not persisted in the database.
    account.id = None

    # Attempting to update the account should raise a DataValidationError
    with self.assertRaises(DataValidationError) as context:
        account.update()

    # Optionally, assert the message of the raised DataValidationError
    self.assertTrue("Update called with empty ID field" in str(context.exception))

```

**test\_update\_account\_without\_id\_raises\_error:** Checks that attempting to update an account without a set id raises a DataValidationError. This test is important for validating that the method enforces the presence of an id to avoid updating non-persisted accounts, ensuring data integrity.

```

def test_delete_account(self):
    """Test deleting an account removes it from the database."""
    # First, create and insert an account into the database
    data = {"name": "Delete Test", "email": "delete@example.com"}
    account = Account(**data)
    account.create() # Persist the account to the database

    # Ensure the account was created by checking if it's retrievable
    created_account = Account.find(account.id)
    self.assertIsNotNone(created_account)

    # Now, delete the account
    account.delete()

    # Attempt to retrieve the account again, expecting None since it should be deleted
    deleted_account = Account.find(account.id)
    self.assertIsNone(deleted_account)

```

**test\_delete\_account:** Ensures that the delete method removes an account from the database. After creating and verifying the existence of an account, it's deleted, and the test verifies that the account can no longer be retrieved, confirming that the deletion process works as intended.

## Task 5

For the first test function that I created was `test_update_a_counter(self)` which creates a counter, and then updates the counter and ensures whether or not there was a successful return code:

```
def test_update_a_counter(self):  
    """Test updating a counter increments its value"""  
    # Step 1: Create a counter  
    self.client.post('/counters/test_update_counter')  
  
    # Step 2: Update the counter and ensure successful return code  
    update_response = self.client.put('/counters/test_update_counter')  
    self.assertEqual(update_response.status_code, status.HTTP_200_OK)
```

With no function in `counter.py` that would achieve this, it resulted in a red error code:

```
Counter tests  
- It should create a counter  
- It should return an error for duplicates  
- Test updating a counter increments its value (FAILED)  
  
=====  
FAIL: Test updating a counter increments its value  
-----
```

Then once the actual update counter was added which processes HTTP PUT requests to increment the value of a specified counter by name via the `/counters/<name>` endpoint. If the counter exists, it's incremented by 1, and the function returns the updated value with a 200 OK status. If no counter with the given name exists, a 404 Not Found error and a corresponding message are returned.

```
@app.route('/counters/<name>', methods=["PUT"])
def update_counter(name):
    """Update a counter"""
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

The RED error was now a GREEN message:

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- Test updating a counter increments its value
```

The following part involved creating the test function `test_read_a_counter(self)` which verifies that a counter can be successfully created and then read back with the correct initial value. It first creates a counter named `test_read_counter` and then makes a GET request to retrieve this counter. The test checks two things: that the response status code is 200, indicating success, and that the returned counter value matches the expected initial value of 0:

```
def test_read_a_counter(self):
    """Test reading a counter returns the correct value"""
    # Step 1: Create a counter
    self.client.post('/counters/test_read_counter')

    # Step 2: Read the counter
    read_response = self.client.get('/counters/test_read_counter')
    self.assertEqual(read_response.status_code, 200)
    self.assertEqual(read_response.json, {'test_read_counter': 0})
```

Which resulted in the following RED error message:

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- Test reading a counter returns the correct value (FAILED)
- Test updating a counter increments its value
```

```
=====
FAIL: Test reading a counter returns the correct value
=====
```

Then after implementing the actual `get_counter` function below which handles GET requests to fetch the current value of a specified counter by its name. If the counter exists within the service, it returns the counter's name and its value with a 200 OK status, ensuring users can easily retrieve and view counter states. If the counter does not exist, the function responds with a 404 Not Found error and a message:

```
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Get a counter's value"""
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    return {name: COUNTERS[name]}
```

The error message went away and was now GREEN:

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- Test reading a counter returns the correct value
- Test updating a counter increments its value
```