Unit Testing Report

<u>Link to fork repository:</u> https://github.com/axelauda/SeniorDesign

Task 2.1:

Method 1: BoardFactory.createBoard()

Location: src/main/java/nl/tudelft/jpacman/board/BoardFactory

Test File: src/test/java/nl/tudelft/jpacman/board/CreateBoardTest.java

To test when a board is created, I needed to import a board factory that would create the board, while also initializing a 2D Square array called grid, filling it with a couple Square objects, and passing grid into the createBoard() method.

My test was for the proportions of the grid (its width and height) to match the proportions of the board created from it. To do that, I had two assert statements: The first would check if the board's width was the same as the grid's width. The second would check the height for both. If they are equal, then the board was created properly.

When the board is created, it also sets each square to know its neighboring squares, to the north, south, east, and west. To check this, I used the board.squareAt() function to get a square, and then the squares in each of the four cardinal directions, and then I check that with the middle square, that its neighbors (obtained by doing square.getSquareAt(Direction)) are the same as the the actual squares that it's surrounded by.

Method 2: Unit.leaveSquare()

Location: src/main/java/nl/tudelft/jpacman/board/Unit

Test File: src/test/java/nl/tudelft/jpacman/board/UnitLeaveSquareTest.java

For testing this method, I needed to check that when a unit leaves a square that it has actually left the square. To start, I began by placing a unit on a square using unit.occupy(). I then checked on the square's end if it's being occupied by the unit with sq.getOccupants(). Since .getOccupants() returns a list of units, and there is one unit currently occupying, then it should be the case that the .isEmpty() of that list is false, which I check with an assert.

After that, I called unit.leaveSquare(), which should remove the unit from the square's occupant list. I then checked using .isEmpty() again, since when I used leaveSquare(), the occupant list should now be empty since it only contained one unit. Therefore, if .isEmpty() was true, then the test passed.

```
package nl.tudelft.jpacman.board;
     import static org.assertj.core.api.Assertions.assertThat;
     import org.junit.jupiter.api.Test;
8 ♥ public class UnitLeaveSquareTest {
         Square sq = new BasicSquare();
         Unit unit = new BasicUnit();
         void leftSquare() {
             unit.occupy(sq);
             List<Unit> occupants = sq.getOccupants();
             assertThat(occupants.isEmpty()).isEqualTo( expected: false);
             unit.leaveSquare();
             occupants = sq.getOccupants();
             assertThat(occupants.isEmpty()).isEqualTo( expected: true);
```

Method 3: PlayerCollisions.PlayerVersusGhost()

Location: src/main/java/nl/tudelft/jpacman/level/PlayerCollisions

Test File: src/test/java/nl/tudelft/jpacman/level/PlayerVersusGhostTest.java

For testing .PlayerVersusGhost() in the PlayerCollisions class, there were three areas I needed to check for the test to pass. That is, when a player and ghost collide:

- (1). The score of the player stays the same.
- (2). The player is killed.
- (3). The player was killed by the ghost and not some other unit.

To check (1), since I had just created Player, its score should have been 0, so as long as after the player and ghost collide the Player.getScore() stays 0, then it passes.

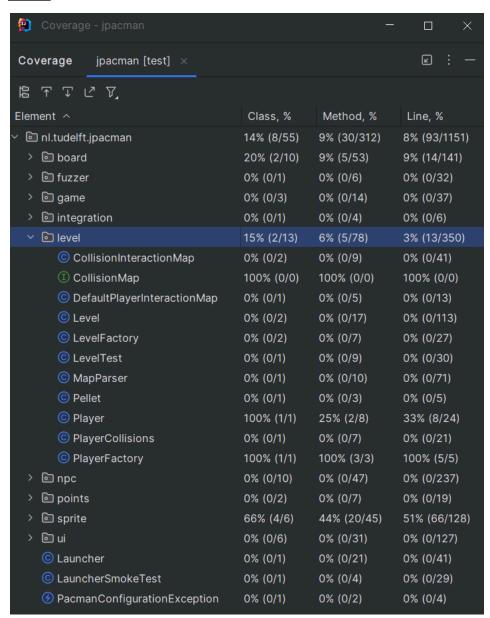
To check (2), I just called Player.isAlive() and checked if it's false, meaning the player is dead.

To check (3), if Player.getKiller() is the ghost that it collided with, then it passes the last check.

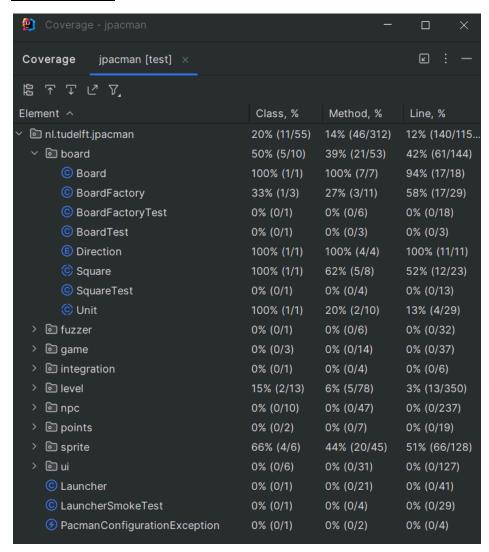
```
package nl.tudelft.jpacman.level;
    > import ...
      public class PlayerVersusGhostTest {
          private final PacManSprites sprites = new PacManSprites();
          private final PlayerFactory playerFactory = new PlayerFactory(sprites);
          private final GhostFactory ghostFactory = new GhostFactory(sprites);
          private final PointCalculator calculator = new DefaultPointCalculator();
          private final PlayerCollisions collisions = new PlayerCollisions(calculator);
          @Test
25 🗞
          void testPlayer() {
              Player player = playerFactory.createPacMan();
              Ghost ghost = ghostFactory.createBlinky();
              collisions.playerVersusGhost(player, ghost);
              assertThat(player.getScore()).isEqualTo( expected: 0);
              assertThat(player.isAlive()).isEqualTo( expected: false);
              assertThat(player.getKiller()).isEqualTo(ghost);
```

Coverage:

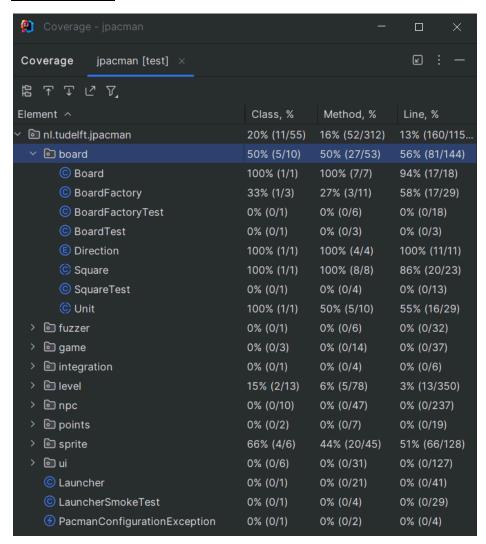
Initial:



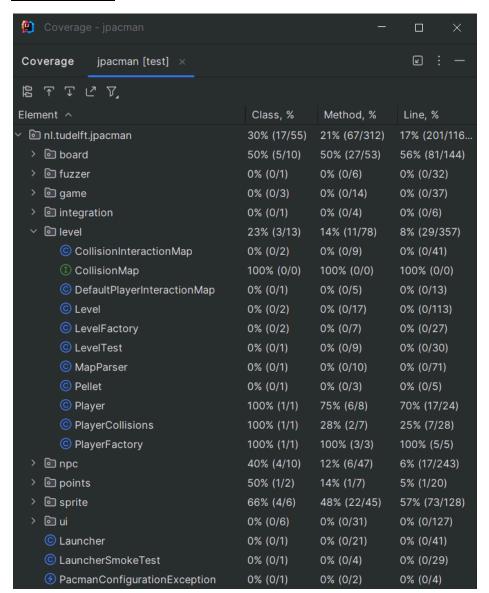
After Method 1:



After Method 2:



After Method 3:



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?

No, the coverage results are sometimes more or less than the coverage of my own tests. For example, the coverage of Board on JaCoCo is 81%, while IntelliJ says that I have 100% class and method coverage. However, I missed one line on JaCoCo in Board, which does correspond and equal the 94% (17/18) line coverage in IntelliJ. Another example is BoardFactory, which in JaCoCo has 94% coverage, but in IntelliJ I have 33% class coverage and 27% method coverage. I believe that JaCoCo is more accurate and determines coverage line by line to provide a total coverage, while in IntelliJ it checks if at least one line has been tested in a class or method to determine if it's been "covered." I also believe that a covered line has a different definition in JaCoCo compared to IntelliJ, with JaCoCo being more lenient on what's considered a line covered, which explains the differing results from time to time.

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

Yes, the way that the code section that isn't being covered is highlighted in yellow makes it easy to see where tests should be made to fix the coverage of the uncovered branch.

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

I prefer JaCoCo's report over IntelliJ's, since JaCoCo highlights code for you to see where coverage is or isn't being done. It also includes branch coverage, which IntelliJ doesn't have, and JaCoCo tells you how many Methods and Lines you've missed which is a bit clearer to me than the fraction and percentage representation in IntelliJ.

Task 4:

```
(venv) C:\Users\alexl\test_coverage>nosetests -s
Test Account Model
 Test creating multiple Accounts
 Test Account creation using known data
 Test account deletion
 Test the representation of an account
 Test updating an existing Account
 Test updating an account with an empty ID
Name
                    Stmts
                            Miss Cover
                                         Missing
models\ init .py
                               0
                                   100%
models\account.py
                       40
                                   100%
TOTAL
                       47
                                   100%
                               0
Ran 8 tests in 1.232s
```

I achieved 100% coverage after implementing the following tests:

```
def test_from_dict(self):
    """ Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account

# Create an account from data dictionary
    account = Account()
    account.from_dict(data)

# Check that attributes are the same as dictionary
    self.assertEqual(account.name, data["name"])
    self.assertEqual(account.email, data["email"])
    self.assertEqual(account.phone_number, data["phone_number"])
    self.assertEqual(account.disabled, data["disabled"])
```

```
def test update account(self):
    """ Test updating an existing Account """
   data = ACCOUNT_DATA[self.rand]
   account = Account(**data)
   account.create()
   updated_name = "Updated Name"
   account.name = updated_name
   account.update()
   updated_account = Account.find(account.id)
   self.assertEqual(updated_account.name, updated_name)
def test update account with empty id(self):
   """ Test updating an account with an empty ID """
   data = ACCOUNT_DATA[self.rand]
   account = Account(**data)
   with self.assertRaises(DataValidationError):
       account.update()
   self.assertEqual(len(Account.all()), 0)
```

```
def test_delete(self):
    """ Test account deletion """
    # Create an account and save it to the database
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()

# Delete it and check if it was removed from the database
    account.delete()
    self.assertEqual(len(Account.all()), 0)
```

Task 5:

test update a counter(self):

```
def test_update_a_counter(self):
    result = self.client.post('/counters/baz')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    self.assertEqual(COUNTERS['baz'], 0)

    result = self.client.put('/counters/baz')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(COUNTERS['baz'], 1)
```

In the update counter test, I first created a new counter, and checked if the counter was initialized to 0. When I ran nosetests, it was GREEN, since creating a counter was previously tested and properly coded.

After creating the counter, I called the put method, which in counter.py would increment the counter by 1. I then checked if the status was OK, and that the counter was equal to 1. This gave me a RED in nosetests, and an AssertionError: 405 != 200, meaning that the return status of the put method wasn't OK but METHOD_NOT_ALLOWED.

This was because I now needed to implement update counter() in counter.py:

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    global COUNTERS

COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

In update_counter(), I REFACTORED the code by incrementing the count of an existing counter by 1, then returning the name and the status OK.

Now when I ran the nosetests, I got GREEN, since now the previous assertion that failed is now 200=200. Lastly, the assertion self.assertEqual(COUNTERS[name],1), passes meaning that the put method was successful and properly covered.

test read a counter(self):

```
def test_read_a_counter(self):
    result = self.client.post('/counters/boo')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    result = self.client.get('/counters/boo')
    self.assertEqual(result.status_code, status.HTTP_200_OK)

    result = self.client.get('/counters/bah')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

In the read counter test, we check that when reading a counter, that either it exists and the status is OK, or that it doesn't exist and it returns the status NOT_FOUND. I first start by creating a counter "boo", and assert check that it was created.

If that passes, I call the get method for "boo", which should return status OK since boo exists as a counter in COUNTERS.

Next, I try to read a counter called "bah", which was never a counter that I created. When I try to make an assertion after doing self.client.get('/counters/bah') that status_code = OK, I get RED, 404 != 200.

This is because inside the read counter() method that I REFACTORED in counter.py:

```
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    """Update a counter"""
    global COUNTERS

    if name not in COUNTERS:
        return {"Message":f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

It checks if the name that's passed to the get method exists as a counter in COUNTERS. If it doesn't, then it returns the status that it couldn't find the counter as 404_NOT_FOUND. When I ran the nosetests after fixing the last assertion so status_code = NOT_FOUND, I get GREEN; since reading the "boo" counter has status OK, and reading the "bah" counter returns status NOT_FOUND, the read_counter() get method has been properly covered.

```
(venv) C:\Users\alex1\tdd>nosetests -s
Counter tests
 It should create a counter
 It should return an error for duplicates
update a counter
                        Miss Cover
                                      Missing
Name
                Stmts
src\counter.py
                               100%
                    20
                           0
src\status.py
                           0
                               100%
                    6
TOTAL
                   26
                           0
                               100%
Ran 4 tests in 0.406s
OK
```

After writing the counter methods based on the test methods, I was sure to achieve 100% coverage.