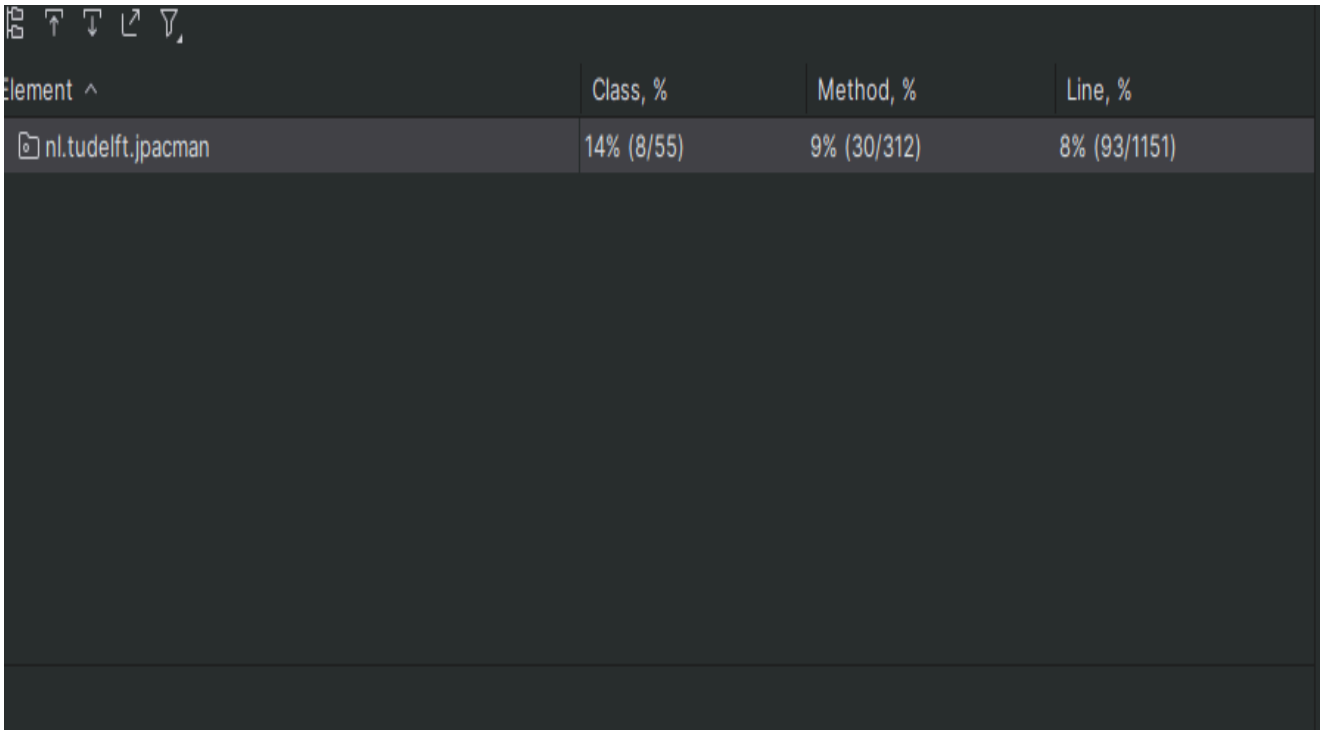
Forked Repo:
https://github.com/Ramirez-Christopher/SeniorDesign

JPACMAN

In Jpacman, the initial test coverage is 14% for classes, 9% for method, and 8% for line.



I have created tests for 2 methods in the Pellet Class and 1 in the Board Class:

The methods I tested in the pellet class are:

Pellet(image):

Pellet is the constructor for the Pellet class. It accepts an image for the sprite and the amount of points it's worth if even by pacman. The test consisted of instantiating a new Pellet Object and giving a point value and an image. Next, I assert that the values returned by the getters are the same as the values used to instantiate the object.

```
@Test
void testConstructor() {
    Pellet testPellet = new Pellet( points: 1,image);
    assertThat( actual: testPellet.getValue() == 1);
    assertThat( actual: testPellet.getSprite() == image);

}
```

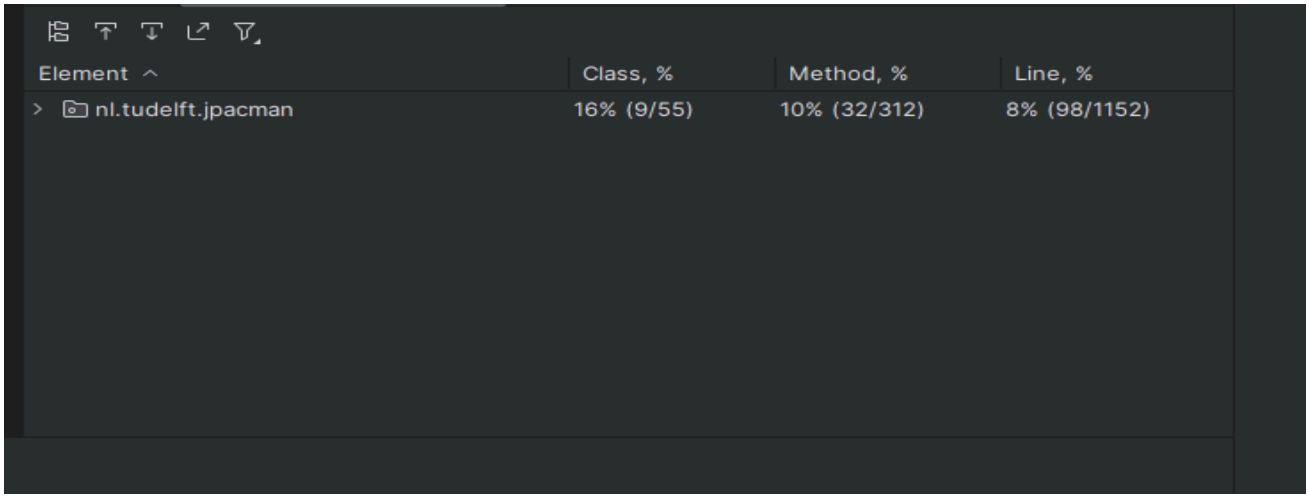
However, before that , I needed to test the getSprite() method. So the next method I created a unit test for was:

getSprite() :

getSprite returns the image of a pellet to whatever procedure that called it. That test consisted of testing whether or not the value returned by getSprite is the same image that was used to instantiate the Pellet object.

```
@Test
void testSprite() {

    assertThat(pellet.getSprite()).isEqualTo(image);
}
```

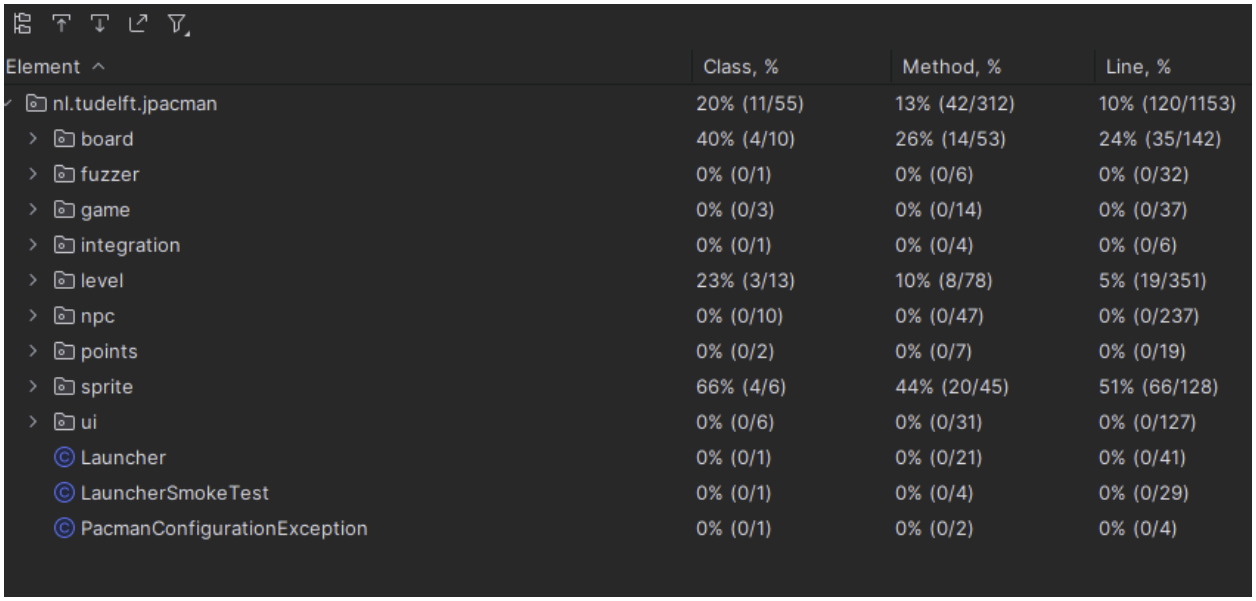
After writing these two unit tests the code coverage went up to 16% for class and 10% for method.



For the Board Class, the method I chose to test for was withinBoard - a function used to assure that a given coordinate was within the boundaries of a given grid/board. This was more complicated than the last two methods I tested for. The test consisted of instantiating a Square array and iterating through it; making every element a new basic square. Then, I instantiate a new board with the square array and assert the values I specify are within the number of elements of the square array.

```java
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;


public class WithinBordersTest {

    1 usage
    private static final int x = 4;
    1 usage
    private static final int y = 4;

    1 usage
    private static Square[][] gridCreator(int x, int y) {

        Square[][] array = new Square[x][y];

        for (int i = 0; i < x; i++) {
            for (int j = 0; j < y; j++) {
                array[i][j] = new BasicSquare();
            }
        }

        return array;
    }

    1 usage
    Square[][] grid = gridCreator(x,y);

    1 usage
    private final Board board = new Board(grid);


    @Test
    void withinBoardTest() {
        assertThat(board.withinBorders( x: 2, y: 3)).isTrue();
    }


}
```

The code coverage after the WithinBordersTest was 20%.



Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	20% (11/55)	13% (42/312)	10% (120/1153)
board	40% (4/10)	26% (14/53)	24% (35/142)
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	23% (3/13)	10% (8/78)	5% (19/351)
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)

JACOCO

Using Jacoco, I was able to understand the test coverage even more:

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	74	155	104	344	21	69	4	12
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
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,213 of 4,694	74%	293 of 637	54%	293	590	229	1,039	51	268	6	47

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

It's similar to the Intellij coverage but it's different due to it reporting the missing branch coverage.

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

Yes it's helpful because it shows that new unit tests need to be implemented in order to address all the edge cases, as well as the missing code coverage in the branches.

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

I prefer IntelliJ because it's more straightforward. However, I acknowledge that Jacoco is more informative and would be more useful in the long run.

Unit Testing Report

The initial state of the repository was 77% code coverage (gathered by nosetests). After writing the unit tests for the 3 missing account functions I was able to bring it back to 100%.

```
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models/__init__.py       7      0   100%
models/account.py       40      0   100%
--------------------------------------------------
TOTAL                   47      0   100%
----------------------------------------------------------------------
Ran 8 tests in 0.409s
```

The remaining functions were:

Account.From_dict(): Responsible for initializing an account using a dictionary that's passed on to it.

```
def test_from_dict(self):
    """Test account from dict"""
    obj = Account()
    obj.from_dict({'key1': 'value1', 'key2': 'value2'})
    self.assertEqual(obj.key1, 'value1')
    self.assertEqual(obj.key2, 'value2')
```

Account.update(): Responsible for updating an account to a database.

```
def test_update_account(self):
    """ Test updating an Account """
    data = ACCOUNT_DATA[self.rand]  # get a random account
    account = Account(**data)
    account.create()
    account.update()

    retrievedAccount = account.find(account.id)

    self.assertEqual(retrievedAccount, account)
```

Account.delete(): Removes an instance of Account from the database.

```
def test_delete_account(self):
    """ Test deleting an Account """
    data = ACCOUNT_DATA[self.rand]  # get a random account
    account = Account(**data)
    account.create()
    account.update()

    account.delete()

    self.assertIsNone(Account.find(account.id))
```

TDD

After following the report(creating create_counter()), I started creating the tests for update_counter() and read_counter().

I first created the unit test for the update method, which put me in the RED phase.

```python
def test_update_a_counter(self):

    result = self.client.post('/counters/test')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    updateResult = self.client.put('/counters/test')
    self.assertEqual(updateResult.status_code, status.HTTP_200_OK)

    self.assertEqual(result.get_json()['test'] + 1, updateResult.get_json()['test'])
```

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- update a counter (FAILED)

======================================================================
FAIL: test_update_a_counter (test_counter.CounterTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/skript/cs472/tdd/tests/test_counter.py", line 44, in test_update_a_counter
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
AssertionError: 405 != 404
```

Afterwards, I started writing the definition of the update_counter() method.

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

    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS

    if not name 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
```

In order to get into the REFACTOR stage, we have to make sure that the counter exist, so I specify the following:

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

    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS

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

I also make sure that when a put request is done on a non existent counter, it should return a 404 status code.

```python
    nullCounter = self.client.put('/counters/NULL')
    self.assertEqual(nullCounter.status_code, status.HTTP_404_NOT_FOUND)
```

Thus, After writing the test case, then the definition of the method, and finally the Specification of making sure that the counter does exist, we finally reach the GREEN stage.

```
(base) skript@skript:~/cs472/tdd$ nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- update a counter

Name             Stmts   Miss  Cover   Missing
-------------------------------------------------
src/counter.py      18      1    94%   32
src/status.py        6      0   100%
-------------------------------------------------
TOTAL               24      1    96%
----------------------------------------------------------------------
Ran 3 tests in 0.078s

OK
```

Moving to the read_counters function, I start by writing the test case first in order to follow the TDD outline. The test case is the following alongside the RED stage message.

```python
def test_read_counter(self):

    result = self.client.post('/counters/newTest')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    for x in range(5):
        self.client.put('/counters/newTest')

    num = self.client.get('/counters/newTest')
    self.assertEqual(num.get_json()['newTest'], 5)
```

```
(base) skript@skript:~/cs472/tdd$ nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- read counter (ERROR)
- update a counter

======================================================================
```

I then move on to writing the definition of read_counter(), which allows me to move on to the GREEN phase:

```python
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):

    """Read  a counter"""
    app.logger.info(f"Request to read counter: {name}")

    global COUNTERS

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

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

```
(base) skript@skript:~/cs472/tdd$ nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- read counter
- update a counter

Name               Stmts   Miss  Cover   Missing
------------------------------------------------
src/counter.py        24      2    92%   32, 46
src/status.py          6      0   100%
------------------------------------------------
TOTAL                 30      2    93%
----------------------------------------------------------------------
Ran 4 tests in 0.084s

OK
```

Afterwards, I went on to the REACTOR stage. To make sure a 404 status code would be returned in the case of a get request getting put on a non existent counter, I added the following to my test unit code:

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

Finally I reach the GREEN stage once more with 100% test coverage:

```
Counter tests
- It should create a counter
- It should return an error for duplicates
- read counter
- update a counter

Name             Stmts   Miss  Cover   Missing
------------------------------------------------
src/counter.py      24      0   100%
src/status.py        6      0   100%
------------------------------------------------
TOTAL               30      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.083s

OK
```