

CS 472

2/3/24

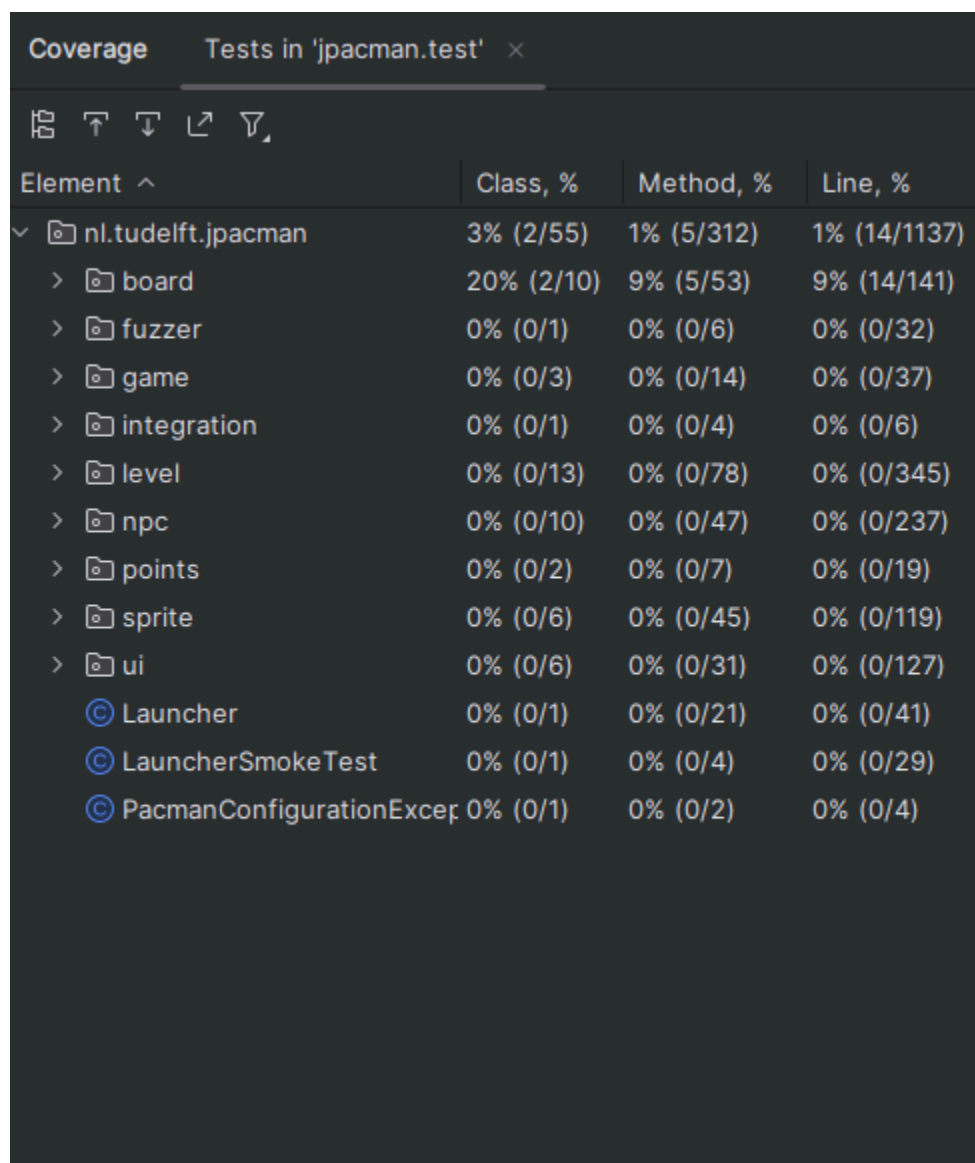
Prof. Businge

Software Testing Report

Fork Repo: <https://github.com/Grimmblood/CS472-Project>

Task 1

Task 1 was purely foundational, ensuring that we could view the coverage report. After finally finding the right combination of settings, I was able to start the assignment. Here is the initial coverage report.



The screenshot shows a coverage report interface with a dark theme. At the top, there's a tab labeled 'Coverage' and another labeled 'Tests in 'jpacman.test' x'. Below the tabs are several icons for navigation and filtering. The main part of the interface is a table with four columns: 'Element ^', 'Class, %', 'Method, %', and 'Line, %'. The table lists various code elements and their coverage statistics.

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)

Task 2

For task 2, we were given the task of increasing the testing coverage on JPacman. We were to accomplish this by writing a simple unit test to check if the player was alive, and then identify at least three methods in JPacman, and write unit tests for those.

Task 2.1.1 - getWidth()

For my first method, I decided to write a unit test for getWidth(). This was a simple method that retrieved the number of squares that made up the width of the game board. Admittedly, this was a fairly simple method, but I did have to create a BoardFactory in order to create the board in order to test the width of the board.

```
public class BoardTest {

    private static final PacManSprites SPRITES = new PacManSprites();

    private BoardFactory Factory = new BoardFactory(SPRITES);

    private Square [][] testGrid = new Square[5][5];

    private Board TheBoard;

    @Test
    void testWidth() {
        for(int i = 0; i < 5; i++)
        {
            for(int j = 0; j < 5; j++) {

                testGrid[i][j] = new Square() {

                    @Override
                    public boolean isAccessibleTo(Unit unit) { return false; }

                    @Override
                    public Sprite getSprite() { return null; }

                };
            }
        }
        TheBoard = Factory.createBoard(testGrid);
        assertEquals(TheBoard.getWidth(), 5);
    }
}
```

Task 2.1.2 - getHeight()

For my second method, I wrote a test for getHeight(). Similar to getWidth(), it returned the number of squares that made up the width of the game board. I also was able to implement it in the same BoardTest class from getWidth(), so I didn't have to reimplement the BoardFactory.

```
@Test
void testHeight() {
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 5; j++) {

            testGrid[i][j] = new Square() {

                @Override
                public boolean isAccessibleTo(Unit unit) { return false; }

                @Override
                public Sprite getSprite() { return null; }
            };
        }
    }
    TheBoard = Factory.createBoard(testGrid);
    assertThat(TheBoard.getHeight()).isEqualTo(5);
}
```

Task 2.1.3 - hasSquare()

For my final method, I decided to write a test for the unit's hasSquare() method, which simply returns true or false depending on if the unit occupies a square on the board. I had to do everything I had to do in the prior methods, since there needed to be a unit standing on a square on the board.

```

public class UnitTest {

    private static final PacManSprites SPRITES = new PacManSprites();

    private PlayerFactory Factory = new PlayerFactory(SPRITES);

    private BoardFactory BFactory = new BoardFactory(SPRITES);

    private Square [][] testGrid = new Square[5][5];

    private Player ThePlayer = Factory.createPacMan();

    private Board TheBoard;

    @Test
    void testHasSquare() {
        for(int i = 0; i < 5; i++)
        {
            for(int j = 0; j < 5; j++) {

                testGrid[i][j] = new Square() {

                    @Override
                    public boolean isAccessibleTo(Unit unit) { return false; }

                    @Override
                    public Sprite getSprite() { return null; }

                };
            }
        }
        TheBoard = BFactory.createBoard(testGrid);
        ThePlayer.occupy(TheBoard.squareAt(2, 3));
        assertThat(ThePlayer.hasSquare()).isEqualTo(true);
    }
}

```

Task 3

Task 3 was basically just introducing us to JaCoCo, which is another coverage utility that we can use to test our code. This one, unlike IntelliJ, opens in your browser.

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

A: The coverage results from JaCoCo are not quite the same as the ones from IntelliJ, as the numbers are different. For example, it says that the Board package has 53 methods on IntelliJ with 52% method coverage, yet JaCoCo seems to be telling me that there are only 40 methods in the Boards package, and that I missed 0. I am probably not reading it right, but that is what it seems to be telling me.

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

A: I found JaCoCo's branch visualization to be helpful, as it would highlight the one line in the middle of the method that I thought I had tested.

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

A: I honestly like looking at the IntelliJ coverage window more, mostly because it's convenient. Additionally, because of the coverage number mismatch between the two, I kind of like the vagueness of the IntelliJ coverage report. There also is a line highlighter that will tell me which lines aren't getting tested, so I can see that without having to change windows, as well.

Task 4

Starting Task 4, after the instructional steps, where we created the tests for `__repr__()` and `to_dict()`, my coverage looked like this:

Name	Stmts	Miss	Cover	Missing
models__init__.py	7	0	100%	
models\account.py	40	11	72%	34-35, 45-48, 52-54, 74-75
TOTAL	47	11	77%	

Ran 4 tests in 1.140s

Then, I set out to write the tests in an effort to get coverage of the Python Testing Lab up to 100%. I wrote tests for the `from_dict()`, `update()`, `delete()`, and `find()` methods.

```
def test_from_dict(self):
    """ Test account from dict """
    test = self.rand
    account = Account()
    account.from_dict(ACCOUNT_DATA[test])
    self.assertEqual(account.name, ACCOUNT_DATA[test]["name"])
    self.assertEqual(account.email, ACCOUNT_DATA[test]["email"])
    self.assertEqual(account.phone_number, ACCOUNT_DATA[test]["phone_number"])
    self.assertEqual(account.disabled, ACCOUNT_DATA[test]["disabled"])
```

```
def test_update(self):
    """ Tests updating an account from the database """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.id = 2
    account.update()
```

```
def test_delete(self):
    """ Tests deleting an account from the database """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.delete()
```

```
def test_find(self):
    """ Tests finding an account in the database """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.id = 345
    self.assertEqual(account.find(345), account)
```

After all was said and done, I ended up with 98% coverage. The last line, 47, is from the update() method, and is “raise DataValidationError("Update called with empty ID field")”. It seems like that line exists solely to prevent update from being called in id is empty, and all my attempts to recreate that situation resulted in line 48 being missed, resulting in the same coverage of 98%.

Name	Stmts	Miss	Cover	Missing
models__init__.py	7	0	100%	
models\account.py	40	1	98%	47
TOTAL	47	1	98%	

Ran 8 tests in 1.767s

OK

Task 5

For Task 5, we were instructed to use Test Driven Development practices to implement tests that fail, then implement methods that pass, then add more to the tests so they fail again, and so on until we are done.

We started off by writing the test for creating a counter using REST API.

```
def test_create_a_counter(self):
    """ Should create a counter """
    client = app.test_client()
    result = client.post('/counters/foo')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
```

Of course, when we ran nosetests it failed because there was no method to create a counter. So we had to add one. The method creates a counter, returning the counter and a status code. The test method that we wrote earlier checks to make sure that the status code that is returned is the correct one.

```

from flask import Flask
from src import status

app = Flask(__name__)

# We will use the app decorator and create a route called slash counters.
# specify the variable in route <name>
# let Flask know that the only methods that is allowed to be called
# on this function is "POST".

COUNTERS = {}

@app.route(rule: '/counters/<name>', methods=['POST'])
def create_counter(name):
    """ Create a counter """
    app.logger.info(f'Request to create counter {name}')
    global COUNTERS
    if name in COUNTERS:
        return {'Message': f'Counter {name} already exists'}, status.HTTP_409_CONFLICT
    COUNTERS[name] = 0
    return {name: COUNTERS[name]}, status.HTTP_201_CREATED

```

Of course, that is the final version of the method. At the time, we were instructed to create a test for handling duplicate counters, and I forgot to take pictures as I was making things. If you create a duplicate counter, there is a conflict, and so we need to test for that.

```

def test_duplicate_counter(self):
    """ It should return an error for duplicates """
    result = self.client.post('/counters/bar')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = self.client.post('/counters/bar')
    self.assertEqual(result.status_code, status.HTTP_409_CONFLICT)

```

Of course, what use is a counter if it can't count? So we need to make a test for that. If the counter doesn't exist, we obviously can't update the counter, so we need to test that first. The last line is commented out because I could not figure out how to formally check the values. However, when they would print out, I could visually see that the values were always only one apart.


```

def test_update_counter(self):
    result = self.client.put('/counters/testCounter')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

    # Create a counter
    result = self.client.post('/counters/testCounter')

    # Ensure that it returned a successful return code
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    # check the counter value as a baseline
    result_base = result.get_data(True)
    # call to update the counter I just created
    result = self.client.put('/counters/testCounter')

    # ensure that it returned a successful return code
    self.assertEqual(result.status_code, status.HTTP_200_OK)

    # check that the counter value is one more than the baseline measured earlier
    # self.assertEqual(result_base, result.get_data(True))

```

Unfortunately, I couldn't figure out how to test the status code '204_NO_CONTENT', as I couldn't really figure out what that code was describing.

```

25 @app.route(rule: '/counters/<name>', methods=['PUT'])
26 def update_counter(name):
27     """ Update a counter """
28     app.logger.info(f'Request to update counter {name}')
29     global COUNTERS
30     if name not in COUNTERS:
31         return {'Message': f'Counter {name} not found'}, status.HTTP_404_NOT_FOUND
32     if name in COUNTERS:
33         if COUNTERS[name] is None:
34             return {name: COUNTERS[name]}, status.HTTP_204_NO_CONTENT
35         COUNTERS[name] = COUNTERS[name] + 1
36         return {name: COUNTERS[name]}, status.HTTP_200_OK

```

The final method that we needed to test was the ability to read a counter. I think this ties in to my issue from a couple pages ago, but I could not really figure out how to get the value of the counter formally. However, I was able to confirm that we are able to read the value, and of course, if the counter does not exist, then we aren't able to read it.

```

def test_read_counter(self):
    result = self.client.get('/counters/testReadCounter')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

    # create a counter
    result = self.client.post('/counters/testReadCounter')

    # ensure successful error code
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    # update the counter
    result = self.client.put('/counters/testReadCounter')

    # check the counter value
    result = self.client.get('/counters/testReadCounter')

    # ensure a successful error code
    self.assertEqual(result.status_code, status.HTTP_200_OK)

```

```

@app.route(rule: '/counters/<name>', methods=['GET'])
def read_counter(name):
    app.logger.info(f'Request to read counter {name}')
    global COUNTERS
    if name not in COUNTERS:
        return {'Message': f'Counter {name} not found'}, status.HTTP_404_NOT_FOUND
    if name in COUNTERS:
        return {name: COUNTERS[name]}, status.HTTP_200_OK

```