

Unit Testing Report









Task 2.1

Before Unit Tests

Here is the package that I was conducting all my unit tests on. As you can see, all classes have no testing completed.

I would be testing three functions within the AnimatedSprite Class:

- getPacManDeathAnimation()
- getGhostSprite()
- getWallSprite()

▼  sprite	0% (0/6)	0% (0/45)	0% (0/119)
 AnimatedSprite	0% (0/1)	0% (0/11)	0% (0/38)
 EmptySprite	0% (0/1)	0% (0/4)	0% (0/4)
 ImageSprite	0% (0/1)	0% (0/7)	0% (0/15)
 PacManSprites	0% (0/1)	0% (0/9)	0% (0/24)
 Sprite	100% (0/0)	100% (0/0)	100% (0/0)
 SpriteStore	0% (0/1)	0% (0/5)	0% (0/22)
 SpriteTest	0% (0/1)	0% (0/9)	0% (0/16)

getPacManDeathAnimation()

Here I would simply create a new PacManSprites object called sprites. Then, I make an AnimatedSprite object called death which is set to the value of sprites.getPacManDeathAnimation(). Afterwards, I use the assertNotNull() function to ensure that the AnimatedSprite object, death, was indeed not a null.

```
package nl.tudelft.jpacman.sprite;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class PacManDeathAnimationTest {

    @Test
    public void testGetPacManDeathAnimation() {
        PacManSprites sprites = new PacManSprites();
        AnimatedSprite death = sprites.getPacManDeathAnimation();

        assertNotNull(death, message: "The pacman death animation should not be null");
    }
}
```

▼ sprite	66% (4/6)	40% (18/45)	44% (57/128)
© AnimatedSprite	100% (1/1)	36% (4/11)	34% (15/44)
© EmptySprite	0% (0/1)	0% (0/4)	0% (0/4)
© ImageSprite	100% (1/1)	85% (6/7)	76% (13/17)
© PacManSprites	100% (1/1)	33% (3/9)	29% (7/24)
🟢 Sprite	100% (0/0)	100% (0/0)	100% (0/0)
© SpriteStore	100% (1/1)	100% (5/5)	95% (22/23)
© SpriteTest	0% (0/1)	0% (0/9)	0% (0/16)

getGhostSprite()

Here I would simply create a new PacManSprites object called sprites. Then, I make four GhostColor objects that correspond with the four available ghost colors. Afterwards, make four Maps objects that would be set to each ghost sprite color respectively. Lastly, I use four assertNotNull() functions to ensure that the Map objects did not return a null.

```
package nl.tudelft.jpacman.sprite;

import nl.tudelft.jpacman.board.Direction;
import nl.tudelft.jpacman.npc.ghost.GhostColor;
import org.junit.jupiter.api.Test;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.*;

public class GhostSpriteTest {

    @Test
    public void testGetGhostSprite() {
        PacManSprites sprites = new PacManSprites();

        GhostColor colorRed = GhostColor.RED;
        GhostColor colorOrange = GhostColor.ORANGE;
        GhostColor colorPink = GhostColor.PINK;
        GhostColor colorCyan = GhostColor.CYAN;

        Map<Direction, Sprite> ghostSpriteMap1 = sprites.getGhostSprite(colorRed);
        Map<Direction, Sprite> ghostSpriteMap2 = sprites.getGhostSprite(colorOrange);
        Map<Direction, Sprite> ghostSpriteMap3 = sprites.getGhostSprite(colorPink);
        Map<Direction, Sprite> ghostSpriteMap4 = sprites.getGhostSprite(colorCyan);

        assertNotNull(ghostSpriteMap1, message: "Ghost sprite map should not be null");
        assertNotNull(ghostSpriteMap2, message: "Ghost sprite map should not be null");
        assertNotNull(ghostSpriteMap3, message: "Ghost sprite map should not be null");
        assertNotNull(ghostSpriteMap4, message: "Ghost sprite map should not be null");
    }
}
```

✓ sprite	66% (4/6)	44% (20/45)	53% (68/128)
🕒 AnimatedSprite	100% (1/1)	36% (4/11)	34% (15/44)
🕒 EmptySprite	0% (0/1)	0% (0/4)	0% (0/4)
🕒 ImageSprite	100% (1/1)	85% (6/7)	76% (13/17)
🕒 PacManSprites	100% (1/1)	55% (5/9)	75% (18/24)
🕒 Sprite	100% (0/0)	100% (0/0)	100% (0/0)
🕒 SpriteStore	100% (1/1)	100% (5/5)	95% (22/23)
🕒 SpriteTest	0% (0/1)	0% (0/9)	0% (0/16)

getWallSprite()

Here I would simply create a new PacManSprites object called sprites. Then, I make a Sprite object called wall which is set to the value of sprite.getWallSprite(). Afterwards, I use the assertNotNull() function to ensure that the Sprite object, wall, was not a null.

```
package nl.tudelft.jpacman.sprite;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class GetWallSpriteTest {

    @Test
    public void testGetWallSprite() {
        PacManSprites sprite = new PacManSprites();

        Sprite wall = sprite.getWallSprite();

        assertNotNull(wall, message: "Wall sprite should not be null");
    }
}
```

▼ sprite	66% (4/6)	46% (21/45)	53% (69/128)
Ⓢ AnimatedSprite	100% (1/1)	36% (4/11)	34% (15/44)
Ⓢ EmptySprite	0% (0/1)	0% (0/4)	0% (0/4)
Ⓢ ImageSprite	100% (1/1)	85% (6/7)	76% (13/17)
Ⓢ PacManSprites	100% (1/1)	66% (6/9)	79% (19/24)
ⓘ Sprite	100% (0/0)	100% (0/0)	100% (0/0)
Ⓢ SpriteStore	100% (1/1)	100% (5/5)	95% (22/23)
Ⓢ SpriteTest	0% (0/1)	0% (0/9)	0% (0/16)

Task 3

After reviewing both of the coverage results from JaCoCo and IntelliJ, it appears that JaCoCo's results provide more information than the results from IntelliJ. IntelliJ's coverage results have four columns of information, while JaCoCo has eight columns of information. The four columns that both coverage results, (Class, Lines, Methods, Element), share are somewhat similar. However, upon further inspection, it seems that the two reports greatly differ in the "Methods" column and the "Lines" column. According to JaCoCo, the only method I missed in the AnimatedSprite Class was the split() method. Whereas, IntelliJ claims that I have missed 7 methods.

Using JaCoCo's source code visualizations on uncovered branches was very helpful. With JaCoCo's visualization features, I was able to identify the missing branches I did not consider within my tests more efficiently.

I would choose JaCoCo's test coverage window because of its ease of use and efficiency.

Task 4

```
def test_from_dict(self):  
    """ Test account from dict """  
    account = Account()  
    data = {  
        'name': 'Robert Smith',  
        'email': 'vespiir@souncloud.com',  
        'phone_number': '702-688-1090',  
        'disabled': 'False',  
        'date_joined': '2017-02-05T12:00:00'  
    }  
    account.from_dict(data)  
    self.assertEqual(account.name, second: 'Robert Smith')  
    self.assertEqual(account.email, second: 'vespiir@souncloud.com')  
    self.assertEqual(account.phone_number, second: '702-688-1090')  
    self.assertEqual(account.disabled, second: 'False')  
    self.assertEqual(account.date_joined, second: '2017-02-05T12:00:00')
```

The `from_dict()` function takes a dictionary and sets the attributes of an `Account()` object to whatever values were stored in the dictionary that was passed in. To test this, I created a dictionary variable named `data` which holds an account's name, email, phone number, and other attributes. I used the `from_dict()` function to store this information into a newly initialized `Account()` object called `account`. Afterwards, I used a series of `assertEqual()` functions to see if each attribute of `account` was equal to the values from `data`.

```
def test_update_existing(self):  
    """ Test updating account with a valid id """  
    new_name = 'Robert Smith'  
    data = ACCOUNT_DATA[self.rand] # get a random account  
    account = Account(**data)  
    account.create()  
    account.name = new_name  
    if account.id is not None:  
        account.update()  
        self.assertEqual(account.name, new_name)
```

The `update()` function updates the database with the latest information of the `Account` object that's being updated. In the code I grab a random account's data and initialize a new `Account` object, `account`, using the data we grabbed. Then, I changed the name attribute of `account` and

only updated it if the account.id existed. Afterwards, I used an assertEquals() to check if account.name is equal to the new name I updated it with.

```
def test_delete_existing(self):  
    """ Test the deletion of an account """  
    data = ACCOUNT_DATA[self.rand] # get a random account  
    account = Account(**data)  
    account.create()  
    account_id = account.id  
    account.delete()  
    found_account = Account.find(account_id)  
    self.assertIsNone(found_account, msg: 'Account still exists.')
```

The delete() function removes an Account() object from the database. In the unit test, I grabbed a random account's data and created a new Account() object, account, in the database using that data. Then, I created a temporary variable, account_id, to hold account.id and deleted account. Then I stored the data from the find() function to a variable called found_account. The find() function grabs the data of an Account() object by using its id attribute. Afterwards, I used an assertIsNone() to ensure that found_account is equal to None.

```
def test_find(self):  
    """ Test account find """  
    data = ACCOUNT_DATA[self.rand] # get a random account  
    account = Account(**data)  
    account.create()  
  
    account_id = account.id  
    found_account = Account.find(account_id)  
    self.assertEqual(found_account, account, msg: 'Not the same account.')
```

The find() function returns an Account() object by passing a valid id. I first created an Account() object, account, and stored it in the database. I then made another variable, found_account, to hold the data from using the find() function. Finally, I used an assertEquals() to verify that found_account is equal to account.

Test Account Model

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 2.270s

Task 5

```
def test_update_a_counter(self):
    """It should update a counter and increment the value"""
    post_result = self.client.post('/counters/car')
    self.assertEqual(post_result.status_code, status.HTTP_201_CREATED)

    updated_result = self.client.put('/counters/car')
    self.assertEqual(updated_result.status_code, status.HTTP_200_OK)

    self.assertNotEqual(updated_result.json['car'], post_result.json['car'])
```

With this unit test, I'm in the red phase. I haven't yet implemented the `.put()` function but this test should cover most of what is asked. This unit test creates a counter called `car` using the `.post()` function and its return value is stored in `post_result`. I then use an `assertEqual()` to verify that the creation of this counter was successful. Then I continue to call the `.put()` function to update the counter by one and store its return value in `updated_result`. Like earlier, I use an `assertEqual()` to verify that the `.put()` function was successful. Finally, I use an `assertNotEqual()` to ensure that the counter value stored in `post_result` is different from the counter value stored in `updated_result`.

```
@app.route(rule: '/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Create a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} doesn't exist"}, status.HTTP_409_CONFLICT
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

The `update_counter()` function is similar to the `create_function`. The only difference is that the counter value is updated by 1. And I replaced the 201 status with a 200 instead. After implementing this function, the unit test finally worked and I'm in the green phase.

```

def test_read_a_counter(self):
    """It should update a counter and increment the value"""
    post_result = self.client.post('/counters/far')
    self.assertEqual(post_result.status_code, status.HTTP_201_CREATED)

    get_result = self.client.get('/counters/far')
    self.assertEqual(get_result.status_code, status.HTTP_200_OK)

    self.assertEqual(get_result.json['car'], post_result.json['car'])

    updated_result = self.client.put('/counters/far')
    self.assertEqual(updated_result.status_code, status.HTTP_200_OK)

    get_result = self.client.get('/counters/far')
    self.assertEqual(get_result.status_code, status.HTTP_200_OK)

    self.assertEqual(get_result.json['car'], updated_result.json['car'])

```

This unit test is similar to the above unit test for the put() function. Here, I decided to test two test cases. The first test is to see if the get() function is able to read the correct initial counter value of far. Then the second test verifies that after a put() function the get() still functions normally and reads the newly updated counter value of far.

```

@app.route(rule: '/counters/<name>', methods=['GET'])
def get_counter(name):
    """Create a counter"""
    app.logger.info(f"Request to get counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} doesn't exist"}, status.HTTP_409_CONFLICT
    return {name: COUNTERS[name]}, status.HTTP_200_OK

```

The get_counter() function is similar to the update_counter() function. The only difference is that the counter value is not updated at all. The function simply returns the current value stored in COUNTERS[name]. After implementing this function, the unit test finally worked and I'm in the green phase.

Arthur Valdez

Github Repository: https://github.com/valdea13/CS472_Group1