Unit Testing

Task 2.1

In Task 2.1 I created six new test cases that covered six different methods for two classes: Pellet and EmptySprite. For the Pellet class that is located at the package ‘level’, there were two methods I tested: getValue and getSprite. The class is called PelletTest and is in a similar format to the template from Task 2, PlayerTest. The hardest part of creating these two unit tests was instantiating an ImageSprite object. The Pellet constructor required an ImageSprite object. Luckily an EmptySprite object can be used as a placeholder which I used to instantiate instead. I then called the Pellet constructor to create a new Pellet object that holds 0 for its value and EmptySprite object as its image. Both getValue and getSprite are one line methods that return the value the Pellet holds and the image of the Pellet object. Since I created the new Pellet to hold 0 for its value I tested the getValue method to assert if the getValue method is equal to 0. Also since I created the new Pellet to be an Empty Sprite object, I tested the getSprite method to assert if the getSprite method is equal to the EmptySprite object I used to pass in for the Pellet constructor. The PelletTest increased the coverage of methods by 3% at the package ‘level’.

```
private final EmptySprite newEmptySprite = new EmptySprite();

// Creates new Pellet class
2 usages
private final Pellet newPellet = new Pellet( points: 0, newEmptySprite);

new *
@Test
void testGetValue() { assertThat(newPellet.getValue()).isEqualTo( expected: 0); }

new *
@Test
void testGetSprite() { assertThat(newPellet.getSprite()).isEqualTo(newEmptySprite); }
```

For the EmptySprite class that is located at the package ‘sprite’, there were four methods I tested: draw, split, getWidth, and getHeight. The class is called EmptySpriteTest. In this test I created a new EmptySprite object and also created a new Graphics object since all methods require an EmptySprite with some Graphics object. The Graphics object is set to null for the purpose of testing. For testing the draw method I called the draw method and set all values to 10. I then asserted to see if the draw is not equal to 10 since the draw method does nothing. For testing the split method I called the split method and set all values to 10 again. I then asserted to see if the split is equal to 0 since the split method does nothing as well. Lastly, for testing getWidth and getHeight I called the draw function to set all values to 10 but the draw function does nothing, so I asserted in both getWidth and getHeight equal to 0. The EmptySpriteTest increased the coverage of methods by 9% at the package ‘sprite’.

```java
// Initiates a new EmptySprite
8 usages
private final EmptySprite newEmptySprite = new EmptySprite();
3 usages
private final Graphics graphics = null;
new *
@Test
void testDraw() {
    newEmptySprite.draw(graphics, x: 10, y: 10, width: 10, height: 10);
    assertThat(newEmptySprite.getWidth()).isNotEqualTo( other: 10);
}

new *
@Test
void testSplit() {
    newEmptySprite.split( x: 10, y: 10, width: 10, height: 10);
    assertThat(newEmptySprite.getWidth()).isEqualTo( expected: 0);
}

new *
@Test
void testGetWidth() {
    newEmptySprite.draw(graphics, x: 10, y: 10, width: 10, height: 10);
    assertThat(newEmptySprite.getWidth()).isEqualTo( expected: 0);
}

new *
@Test
void testGetHeight() {
    newEmptySprite.draw(graphics, x: 10, y: 10, width: 10, height: 10);
    assertThat(newEmptySprite.getHeight()).isEqualTo( expected: 0);
}
```

Task 3

The coverage results from JaCoCo are similar to the ones I got from Intellij in the last task. Both coverage results show which lines are covered, but the main difference is that JaCoCo shows which branches are not covered and lists it as a separate column in the coverage report. Also, JaCoCo colors the coverage a little differently when it comes to branch coverage. I did find the source code visualization from JaCoCo on uncovered branches helpful as it is different colors to branches that are covered in red and or yellow instead of Intellij only using red or green colors for coverage. I prefer JaCoCo's report coverage window as it has more detail on branch coverage as well as separate colors to help to identify which branches are missed.

Task 4

In Task 4 I created 6 new test units to achieve overall 100% test coverage. The 6 new test units include: test_repr, test_to_dict, test_from_dict, test_update, test_find, and test_delete. For test_repr, I followed the instructions word for word and 'nosetested' to see that line 26 was covered and changed the coverage from 72 to 74%. For test_to_dict I followed the instructions word for word and 'nosetested' to see that line 30 was covered and changed the coverage from 74 to 76%.

For test_from_dict, I created random data by getting a random account and called the constructor Account inputting the random data. I then called the from_dict method and self asserted to see if account data is equal to the random data generated. The coverage changed from 76% to 81%.

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

For test_update, I again created random data in a data variable and called the constructor Account inputting the random data. I then called the from_dict method and then called the create method. I then made a new updated data list called new_updated_data and made new data under the name, email, and phone_number attributes. I then recalled the from_dict method and passed in the new_updated_data attached to the "account" variable and then called update method. I

then self asserted to see if the “account” variable is equal to the attributes that were created in the new_updated_data (“Updated name”, “updated@gmail.com”, and “000-000-0000”). While I thought I was done, I saw that I missed when the test_update method can throw a DataValidationError when trying to update an account with an empty ID field. So I recalled the account constructor and did a try, except branch making sure the error message “Update called with empty ID field” is the same as the DataValidationError test_update throws.

```
def test_update(self):
    """ Test update account in the database """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.from_dict(data)
    account.create()
    new_updated_data = {
        "name": "Updated name",
        "email": "updated@gmail.com",
        "phone_number": "000-000-0000",
    }
    account.from_dict(new_updated_data)
    account.update()
    self.assertEqual(account.name, "Updated name")
    self.assertEqual(account.email, "updated@gmail.com")
    self.assertEqual(account.phone_number, "000-000-0000")

    """ Test update account with empty ID field"""
    account = Account()
    try:
        account.update()
    except DataValidationError as error:
        error_expected_message = "Update called with empty ID field"
        self.assertEqual(str(error), error_expected_message)
```

For test_find, I did the same procedures as previous test cases. That is making the same random data variable, calling the Account constructor, calling from_dict method, and creating the account. Then I called the Account.find method passing in the account.id making the variable result hold the id and self asserted to see if result.id is equal to account.id.

```
def test_find(self):
    """ Test find account by its ID """
    data = ACCOUNT_DATA[self.rand]
    account = Account()
    account.from_dict(data)
    account.create()
    result = Account.find(account.id)
    self.assertEqual(result.id, account.id)
```

For test_delete, I did the same procedures as previous test cases. That is making the same random data variable, calling the Account constructor, calling from_dict method, creating the account, and then deleting the account calling the delete method. I then self asserted to see if the length of Accounts is equal to 0 since the account was deleted and no accounts should be created in the Account list.

```
def test_delete(self):
    """ Test delete account in the database """
    data = ACCOUNT_DATA[self.rand]
    account = Account()
    account.from_dict(data)
    account.create()
    account.delete()
    self.assertEqual(len(Account.all()), 0)
```

Task 5

For implementing the test case for test_update_a_counter I first wrote the test case code. In this code I started off with a comment that states the test case should update the counter. I then made a call to Create a counter by setting client to app.test_client() and result to the new counter by calling client.post('/counter/<name>'). I ensured that it returned a successful return code by asserting the result status code is equal to the status message 201_CREATED. I then checked the counter value by setting a new variable called baseline_value to result.get_json value. I then asserted the baseline_value to see if it was equal to 0 since a new counter value should hold 0. I then made a new variable called update_result and called the update_a_counter method by client.put('/counter/<name>'). I then asserted the update_result to see if it is equal to the status message 200_OK since an update uses a PUT request and returns code 200_OK. I then set a new variable called update_result_value and grabbed the result by using .get_json() and asserted to see if update_result_value is equal to baseline_value + 1.When I ran nosetests I received an assertion error 405 != 200 because I did not make the update_a_counter method and have not created a route for the method PUT on endpoint /counters/<name> for the method. This would be the RED phase in TDD testing.

```
def test_update_a_counter(self):
    """It should update the counter"""
    client = app.test_client()
    result = client.post('/counters/col')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    baseline_value = result.get_json()['col']
    self.assertEqual(baseline_value, 0)
    update_result = client.put('/counters/col')
    self.assertEqual(update_result.status_code, status.HTTP_200_OK)
    update_result_value = update_result.get_json()['col']
    self.assertEqual(update_result_value, baseline_value + 1)
```

So, I then moved on to creating the update_counter method in counter.py by first creating the route for method PUT on endpoint /counters/<name>. I did this by the line of code "@app.route('/counters/<name>', methods=['PUT']). In the method I first put a comment that the method is supposed to update a counter. I then created an if statement that checks if the name passed in to the method is in the COUNTERS list. If so It will return the name: COUNTERS[name] as well as the status message 200_OK. When I ran nosetests the test passed and covered all lines. This would be the GREEN phase in TDD testing.

```python
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    if name in COUNTERS:
        COUNTERS[name] += 1
        return{name: COUNTERS[name]}, status.HTTP_200_OK
```

For implementing the test case for read_a_counter I first wrote the test case code. In this code I started off with a line comment that states the test case should read the counter. I then made a call to Create a counter by setting client to app.test_client() and result to the new counter by calling client.post('/counter/<name>'). I ensured that it returned a successful return code by asserting the result status code is equal to the status message 201_CREATED. I then called the read_a_counter method by creating a new variable called read_result set to client.get('/counters/loo/') since the GET method calls the read_a_counter method. I then asserted if the read_result status code is equal to 200_OK since the GET request outputs a 200_OK response. This is where I stopped and the first RED phase when testing using nosetests. I received an assertion error 405 != 200 because I did not make the read_a_counter method and have not created a route for the method GET on the endpoint for the method.

```python
def test_read_a_counter(self):
    """It should read the counter"""
    client = app.test_client()
    result = client.post('/counters/loo')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    read_result = client.get('/counters/loo')
    self.assertEqual(read_result.status_code, status.HTTP_200_OK)
    fake_result = client.get('/counters/fake')
    self.assertEqual(fake_result.status_code, status.HTTP_404_NOT_FOUND)
```

So, I then moved on to creating the read_a_counter method in counter.py by first creating the route for method GET on endpoint/counters/<name>. I did this by the line of code "@app.route('/counters/<name>', methods=['GET']. In the method I put a comment that states it reads a counter. I then created an if statement that checks if the name passed in is in the COUNTERS list. If so It will return the names and status message 200_OK. When I ran nosetests the test passed and covered all lines. This would be the first GREEN phase in TDD

testing. I then realized that the code can be entered into the REFACTOR phase when trying to read a fake or no COUNTER to be read in. So I called a fake result and asserted to see if the fake status code is equal to 404_NOT_FOUND. This is the second RED phase in TDD testing. So moving onto counters.py I add an else branch that just returns an error message that the name does not exist and the status message 404_NOT_FOUND.

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

Trevor Tippery
GitHub link: https://github.com/trevortippery/CS472_Group1