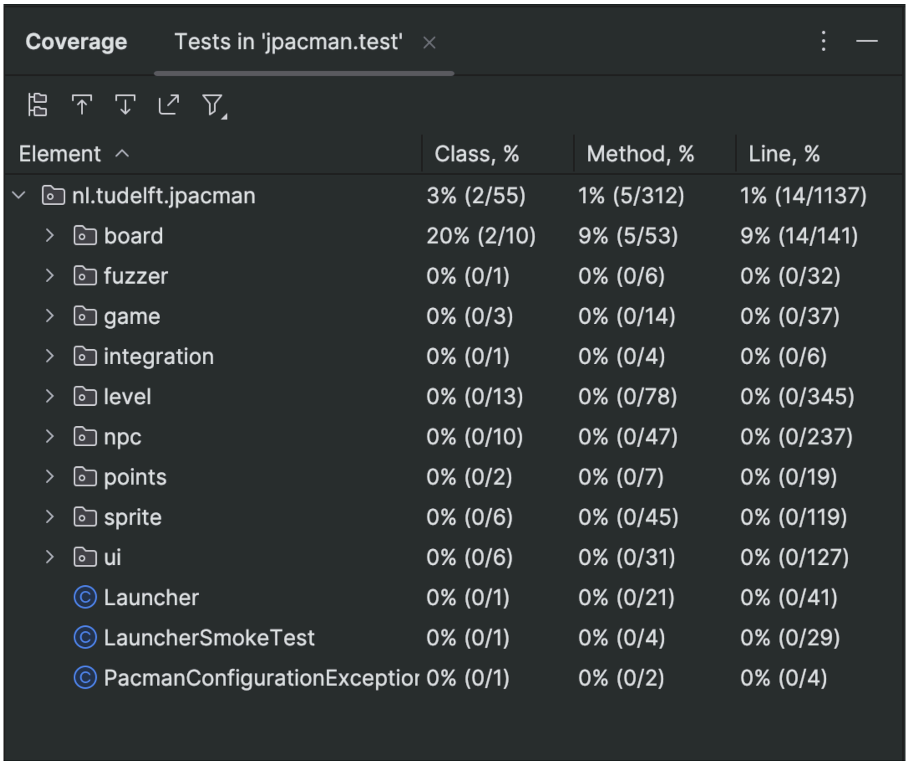
CS472 Unit Testing Report

Task 1

This is the initial coverage of the project which is not good enough for a project.

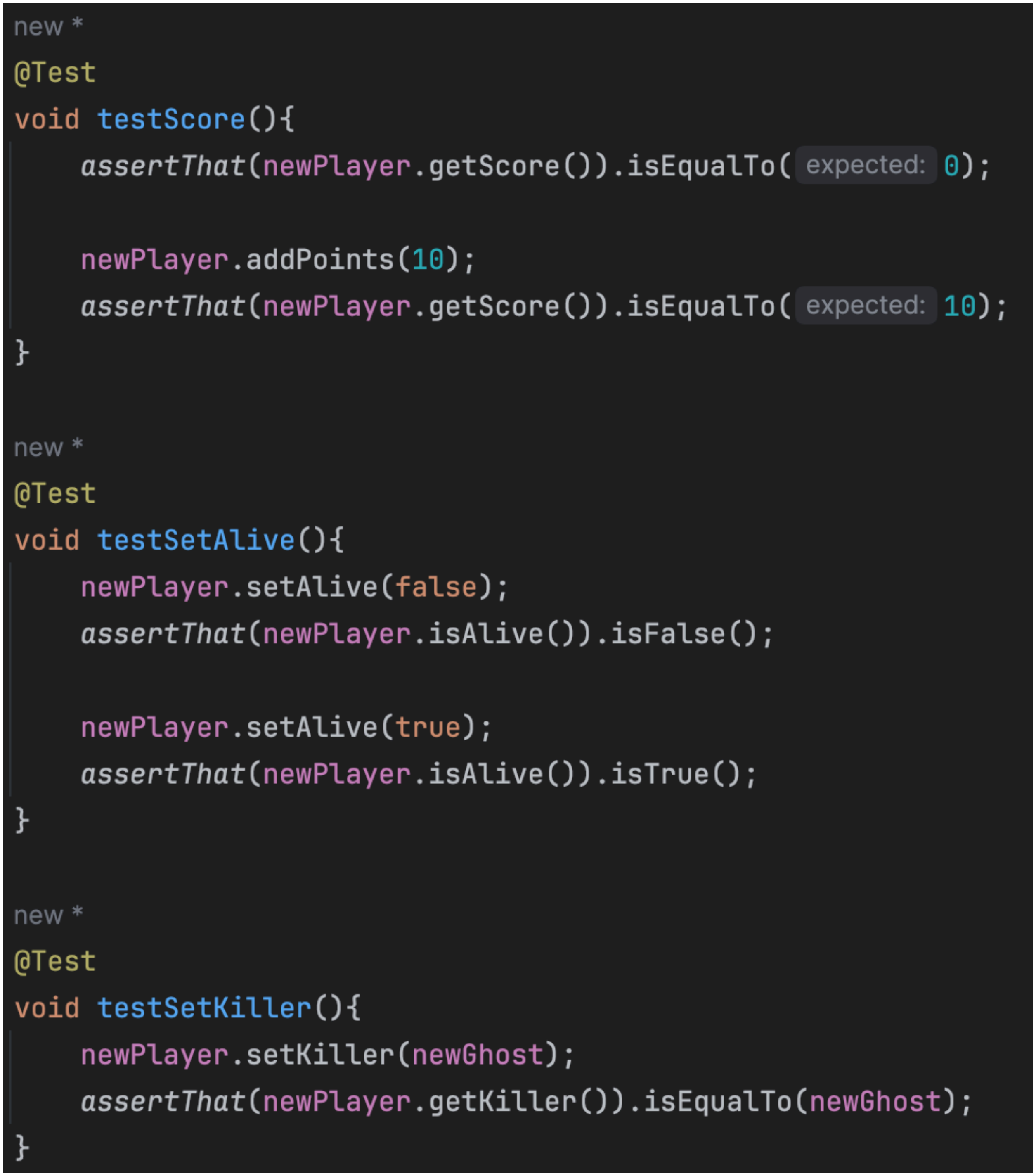


Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	0% (0/13)	0% (0/78)	0% (0/345)
npc	0% (0/10)	0% (0/47)	0% (0/237)
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)
PacmanConfigurationExceptior	0% (0/1)	0% (0/2)	0% (0/4)

Task 2.1

This task included finding three more methods to conduct unit tests on. I chose to test 3 more methods in the Player class. I intended to only test addPoints(), setAlive(), and setKiller(). While I was writing the unit test for those functions, I realized that a useful test would have to do more than just call those methods. It led me to also include the methods getPoints(), and getKiller().

The following image is a code snippet of my unit tests. My unit tests include the methods I listed above.

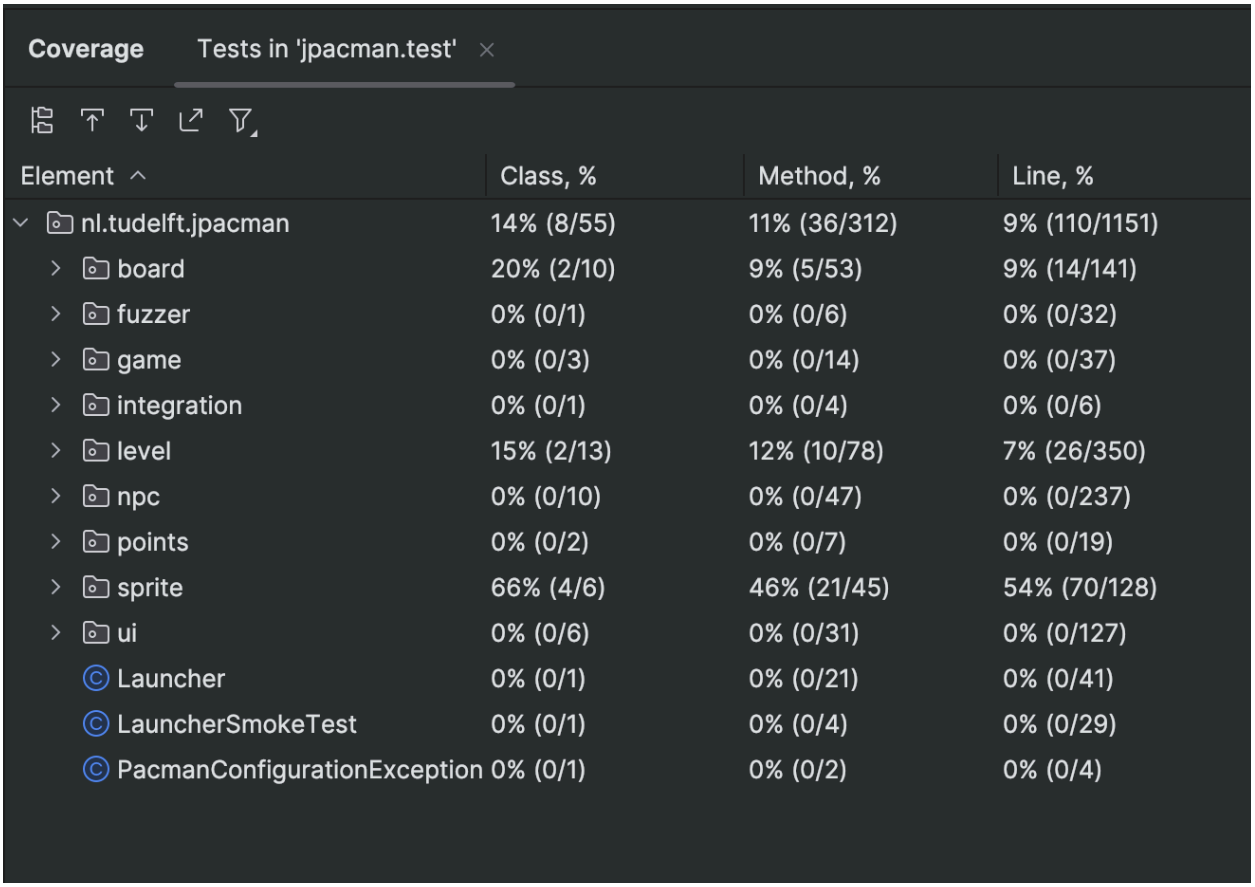


```
new *
@Test
void testScore(){
    assertThat(newPlayer.getScore()).isEqualTo( expected: 0);

    newPlayer.addPoints(10);
    assertThat(newPlayer.getScore()).isEqualTo( expected: 10);
}

new *
@Test
void testSetAlive(){
    newPlayer.setAlive(false);
    assertThat(newPlayer.isAlive()).isFalse();

    newPlayer.setAlive(true);
    assertThat(newPlayer.isAlive()).isTrue();
}

new *
@Test
void testSetKiller(){
    newPlayer.setKiller(newGhost);
    assertThat(newPlayer.getKiller()).isEqualTo(newGhost);
}
```

The test testScore() initially asserts that the newPlayer object has a score of 0. This is what tests the getScore() function. It should assert that the score is 0 because as a new object of newPlayer, it wouldn't have any points. Then we use the addPoints() function to add 10 points. This is then tested asserting that the newPlayer object now has a score of 10.

The method testSetAlive() is used to test the setAlive() function of the Player class. We start by setting the newPlayer isAlive() to false by calling setAlive() with the false parameter. We check this by then asserting that the newPlayer object isAlive() is indeed false. We further check the setAlive() by calling it with the true parameter, and then asserting that the isAlive() variable is set to true.

The method testSetKiller() is used to test the setKiller() function of the Player class. We start by calling the setKiller() to an object of Ghost called newGhost. Then, we assert that the newPlayer's killer is indeed that newGhost by asserting that getKiller() is equal to that newGhost.



Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	14% (8/55)	11% (36/312)	9% (110/1151)
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	15% (2/13)	12% (10/78)	7% (26/350)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	66% (4/6)	46% (21/45)	54% (70/128)
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)

This screenshot shows the new test coverage after I implemented the tests. We can see that the line coverage in the level package has increased from 0% to 7%. We can also see that class coverage is up 15% and method coverage is up 12%.

Task 3:

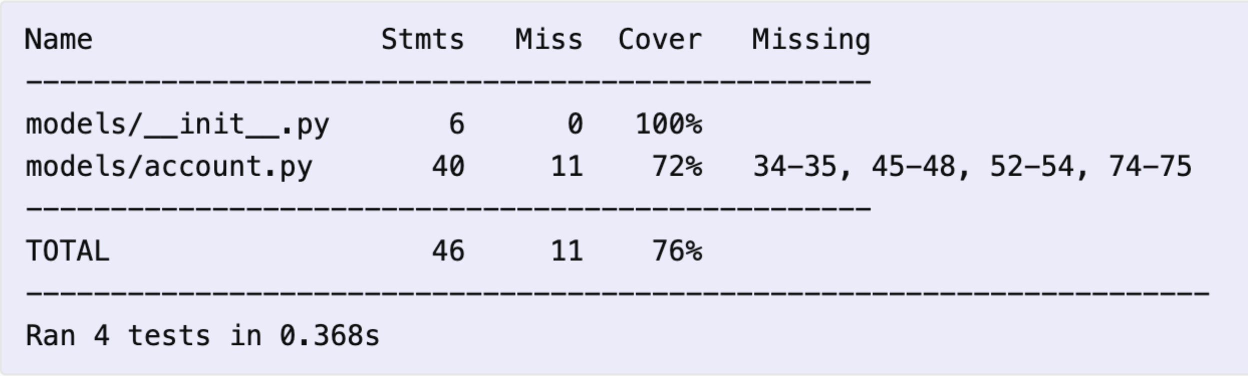
The coverage results from JaCoCo are indeed similar to the ones I got from IntelliJ in the last task. While it may not be exactly the same, JaCoCo reported that the tests had 83% line test coverage while IntelliJ reported that the tests had 87% line test coverage.

I found the source code visualization from JaCoCo on uncovered branches helpful because it shows that while your line coverage is tested, there are still situations in the code where it still is not.

I prefer IntelliJ's coverage window because it is included in the IDE. I like to have all of my tools in the same interface because it makes it easy for me to navigate through. Even though JaCoCo’s report includes branch coverage, the line coverage from IntelliJ’s is a great indicator of my testing.

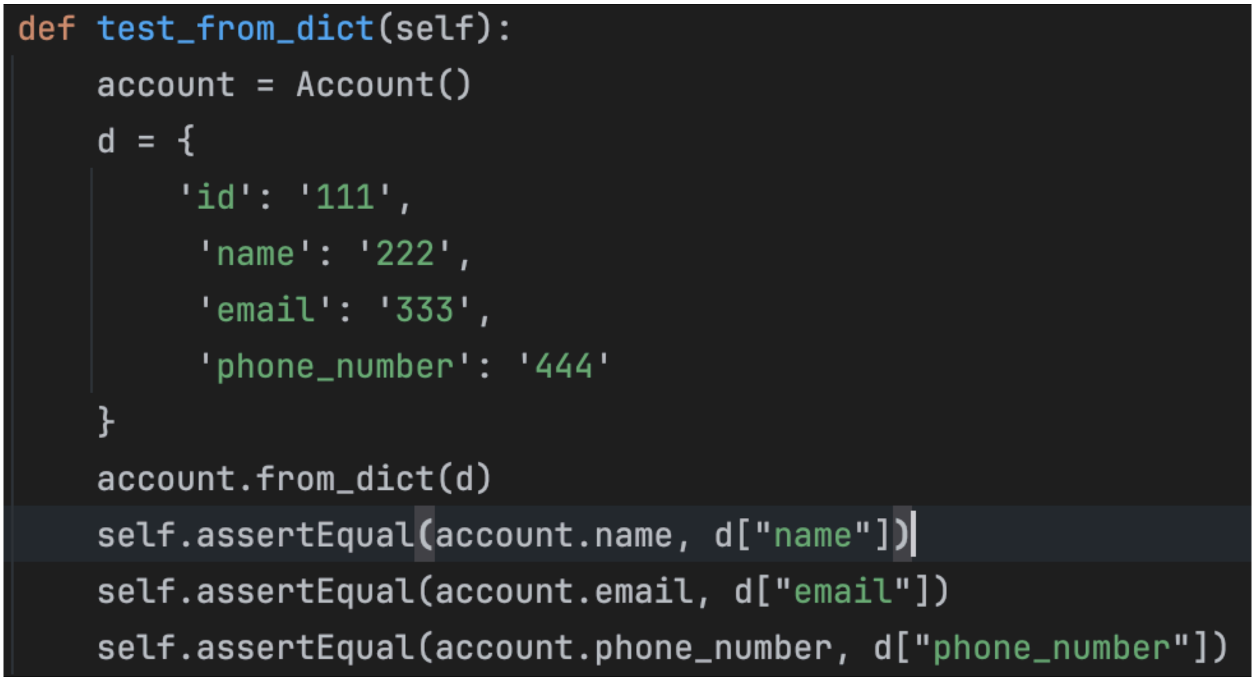
Task 4:

Task 4 starts off with 76% line coverage



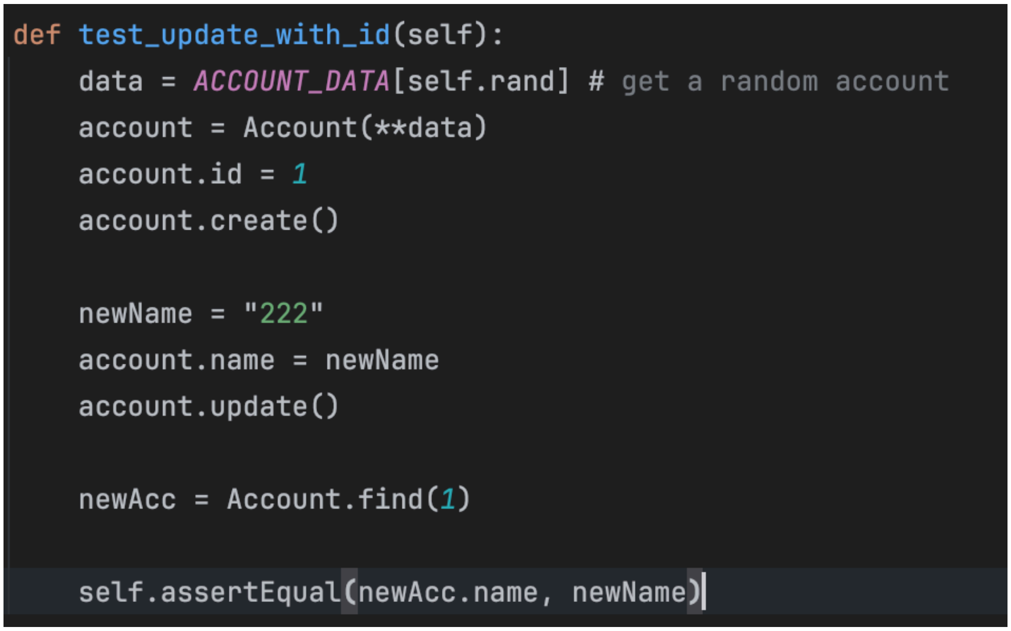
```
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models/__init__.py       6      0   100%
models/account.py       40     11    72%   34-35, 45-48, 52-54, 74-75
--------------------------------------------------
TOTAL                   46     11    76%
----------------------------------------------------------------------
Ran 4 tests in 0.368s
```

In order to increase this coverage, I had to write some tests for methods in models/account.py. I created five more unit tests.



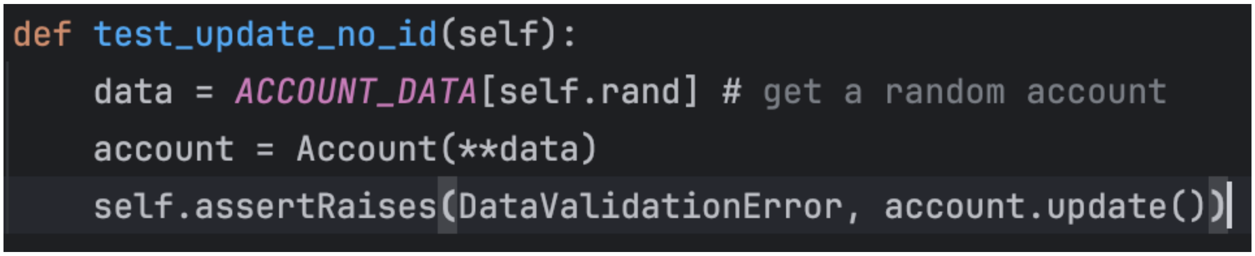
```python
def test_from_dict(self):
    account = Account()
    d = {
        'id': '111',
         'name': '222',
         'email': '333',
         'phone_number': '444'
    }
    account.from_dict(d)
    self.assertEqual(account.name, d["name"])
    self.assertEqual(account.email, d["email"])
    self.assertEqual(account.phone_number, d["phone_number"])
```

The first test is for the from_dict() method. I created an Account object and a dictionary containing some information about this account. I then called the from_dict() method and passed in the dictionary. In order to test that the from_dict() method worked, I checked that the account matched the name, email, and phone number given in the dictionary.



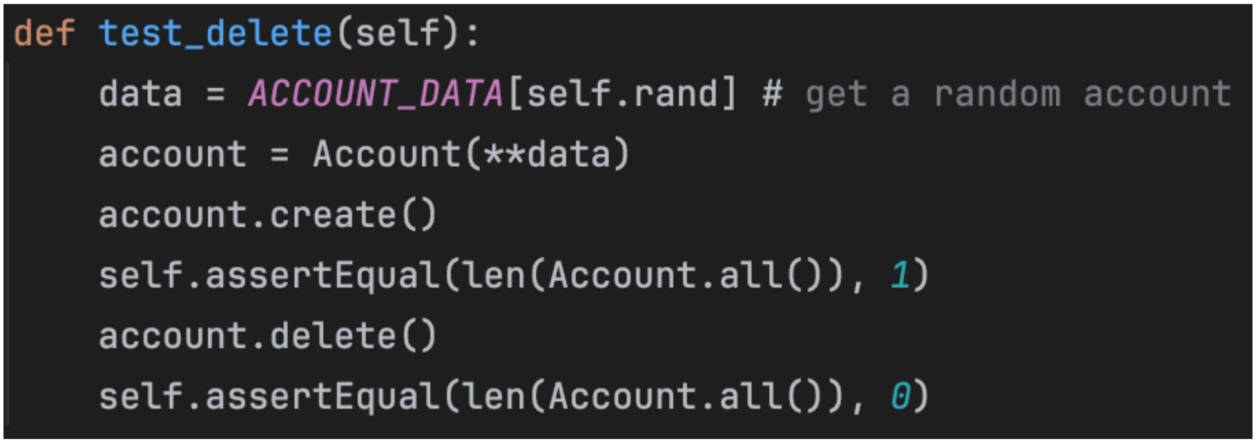
```python
def test_update_with_id(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.id = 1
    account.create()

    newName = "222"
    account.name = newName
    account.update()

    newAcc = Account.find(1)

    self.assertEqual(newAcc.name, newName)
```

The next test is for the update() method with an ID passed in as the parameter. I created a random account with the ID of 1. The ID was given in order to be able to use the find() method. I then created a newName variable and then updated the account name. I used the find method to create a newAcc variable which would hold the information of the original account. Then I asserted that the newAcc name and the variable newName were equal to see if the update() method worked.



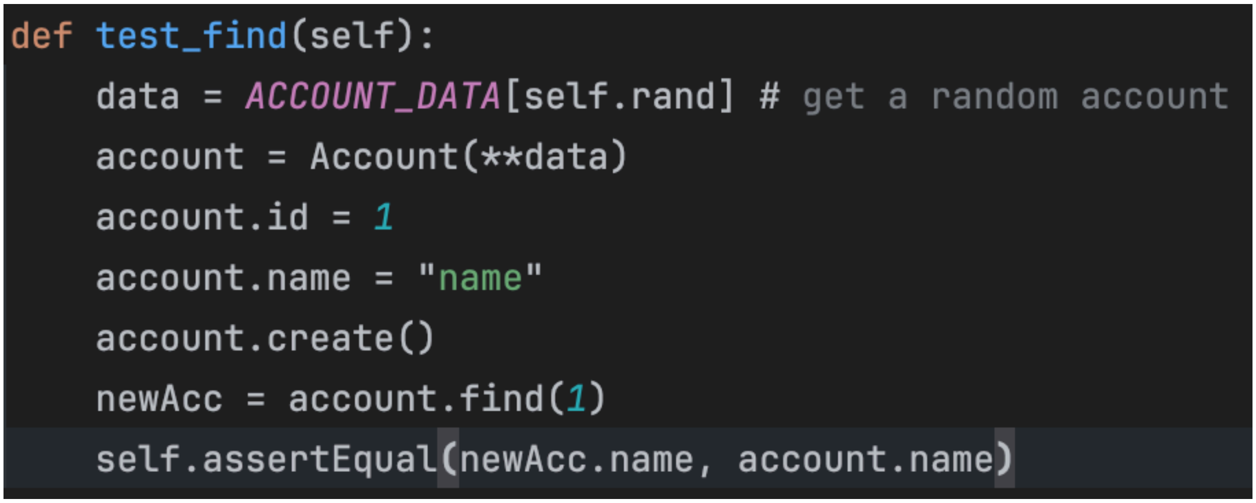
```
def test_update_no_id(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    self.assertRaises(DataValidationError, account.update())
```

Next test was for the update() method when no ID is passed. This is because in the update() method, there is a branch for when there is no ID passed. I called the update() method with no ID after creating a random account. Then I checked if the validation error was thrown because there was no ID.



```
def test_delete(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    self.assertEqual(len(Account.all()), 1)
    account.delete()
    self.assertEqual(len(Account.all()), 0)
```

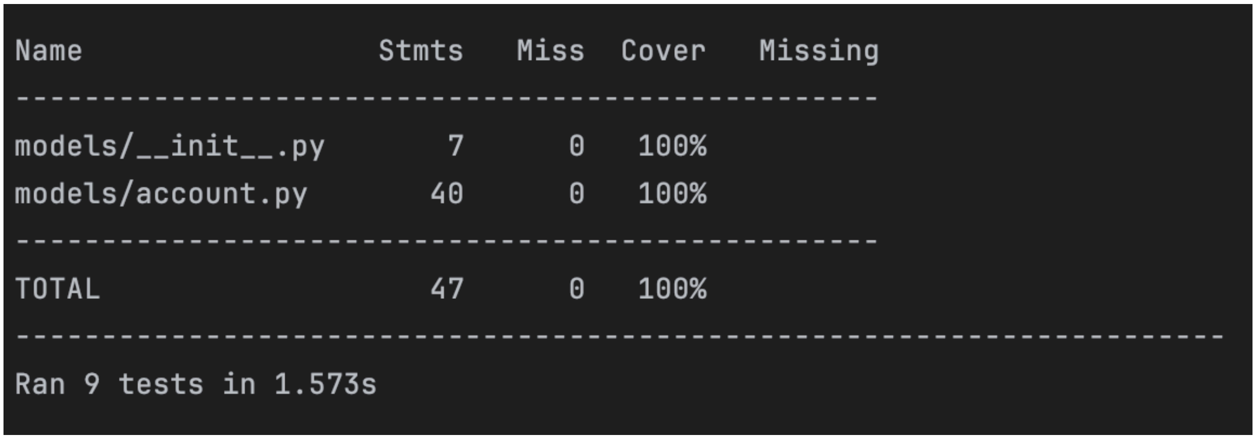
This test was for the delete() method. It was pretty straight forward. I created a random account and asserted that the length of all of the accounts was 1. This was done by calling the all() method which returns all of the accounts. Then I delete the account and assert that the length of all accounts is 0.



```python
def test_find(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.id = 1
    account.name = "name"
    account.create()
    newAcc = account.find(1)
    self.assertEqual(newAcc.name, account.name)
```

This test for the find() method. I start by creating a random account and assigning it an id of 1 and a name of ‘name’. I then create a newAcc using the find() method and passing in the ID of 1 to get the same account I just created. By asserting that the newAcc name and original account name are equal, we can verify that the find() method works as intended.

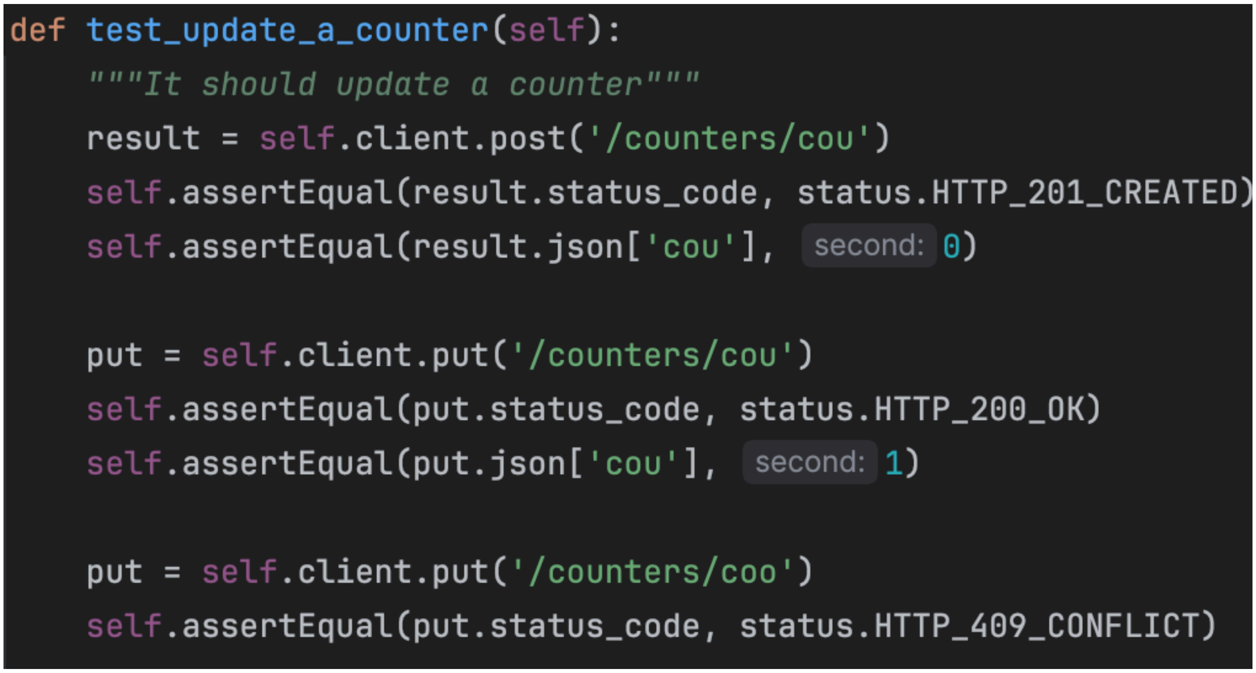
This is the test coverage with the new tests implemented.



```
Name                     Stmts   Miss  Cover   Missing
-----------------------------------------------------
models/__init__.py           7      0   100%
models/account.py           40      0   100%
-----------------------------------------------------
TOTAL                       47      0   100%
----------------------------------------------------------------------
Ran 9 tests in 1.573s
```

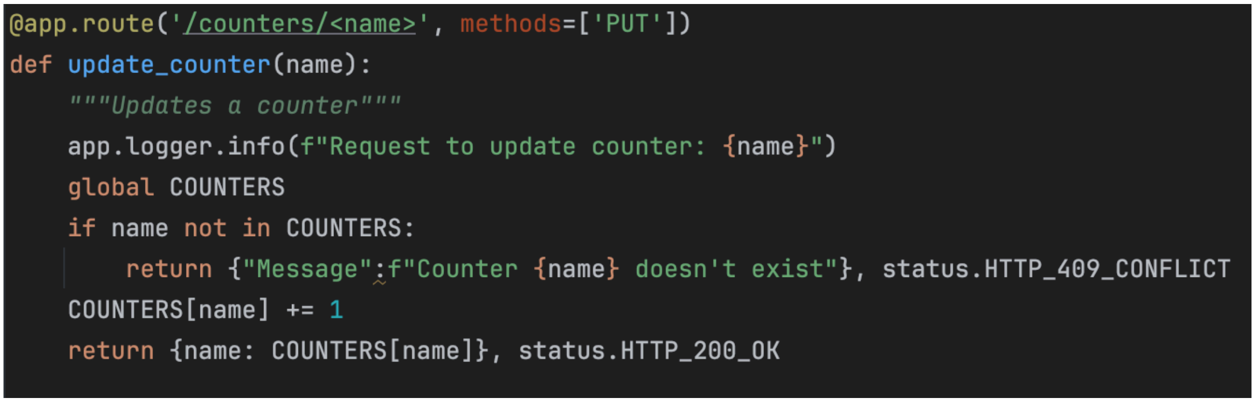
Task 5:

For this task, we had to implement tests before implementing the logic for the method.



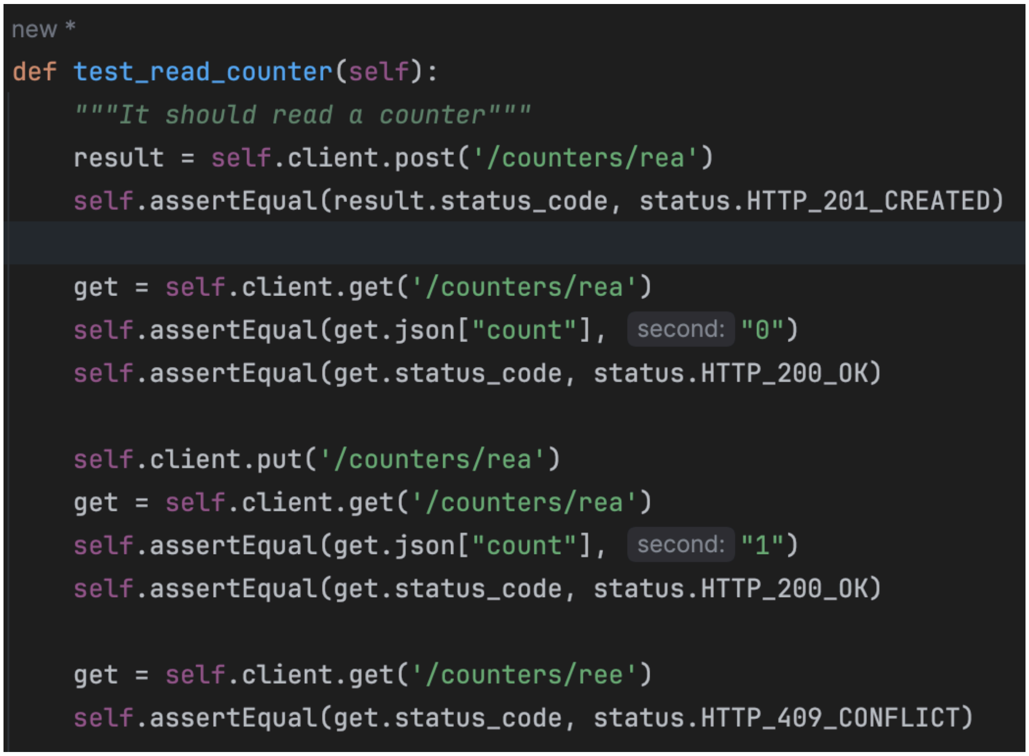
```
def test_update_a_counter(self):
    """It should update a counter"""
    result = self.client.post('/counters/cou')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    self.assertEqual(result.json['cou'], second: 0)

    put = self.client.put('/counters/cou')
    self.assertEqual(put.status_code, status.HTTP_200_OK)
    self.assertEqual(put.json['cou'], second: 1)

    put = self.client.put('/counters/coo')
    self.assertEqual(put.status_code, status.HTTP_409_CONFLICT)
```

This is the test_update_a_counter() test. I made a post request to create a new counter named "cou". I then checked to make sure that this request was created and that it was equal to 0. I then made a put request in order to update the "cou" counter. I checked that the request was ok by asserting that the status code was equal to HTTP_200_OK. Then I checked that the value from the response form the put request was equal to 1, meaning that it had updated. Then I checked the case that a counter that didn't exist was being updated by making a put request with a counter named "coo" which does not exist and making sure that the status code returned HTTP_409_CONFLICT. I ran nosetest and made sure that the response was RED.



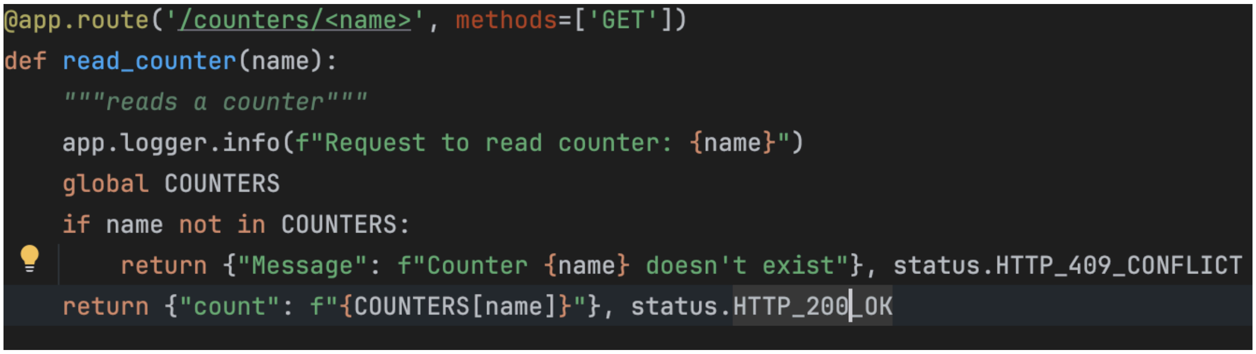
```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Updates a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message":f"Counter {name} doesn't exist"}, status.HTTP_409_CONFLICT
  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

This is the update_counter() method which takes in a parameter name to the route '/counters/<name>' which is a put request. The method makes sure the name is in COUNTERS to check that it exists. Otherwise, it sends an HTTP_409_CONFLICT. If the name is in COUNTERS, it increments the value of the specific counter by 1 and returns the counter and the status HTTP_200_OK. After testing, this gives a GREEN status for the test.



```
new *
def test_read_counter(self):
    """It should read a counter"""
    result = self.client.post('/counters/rea')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    get = self.client.get('/counters/rea')
    self.assertEqual(get.json["count"], second: "0")
    self.assertEqual(get.status_code, status.HTTP_200_OK)

    self.client.put('/counters/rea')
    get = self.client.get('/counters/rea')
    self.assertEqual(get.json["count"], second: "1")
    self.assertEqual(get.status_code, status.HTTP_200_OK)

    get = self.client.get('/counters/ree')
    self.assertEqual(get.status_code, status.HTTP_409_CONFLICT)
```

This is the test_read_counter() test. It creates a new counter named 'rea' and checks that it has been created by comparing the status code to HTTP_201_CREATED. Then it creates a get request to check if the response is equal to 0 and check if the status code is HTTP_200_OK. Then it creates a put request to update the counter. It creates a second variable 'get' to read in the count. It then checks that the new variable count is equal to 1 to make sure that it has been updated. It also checks if a counter that doesn't exist wants to read its and denies it by making sure the status code is equal to HTTP_409_CONFLICT.



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

This is the read_counter() method which takes in a parameter name to the route ‘/counters/<name>’ which is a get request. The method makes sure the name is in COUNTERS to check that it exists. Otherwise, it sends an HTTP_409_CONFLICT. If the name is in COUNTERS, it returns the value counter and the status HTTP_200_OK. After testing, this gives a GREEN status for the test.

Summary:

I expected to learn a lot while writing these tests. I have written unit tests before but always as an afterthought. This lab was useful in the sense that it helped me learn how to set up my code for success. Writing unit tests before the code sets up a foundation for the code and really guides you in the process.

Aligary Patawaran

GitHub Link: https://github.com/Aligary/CS472_Group1.git