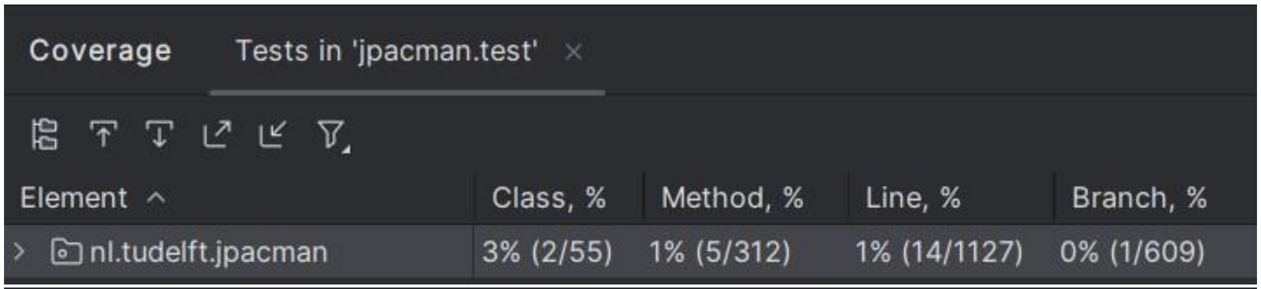
Unit Test Report

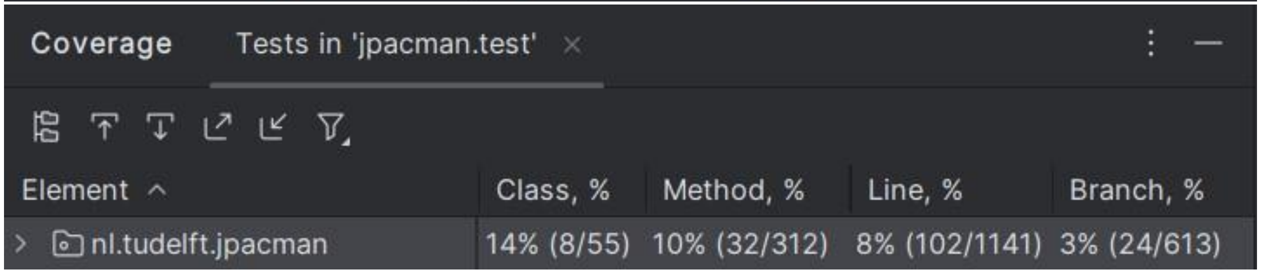
Fork Repository: https://github.com/thecatfour/CS472-Git-and-GitHub/tree/jpacman_tests

Task 1

Although the isAlive() unit test increases the class and method coverage by about 10%, it is not good enough as a vast majority of the code is still untested. The top image shows the results before the isAlive() test, and the bottom image shows the coverage after the test is implemented.



Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1127)	0% (1/609)



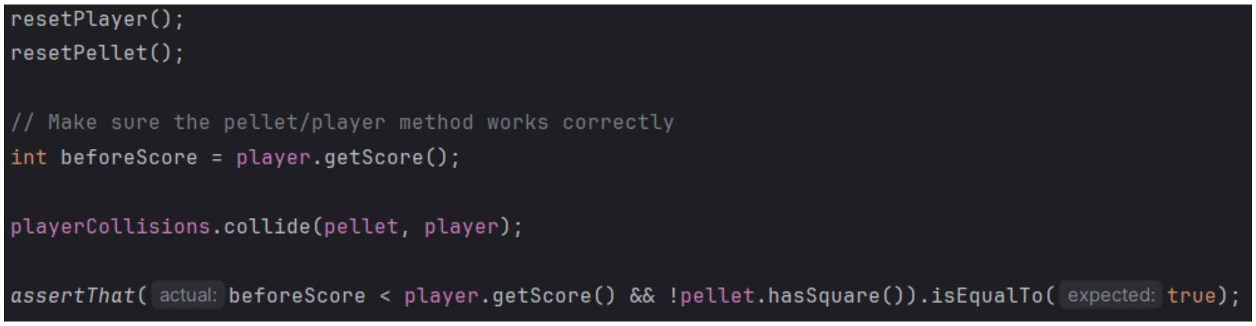
Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	14% (8/55)	10% (32/312)	8% (102/1141)	3% (24/613)

Task 2

This report provides an overview of the unit tests for the PlayerCollisions.java class. The unit tests involve the following process:

1. Initialize test objects
2. Call the method being tested
3. Check that appropriate changes were made to the test objects

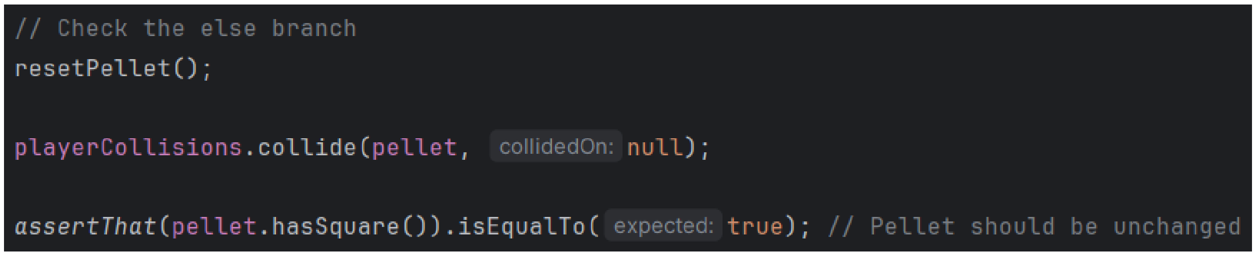
The following code segment shows the general pattern followed by the unit tests.



```
resetPlayer();
resetPellet();

// Make sure the pellet/player method works correctly
int beforeScore = player.getScore();

playerCollisions.collide(pellet, player);

assertThat( actual: beforeScore < player.getScore() && !pellet.hasSquare()).isEqualTo( expected: true);
```

This section checks collision between a player and a pellet. If the collide method is working properly, the player should have a higher score. Also, the pellet should have been moved off the board, so the pellet object should not have a square.

In order to check the branches where the condition is false, the same process is followed, but different arguments are given to the methods as shown by the following code segment.

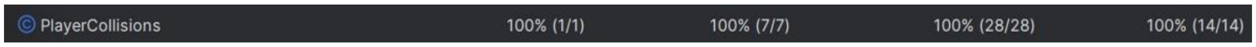


```
// Check the else branch
resetPellet();

playerCollisions.collide(pellet, collidedOn: null);

assertThat(pellet.hasSquare()).isEqualTo( expected: true); // Pellet should be unchanged
```

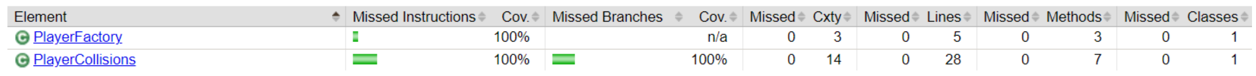
Basically, the second argument is given "null" to ensure that no branch is taken. The assert line then checks the object to make sure no change was made. In this case, the pellet should still be on its square.

Task 3

Regarding the test results for the class that I wrote unit tests for (PlayerCollisions.java), the IntelliJ and JaCoCo results are similar. As shown by the images below, both show a 100% coverage for PlayerCollisions.



PlayerCollisions	100% (1/1)	100% (7/7)	100% (28/28)	100% (14/14)



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
PlayerFactory	🟩	100%		n/a	0	3	0	5	0	3	0	1
PlayerCollisions	🟩	100%	🟩	100%	0	14	0	28	0	7	0	1

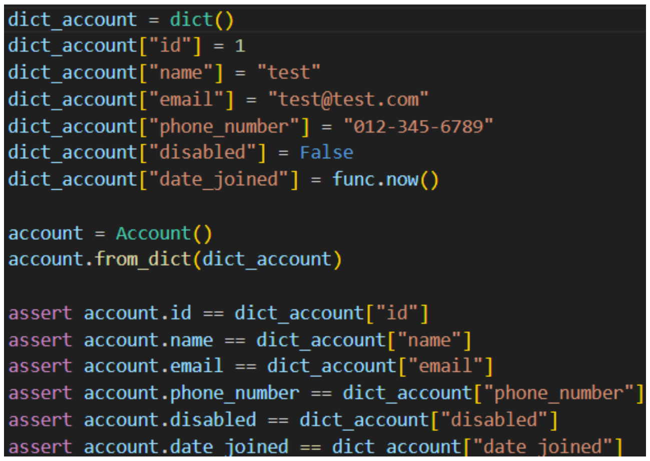
However, the two reports also differ quite significantly. For example, the JaCoCo report has a "missed instruction" metric while IntelliJ does not. This suggests that the two reports have different ways of calculating test coverage.

The source code visualization on JaCoCo is useful for spotting branches that are not covered by tests. It uses a color code to help these sections pop. Green means the branches are all covered, yellow means some are covered, and red means the branches are missing test coverage. The colors make it easier to spot these problems by highlighting which instructions are not being tested.

In terms of convenience, IntelliJ's coverage report is more useful as it is immediately accessible in the IDE. However, in terms of helping find what parts need coverage, JaCoCo is more useful as it clearly shows which branches and sections of code are not covered by tests.

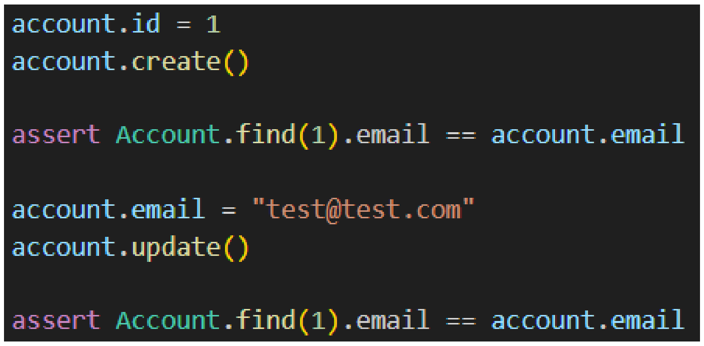
Task 4

Lines 34-35 involve changing an account object's fields using a dict with the from_dict() method. The test involves creating a dict, calling the method, and comparing the results.

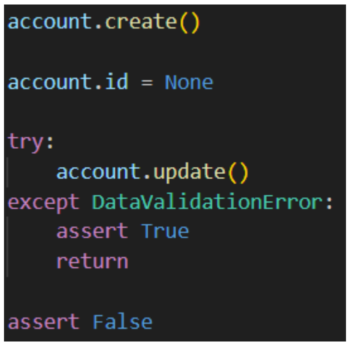


```python
dict_account = dict()
dict_account["id"] = 1
dict_account["name"] = "test"
dict_account["email"] = "test@test.com"
dict_account["phone_number"] = "012-345-6789"
dict_account["disabled"] = False
dict_account["date_joined"] = func.now()

account = Account()
account.from_dict(dict_account)

assert account.id == dict_account["id"]
assert account.name == dict_account["name"]
assert account.email == dict_account["email"]
assert account.phone_number == dict_account["phone_number"]
assert account.disabled == dict_account["disabled"]
assert account.date_joined == dict_account["date_joined"]
```

Lines 45-48 involve updating an account that is already in the database. Due to a branch in case of a lack of an account id, there needs to be two separate tests. The left test attempts to change the email of an account. The right test attempts to update an account without an id.

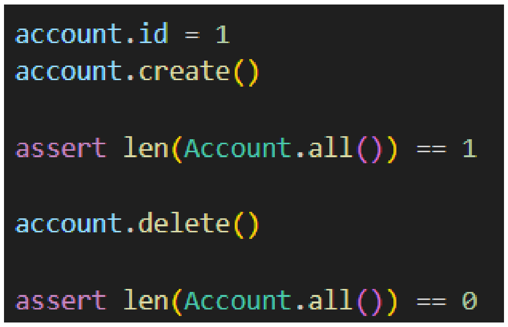


```python
account.id = 1
account.create()

assert Account.find(1).email == account.email

account.email = "test@test.com"
account.update()

assert Account.find(1).email == account.email
```



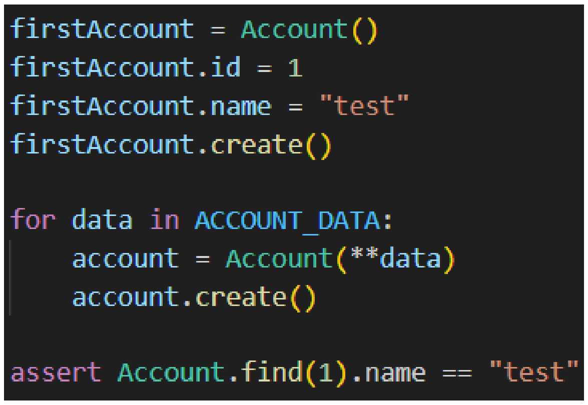
```python
account.create()

account.id = None

try:
    account.update()
except DataValidationError:
    assert True
    return

assert False
```

Lines 52-54 involves deleting an account that is in the database. In order to properly test this method, there first needs to be an account in the database. The test method creates and deletes an account and checks the database to make sure it was changed.

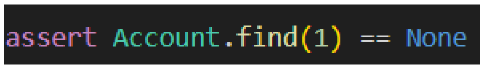


```python
account.id = 1
account.create()

assert len(Account.all()) == 1

account.delete()

assert len(Account.all()) == 0
```

Lines 74-75 involve the find function which finds an account in the database with the inputted id. In order to test this method, there needs to be a case with multiple entries in the database as well as a case with no entries. The left test creates a specific account followed by other random accounts to fill the database. It then tries to find the correct database. The right test simply makes sure a find call with an id not in the database returns None.



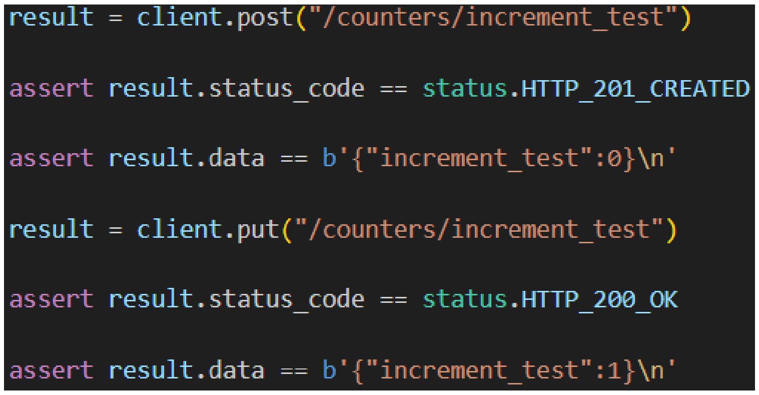
```python
firstAccount = Account()
firstAccount.id = 1
firstAccount.name = "test"
firstAccount.create()

for data in ACCOUNT_DATA:
    account = Account(**data)
    account.create()

assert Account.find(1).name == "test"
```



```python
assert Account.find(1) == None
```

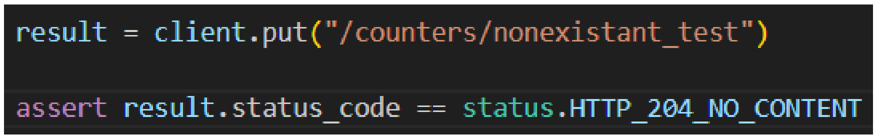
Task 5

When first running the test_update_a_counter() method, the error that it gives regards the status code after the .put call. Since the update_counter() method was not implemented yet, the test fails with a 405 METHOD NOT ALLOWED error. The following code segment is the update_a_counter() test.



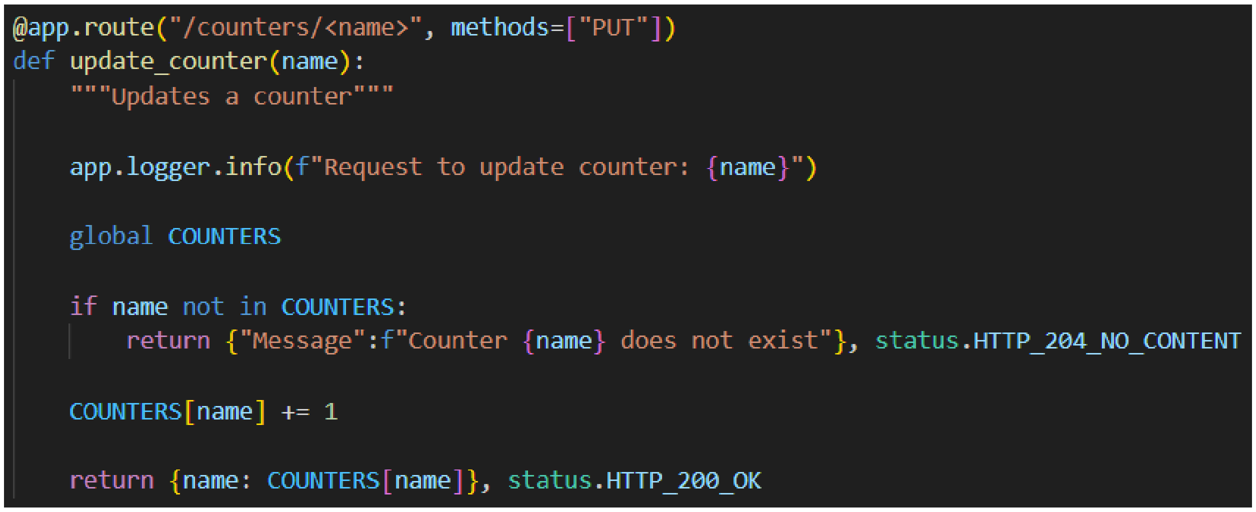
```python
result = client.post("/counters/increment_test")

assert result.status_code == status.HTTP_201_CREATED

assert result.data == b'{"increment_test":0}\n'

result = client.put("/counters/increment_test")

assert result.status_code == status.HTTP_200_OK

assert result.data == b'{"increment_test":1}\n'
```

However, this does not cover the case where a client tries to update a counter that does not exist. For this, a new test can be made called test_update_nonexistant_counter().



```python
result = client.put("/counters/nonexistant_test")

assert result.status_code == status.HTTP_204_NO_CONTENT
```

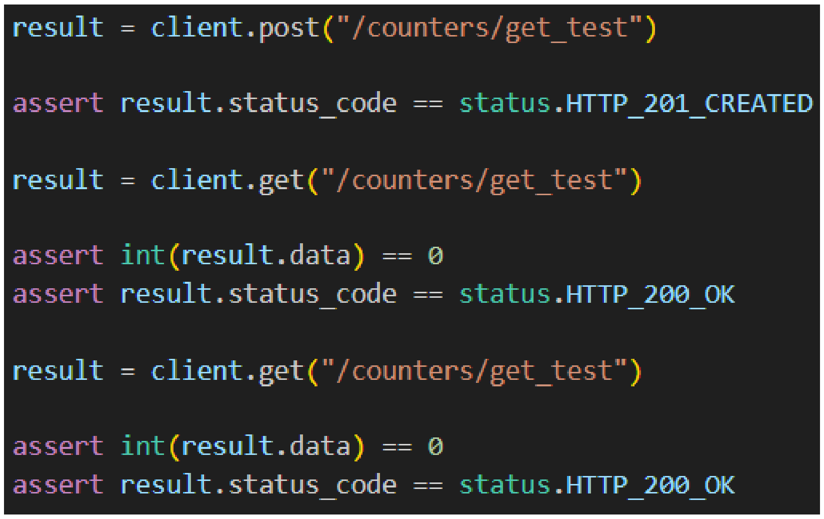
Both of these tests are RED and will fail. The next step is to implement the put method to ensure that both cases can pass. The following code segment is my implementation of the .put method. The code checks if the counter exists, and if it does, it increments the counter by 1.



```python
@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} does not exist"}, status.HTTP_204_NO_CONTENT

    COUNTERS[name] += 1

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

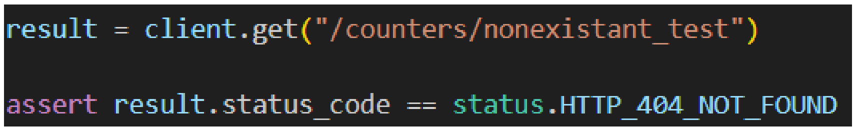
Now, running pytest results in green tests as well as 100% coverage.

After implementing the PUT request, the next task is to implement a GET request. A GET request requires an existing entry, so the test will need to create a counter and then try to get the value several times to make sure the counter is not changed when calling get.



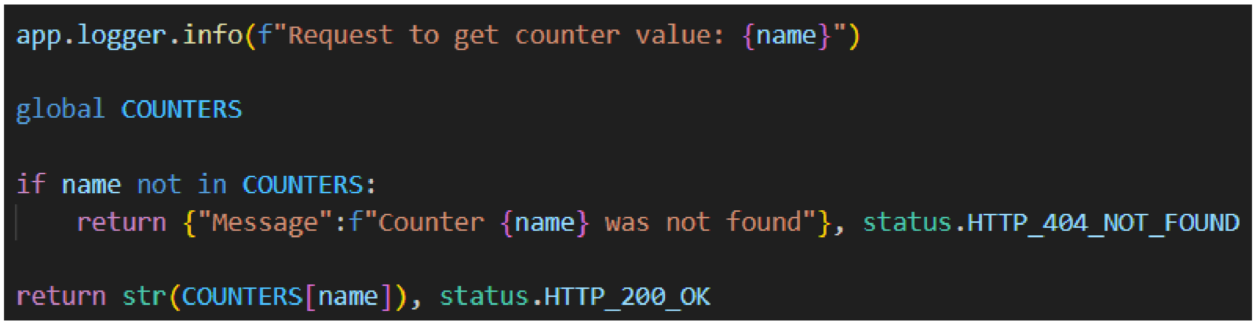
```python
result = client.post("/counters/get_test")

assert result.status_code == status.HTTP_201_CREATED

result = client.get("/counters/get_test")

assert int(result.data) == 0
assert result.status_code == status.HTTP_200_OK

result = client.get("/counters/get_test")

assert int(result.data) == 0
assert result.status_code == status.HTTP_200_OK
```

This does not cover the case where a GET request is made on a counter that does not exist, so this additional test case must be implemented.



```python
result = client.get("/counters/nonexistant_test")

assert result.status_code == status.HTTP_404_NOT_FOUND
```

Both of these test will fail, so the following code must be implemented to provide GET request function. It checks if the counter exists, and if it does, it returns the value of the counter as a string in the response.



```python
app.logger.info(f"Request to get counter value: {name}")

global COUNTERS

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

return str(COUNTERS[name]), status.HTTP_200_OK
```

With this, the GET method is added to counter.py. These are all the steps required in task 5.