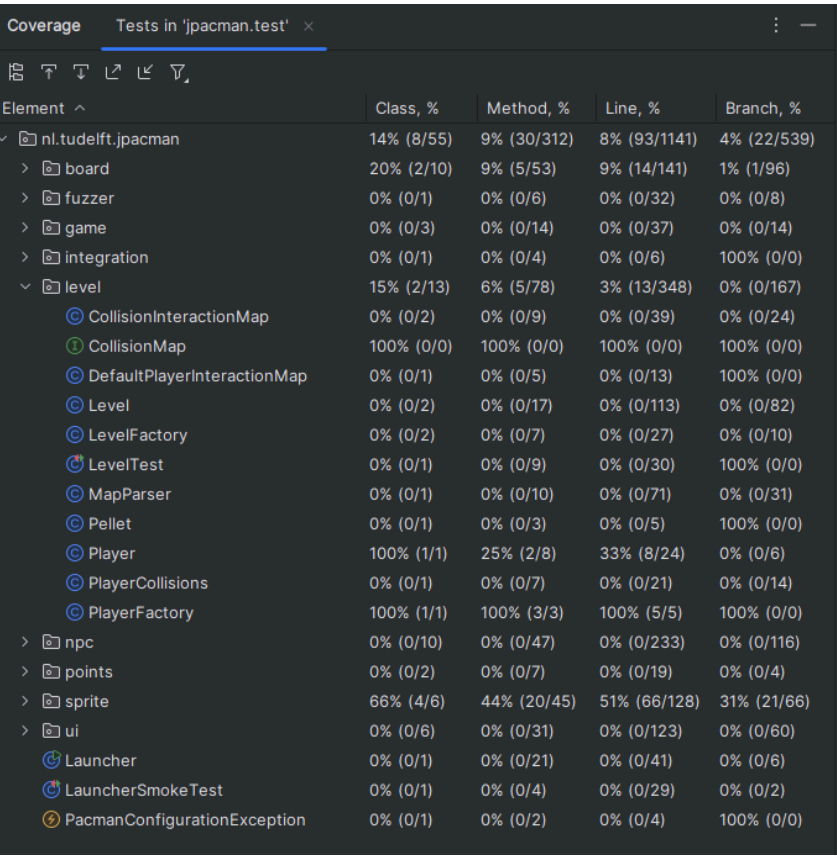
Nathan Nguyen
14 September 2024

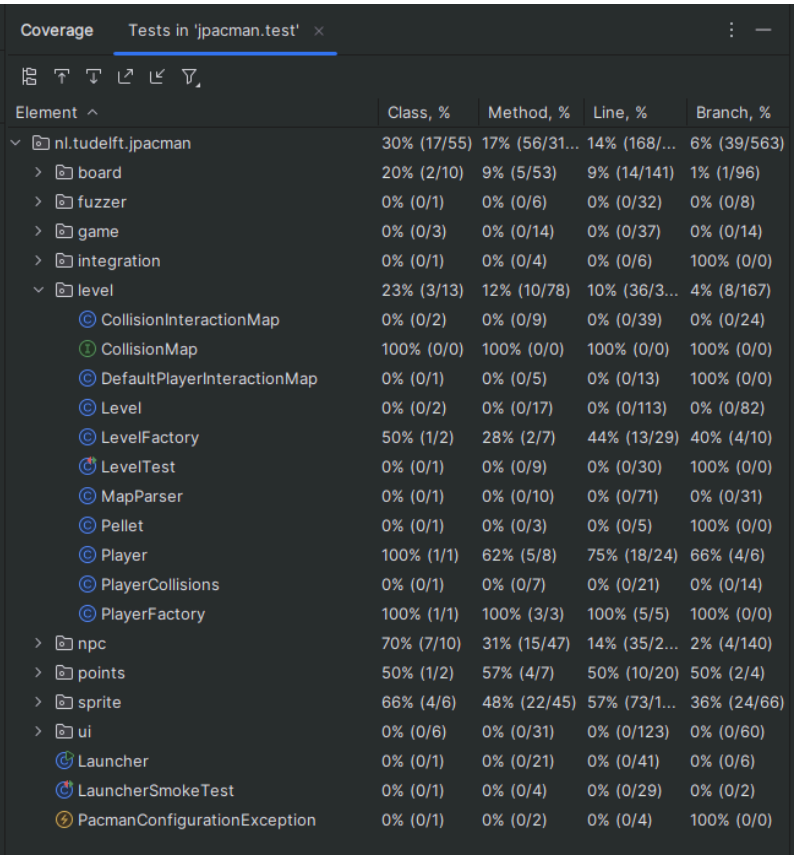
Fork Repository: https://github.com/nnguyen144/Main

IntelliJ



Coverage | Tests in 'jpacman.test'

Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	14% (8/55)	9% (30/312)	8% (93/1141)	4% (22/539)
board	20% (2/10)	9% (5/53)	9% (14/141)	1% (1/96)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
level	15% (2/13)	6% (5/78)	3% (13/348)	0% (0/167)
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/39)	0% (0/24)
CollisionMap	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/13)	100% (0/0)
Level	0% (0/2)	0% (0/17)	0% (0/113)	0% (0/82)
LevelFactory	0% (0/2)	0% (0/7)	0% (0/27)	0% (0/10)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)	100% (0/0)
MapParser	0% (0/1)	0% (0/10)	0% (0/71)	0% (0/31)
Pellet	0% (0/1)	0% (0/3)	0% (0/5)	100% (0/0)
Player	100% (1/1)	25% (2/8)	33% (8/24)	0% (0/6)
PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)	0% (0/14)
PlayerFactory	100% (1/1)	100% (3/3)	100% (5/5)	100% (0/0)
npc	0% (0/10)	0% (0/47)	0% (0/233)	0% (0/116)
points	0% (0/2)	0% (0/7)	0% (0/19)	0% (0/4)
sprite	66% (4/6)	44% (20/45)	51% (66/128)	31% (21/66)
ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)	0% (0/6)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)	0% (0/2)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)



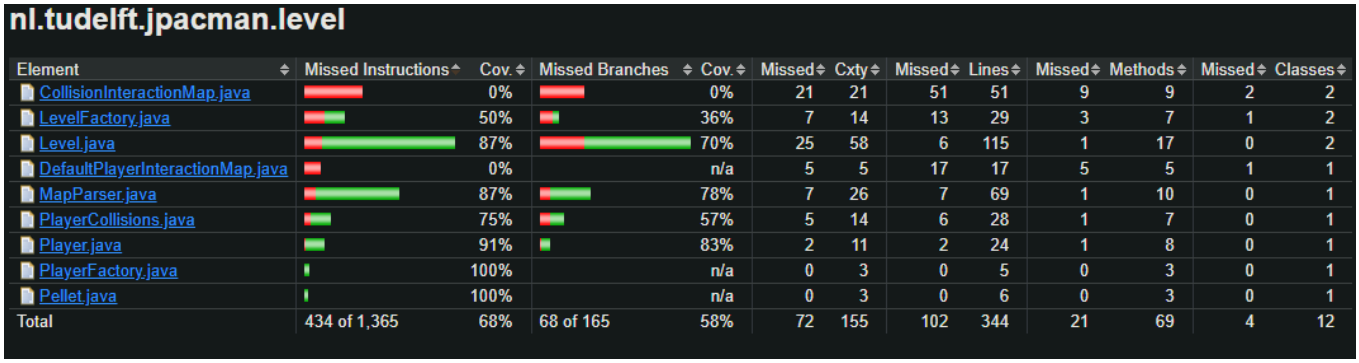
Coverage | Tests in 'jpacman.test'

Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	30% (17/55)	17% (56/31...	14% (168/...	6% (39/563)
board	20% (2/10)	9% (5/53)	9% (14/141)	1% (1/96)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
level	23% (3/13)	12% (10/78)	10% (36/3...	4% (8/167)
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/39)	0% (0/24)
CollisionMap	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/13)	100% (0/0)
Level	0% (0/2)	0% (0/17)	0% (0/113)	0% (0/82)
LevelFactory	50% (1/2)	28% (2/7)	44% (13/29)	40% (4/10)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)	100% (0/0)
MapParser	0% (0/1)	0% (0/10)	0% (0/71)	0% (0/31)
Pellet	0% (0/1)	0% (0/3)	0% (0/5)	100% (0/0)
Player	100% (1/1)	62% (5/8)	75% (18/24)	66% (4/6)
PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)	0% (0/14)
PlayerFactory	100% (1/1)	100% (3/3)	100% (5/5)	100% (0/0)
npc	70% (7/10)	31% (15/47)	14% (35/2...	2% (4/140)
points	50% (1/2)	57% (4/7)	50% (10/20)	50% (2/4)
sprite	66% (4/6)	48% (22/45)	57% (73/1...	36% (24/66)
ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)	0% (0/6)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)	0% (0/2)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)

Before 3 Unit Tests

After 3 Unit Tests

JaCoCo



nl.tudelft.jpacman.level

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CollisionInteractionMap.java		0%		0%	21	21	51	51	9	9	2	2
LevelFactory.java		50%		36%	7	14	13	29	3	7	1	2
Level.java		87%		70%	25	58	6	115	1	17	0	2
DefaultPlayerInteractionMap.java		0%		n/a	5	5	17	17	5	5	1	1
MapParser.java		87%		78%	7	26	7	69	1	10	0	1
PlayerCollisions.java		75%		57%	5	14	6	28	1	7	0	1
Player.java		91%		83%	2	11	2	24	1	8	0	1
PlayerFactory.java		100%		n/a	0	3	0	5	0	3	0	1
Pellet.java		100%		n/a	0	3	0	6	0	3	0	1
Total	434 of 1,365	68%	68 of 165	58%	72	155	102	344	21	69	4	12

Questions:

- Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?
  The coverage results provided by JaCoCo are not similar to the ones I received from IntelliJ. For example, for the *level* package, JaCoCo has 8/12 for classes (meaning I missed only 4), while IntelliJ has 3/13. JaCoCo has 48/69 for methods while IntelliJ has 10/78. JaCoCo has 242/344 for lines while IntelliJ has 36/350. Overall, the results are not similar because JaCoCo's coverage results are much higher than IntelliJ's. I believe the discrepancy has to do with each coverage program calculating the number of lines/methods/etc… differently.

- Did you find helpful the source code visualization from JaCoCo on uncovered branches?
  I did find the source code visualization from JaCoCo on uncovered branches to be helpful. For example, when I wrote the unit test for the createGhost() method in the LevelFactory class, JaCoCo displayed in green the lines that were covered in the unit tests, and in red, the lines that were not accessed, which could help me realize if I missed any scenarios for that particular unit test.

```
Ghost createGhost() {
    ghostIndex++;
    ghostIndex %= GHOSTS;
    switch (ghostIndex) {
        case BLINKY:
            return ghostFact.createBlinky();
        case INKY:
            return ghostFact.createInky();
        case PINKY:
            return ghostFact.createPinky();
        case CLYDE:
            return ghostFact.createClyde();
        default:
            return new RandomGhost(sprites.getGhostSprite(GhostColor.RED));
    }
}
```

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

I prefer JaCoCo's report simply because of the source code visualization that it provides, which allows me to see the scenarios I missed in my unit tests. Although IntelliJ's coverage abilities has the advantage of being included in the IDE, at the same time, I think IntelliJ's coverage UI is not great as it blocks a portion of the screen of the IDE, while JaCoCo is on a web browser which can be used on a second monitor.

Unit Tests:

```java
package nl.tudelft.jpacman.level;

import nl.tudelft.jpacman.npc.ghost.Blinky;
import nl.tudelft.jpacman.npc.ghost.Inky;
import nl.tudelft.jpacman.npc.ghost.Pinky;
import nl.tudelft.jpacman.npc.ghost.Clyde;
import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.points.PointCalculator;
import nl.tudelft.jpacman.points.PointCalculatorLoader;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertTrue;


public class LevelFactoryTest {
    PacManSprites pcSprites = new PacManSprites();  2 usages
    GhostFactory ghostFactory = new GhostFactory(pcSprites);  1 usage
    PointCalculatorLoader pcl = new PointCalculatorLoader();  1 usage
    PointCalculator pc = pcl.load();  1 usage

    @Test
    void test_createGhost(){
        LevelFactory levelFactory = new LevelFactory(pcSprites, ghostFactory, pc);
        assertTrue(levelFactory.createGhost() instanceof Blinky);
        assertTrue(levelFactory.createGhost() instanceof Inky);
        assertTrue(levelFactory.createGhost() instanceof Pinky);
        assertTrue(levelFactory.createGhost() instanceof Clyde);
        assertTrue(levelFactory.createGhost() instanceof Blinky);
    }

}
```

src/main/java/nl/tudelft/jpacman/level/LevelFactory.createGhost

```java
package nl.tudelft.jpacman.level;

import nl.tudelft.jpacman.sprite.PacManSprites;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

public class PlayerTest {
    @Test
    void test_isAlive() {
        PacManSprites pcSprite = new PacManSprites();
        PlayerFactory factory = new PlayerFactory(pcSprite);
        Player pc = factory.createPacMan();
        Assertions.assertThat(pc.isAlive()).isEqualTo( expected: true);
    }
    @Test
    void test_getScore() {
        PacManSprites pcSprite = new PacManSprites();
        PlayerFactory factory = new PlayerFactory(pcSprite);
        Player pc = factory.createPacMan();
        Assertions.assertThat(pc.getScore()).isEqualTo( expected: 0);
        pc.addPoints(5);
        Assertions.assertThat(pc.getScore()).isEqualTo( expected: 5);
    }

    @Test
    void test_setAlive() {
        PacManSprites pcSprite = new PacManSprites();
        PlayerFactory factory = new PlayerFactory(pcSprite);
        Player pc = factory.createPacMan();
        pc.setAlive(false);
        Assertions.assertThat(pc.isAlive()).isEqualTo( expected: false);
        pc.setAlive(true);
        Assertions.assertThat(pc.isAlive()).isEqualTo( expected: true);
    }

}
```

src/main/java/nl/tudelft/jpacman/level/Player.getScore and
src/main/java/nl/tudelft/jpacman/level/Player.setAlive

Task 4 Unit Tests

```python
def test_create_all_accounts():
    """ Test creating multiple Accounts """
    for data in ACCOUNT_DATA:
        account = Account(**data)
        account.create()
    assert len(Account.all()) == len(ACCOUNT_DATA)

def test_create_an_account():
    """ Test Account creation using known data """
    rand = randrange(0, len(ACCOUNT_DATA))
    data = ACCOUNT_DATA[rand]  # get a random account
    account = Account(**data)
    account.create()
    assert len(Account.all()) == 1

def test_repr():
    """Test the representation of an account"""
    account = Account()
    account.name = "Foo"
    assert str(account) == "<Account 'Foo'>"

def test_to_dict():
    """ Test account to dict """
    rand = randrange(0, len(ACCOUNT_DATA))  # Generate a random index
    data = ACCOUNT_DATA[rand]  # get a random account
    account = Account(**data)
    result = account.to_dict()

    assert account.name == result["name"]
    assert account.email == result["email"]
    assert account.phone_number == result["phone_number"]
    assert account.disabled == result["disabled"]
    assert account.date_joined == result["date_joined"]

def test_from_dict():
    """ Test account from dict """
    data = {
        "name": "Jimmy",
        "email": "jimmy144@gmail.com",
        "phone_number": "206-144-1444",
        "disabled": False,
        "date_joined": db.DateTime("2020-01-04T00:00:00.000Z")
    }
    account = Account()
    account.from_dict(data)

    assert account.name == data["name"]
    assert account.email == data["email"]
```

```
    assert account.phone_number == data["phone_number"]
    assert account.disabled == data["disabled"]
    assert account.date_joined == data["date_joined"]

def test_update_id():
    data = { "id": 1 }
    account = Account(**data)
    with patch('models.account.db.session.commit') as mock:
        account.update()
        mock.assert_called_once()

def test_update_no_id():
    data = { "id": None }
    account = Account(**data)
    try:
        account.update()
    except:
        assert DataValidationError

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

def test_find_success():
    data = { "id": 1 }
    account = Account(**data)
    account.create()
    find = Account.find(account_id=1)
    assert find == account

def test_find_failure():
    find = Account.find(account_id=1)
    assert find is None
```

100% Coverage

```
---------- coverage: platform win32, python 3.12.6-final-0 -----------
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models\__init__.py       7      0   100%
models\account.py       40      0   100%
--------------------------------------------------
TOTAL                   47      0   100%
```

For the test_from_dict() test, I created a ‘data’ dictionary that will be used later on in the asserts, and then I created an account without any data. I then used account.from_dict(data), which should in theory take each key-value pair in the dict and put them in the data section of account. I used assert to compare the values of ‘account’ and ‘data’ for every key to make sure from_dict works properly.

For the test_update_id() test, I split it into two tests to cover two scenarios: one where there is an ID, and one where there isn’t. For the former, I created an account with an ID and mocked doing the method account.update(). The following line, mock.assert_called_once(), will pass if the account.update() method calls a db.session.commit() (due to the line “with patch('models.account.db.session.commit') as mock:”), and will fail if account.update() did not call the session commit.

For the test_update_no_id() test, I created an account with no ID and used a “try-except”, where I tried to do account.update(), which should create a DataValidationError (according to the code of update(self) which creates an error when the account has no ID), and thus the “except” part of the “try-except” occurs, in which I asserted that a DataValidationError occurred.

For test_delete(), I created an account (using random data) and asserted that there exists exactly 1 account, using “assert len(Account.all()) == 1”. I then used account.delete(), and so if the delete had worked, then that one account would be deleted leaving there to be 0 accounts, which I asserted with len(Account.all()) == 1.

For test_find(), I split it into two tests: one test for when the account with a specific ID does exist, and one where the account with a specific ID doesn’t. For the former, I created an account with id=1 and then used “find = Account.find(account_id=1)”. If the find() method works correctly, then the account should be returned, which I asserted with “assert find == account”.

For the test_find() scenario where the account with the specific ID doesn’t exist, I simply used “find = Account.find(account_id=1)”, however there is no account with an id of 1. Thus, according to the find() method, a None should be returned, which I asserted with “assert Find is None”.

Task 5 Methods

```
from flask import Flask
from . import status

app = Flask(__name__)
COUNTERS = {}

# 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 called
# on this function is "POST".
@app.route('/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

@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if not name in COUNTERS:
        return {"Message":f"Counter {name} does not exist"},
status.HTTP_404_NOT_FOUND
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK

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

Using create_counter(name) as a reference, I created update_counter(name) and read_counter(name). The former is a method for PUT, which updates a counter by incrementing by 1. Thus, the name for the counter needs to already exist, and if it doesn't, a 404_NOT_FOUND is produced. If the name exists, then its counter is incremented and a 200_OK is produced. real_counter(name) is similar to update_counter(name), except an increment of the counter is not needed. Getting access to the counter value is up to the caller of read_counter.

Task 5 Unit Tests

```
import pytest

# we need to import the unit under test - counter
from src.counter import app, create_counter

# we need to import the file that contains the status codes
from src import status

@pytest.fixture()
def client():
    return app.test_client()

@pytest.mark.usefixtures("client")
class TestCounterEndPoints:
    """Test cases for Counter-related endpoints"""

    def test_create_a_counter(self, client):
        """It should create a counter"""
        result = client.post('/counters/foo')
        assert result.status_code == status.HTTP_201_CREATED

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

def test_update_a_counter(self, client):
    """It should update a counter"""
    result = client.post('/counters/swag')
    assert result.status_code == status.HTTP_201_CREATED
    num = client.get('/counters/swag')
    assert num.status_code == status.HTTP_200_OK
    result = client.put('/counters/swag')
    assert result.status_code == status.HTTP_200_OK
    result = client.get('/counters/swag')
    assert result.status_code == status.HTTP_200_OK
    assert result.get_json()["swag"] == num.get_json()["swag"] + 1

def test_update_a_counter_fail(self, client):
    result = client.put('/counters/swag3')
    assert result.status_code == status.HTTP_404_NOT_FOUND
```

```python
def test_read_a_counter(self, client):
    """It should read a counter"""
    result = client.post('/counters/swag2')
    assert result.status_code == status.HTTP_201_CREATED
    result = client.get('/counters/swag2')
    assert result.status_code == status.HTTP_200_OK
    assert result.get_json()["swag2"] == 0

def test_read_a_counter_fail(self, client):
    result = client.get('/counters/swag4')
    assert result.status_code == status.HTTP_404_NOT_FOUND
```

test_update_a_counter(self, client) was burdensome. For testing, I first created a counter (using client.post in the first line) and checked that a counter was created (using the assert in the second line). Then I used client.get and put the result in a new variable (num) in order to store the initial value of the counter. However, this proved to be an issue as initially, I didn't know how to retrieve that value. I adjusted the return of read_counter(name), which led to a 500 INTERNAL SERVER ERROR, which meant the status code was 500 and thus failed the assert for the status code to be equal to 200. After I fixed the return to be the same as the other methods, I updated the counter using 'put' and then retrieved the counter using 'get', all while checking the status codes. At the very end, I had to assert that the value of the counter after the update was 1 more than the value before the encounter. To do this, I had to take the variables holding the client.get's (result, which held the post-update counter, and num, which held the pre-update counter), and used .get_json() to get the value of the counters.

test_read_a_counter(self, client) was simply creating a counter using "POST" and getting the counter value with "GET", all while checking status codes. Finally, I checked that the newly made counter has a value of 0 using an assert.

For the fail scenarios for the previous two tests, I simply used "PUT" and "GET" respectively without creating a counter first, and if a counter does not exist, then the counter cannot be retrieved nor updated, which gives a 404_NOT_FOUND. I asserted that the status code is equal to this 404.

100% Coverage

```
---------- coverage: platform win32, python 3.12.6-final-0 -----------
Name              Stmts   Miss  Cover   Missing
-------------------------------------------------
src\__init__.py       0      0   100%
src\counter.py       24      0   100%
src\status.py         6      0   100%
-------------------------------------------------
TOTAL                30      0   100%
```