Miguel Alvarez Uriarte

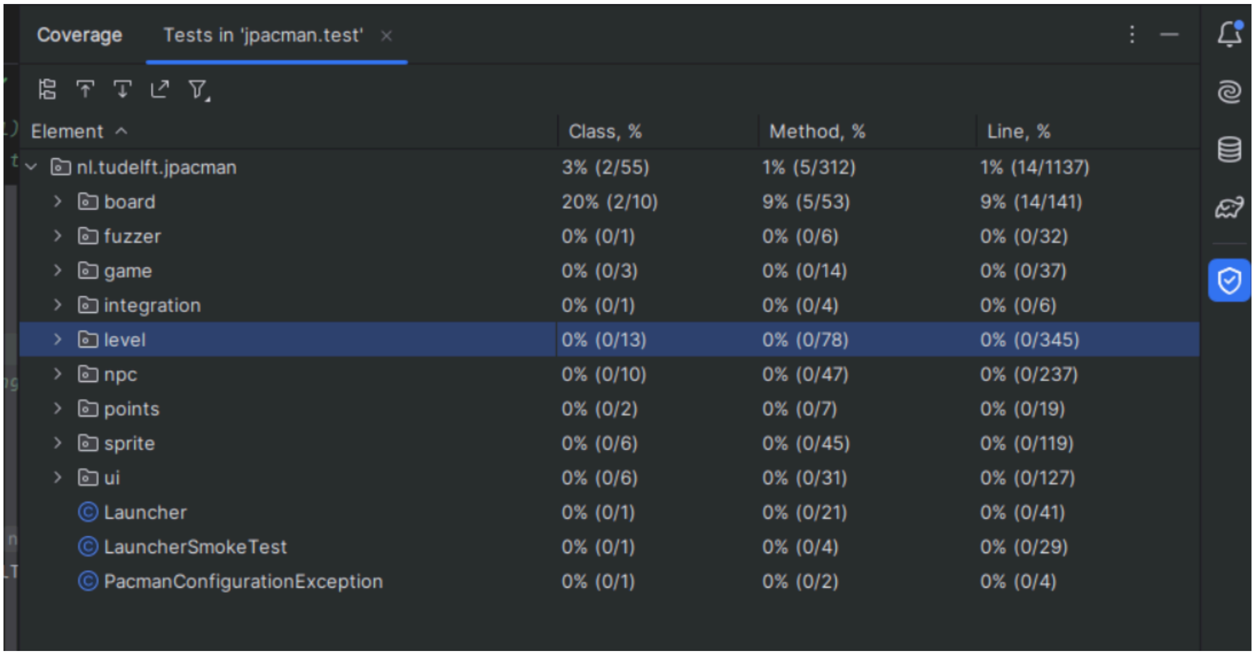
CS 472

Software Testing Lab

2/05/2024

Fork Repo Link: https://github.com/BitsyBirb/Barbell.git

## Task 1: Jpacman Code Coverage



| 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) |
| PacmanConfigurationException | 0% (0/1) | 0% (0/2) | 0% (0/4) |

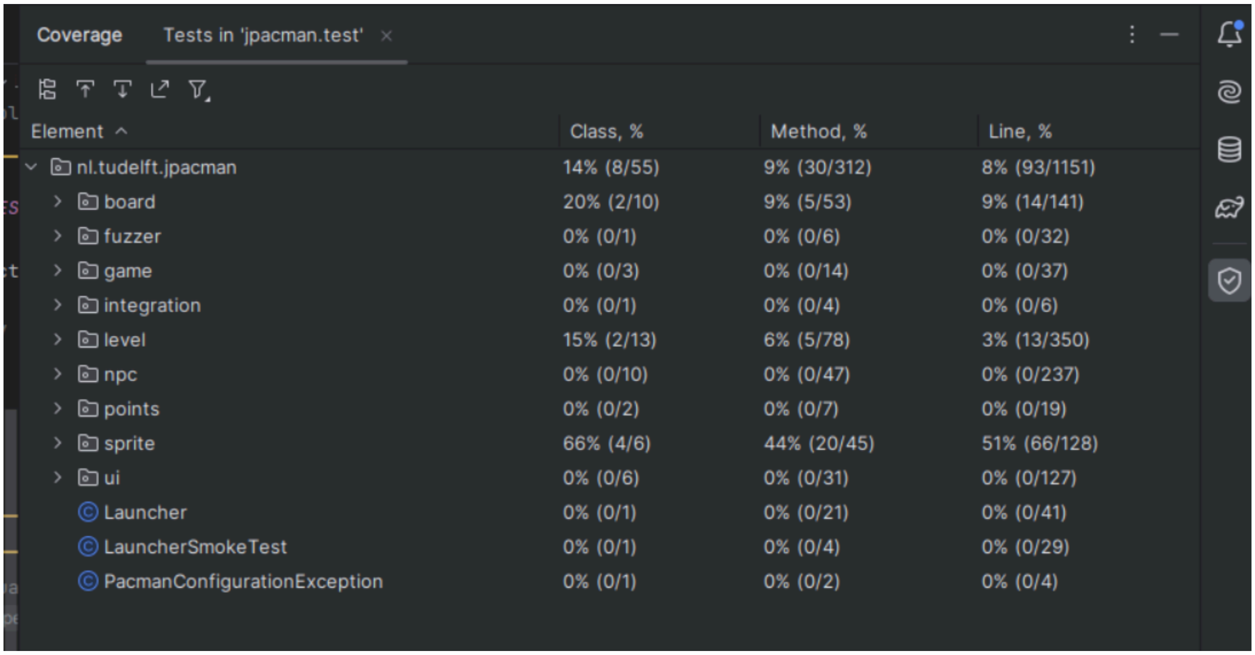
Figure 1: Code coverage using gradlew before adding more unit tests

As seen above, I can’t say this is the best code coverage at all. We only test the board package itself, and even then it has less than 20% coverage across all three sections which leaves a lot to be desired. Essentially none of the code has been tested whatsoever.

## Task 2: Increasing coverage on JPacman

We are creating some simple unit tests to increase code coverage.

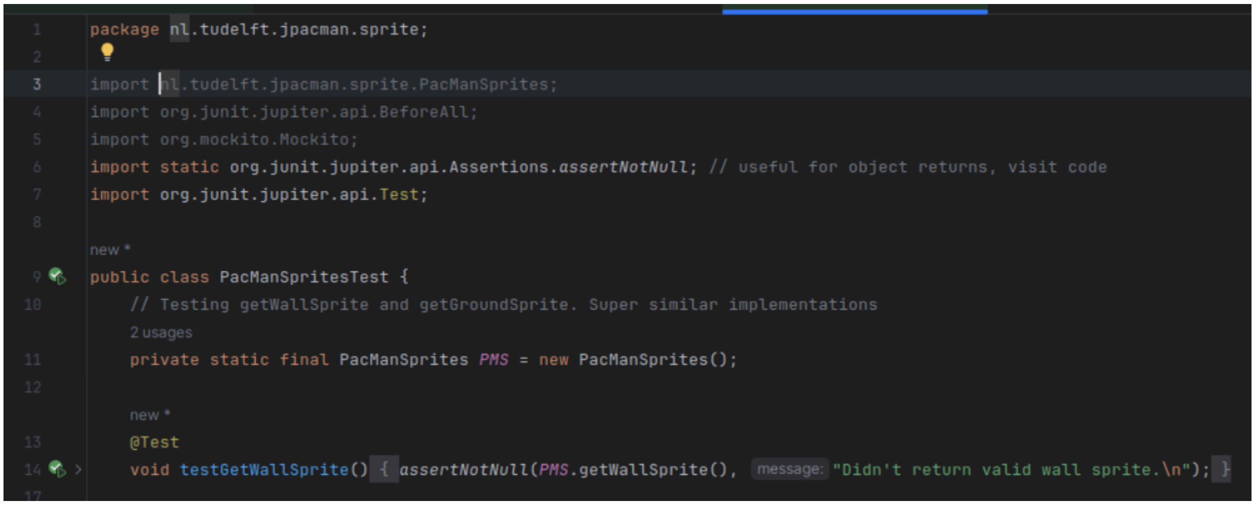
**isAlive() coverage:** Level coverage went up from 0 to 15% class coverage, 6% method coverage, and 3% line coverage.



| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| nl.tudelft.jpacman | 14% (8/55) | 9% (30/312) | 8% (93/1151) |
| 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) | 6% (5/78) | 3% (13/350) |
| npc | 0% (0/10) | 0% (0/47) | 0% (0/237) |
| points | 0% (0/2) | 0% (0/7) | 0% (0/19) |
| sprite | 66% (4/6) | 44% (20/45) | 51% (66/128) |
| 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) |

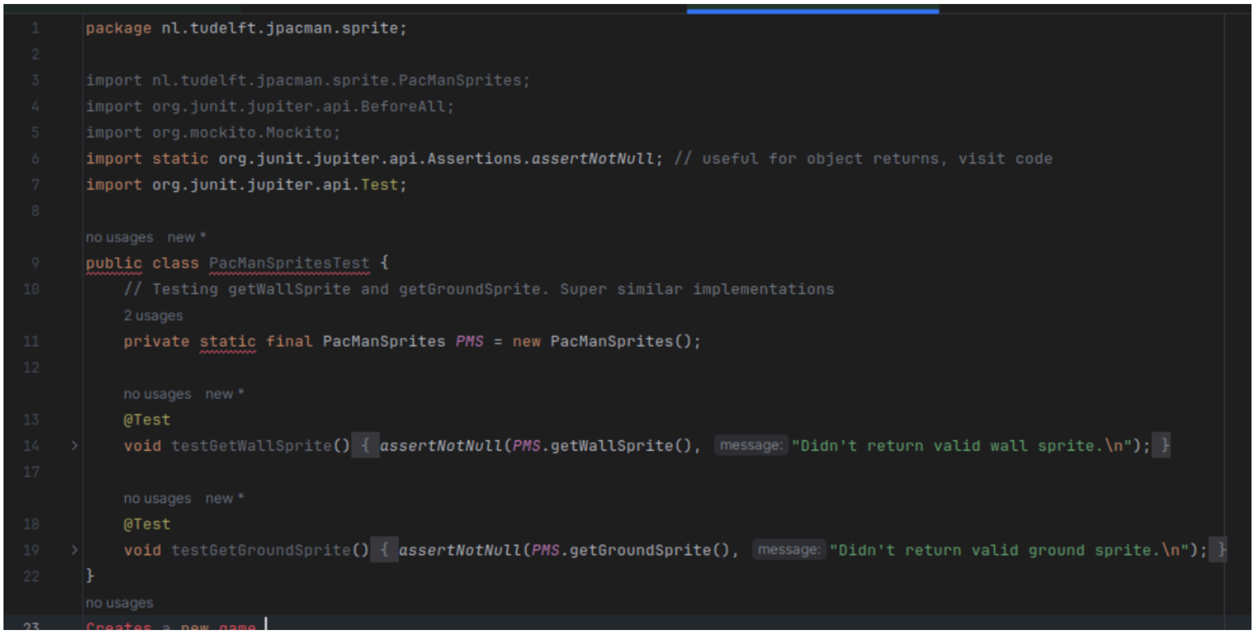
Figure 2: Code coverage after implementing isAlive() unit testing

*Note that this coverage is also that of before the unit tests for task 2.1*



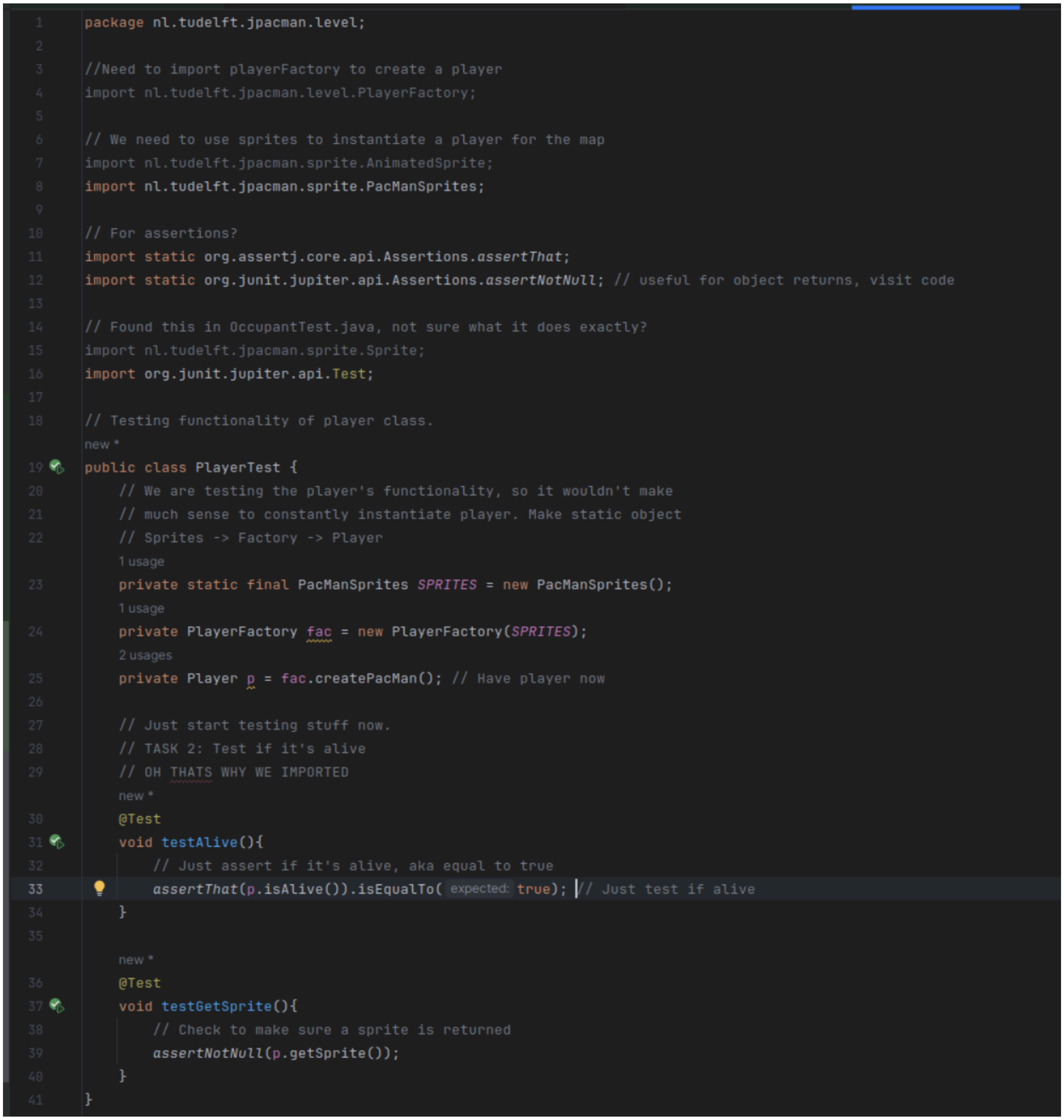
```java
package nl.tudelft.jpacman.sprite;

import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.BeforeAll;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.assertNotNull; // useful for object returns, visit code
import org.junit.jupiter.api.Test;

new *
public class PacManSpritesTest {
    // Testing getWallSprite and getGroundSprite. Super similar implementations
    2 usages
    private static final PacManSprites PMS = new PacManSprites();

    new *
    @Test
    void testGetWallSprite() { assertNotNull(PMS.getWallSprite(), message: "Didn't return valid wall sprite.\n"); }
```

Figure 3: Unit test for src/main/java/nl/tudelft/jpacman/sprite/PacManSprites.getWallSprite()



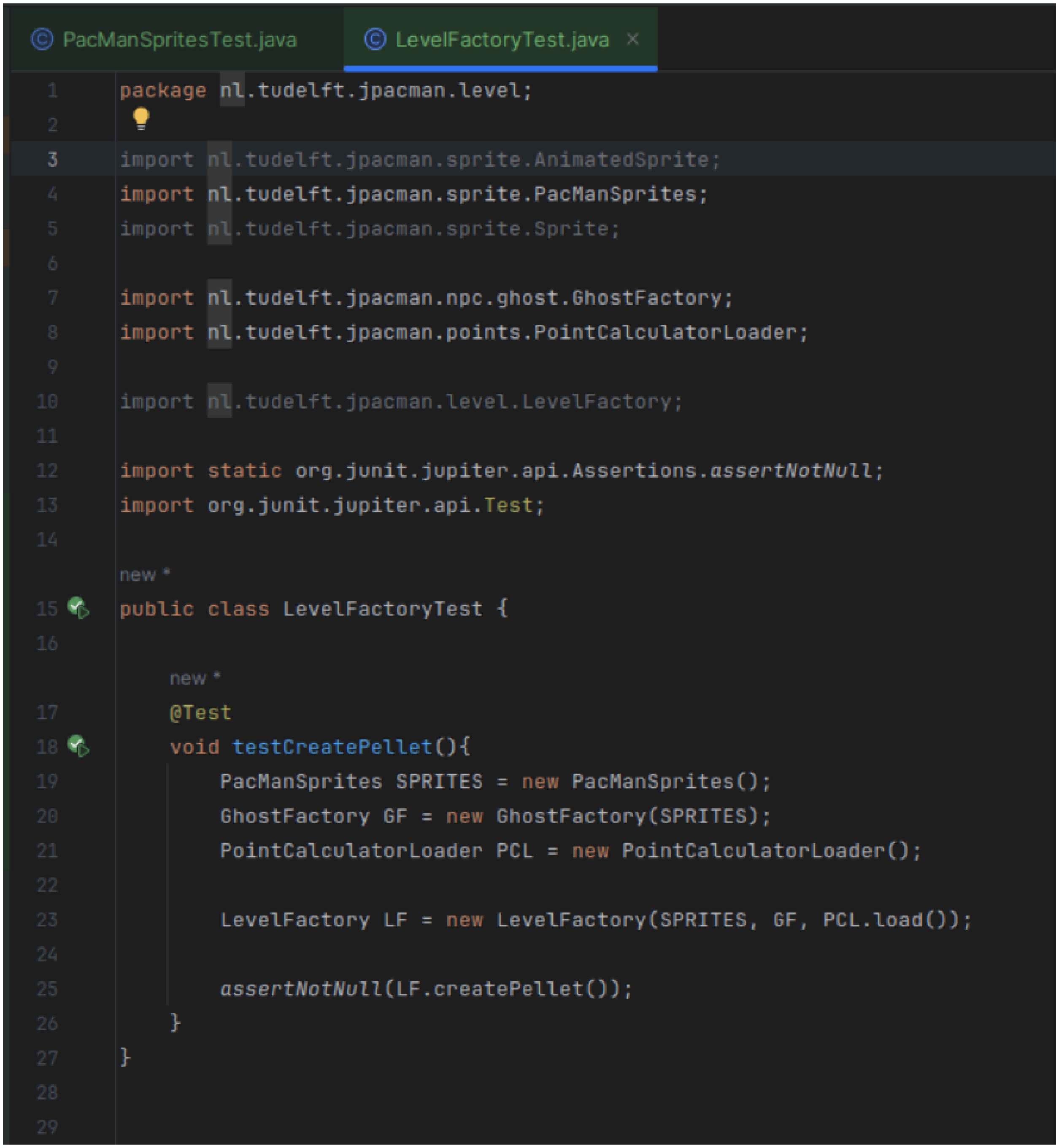
```java
package nl.tudelft.jpacman.sprite;

import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.BeforeAll;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.assertNotNull; // useful for object returns, visit code
import org.junit.jupiter.api.Test;

no usages   new *
public class PacManSpritesTest {
    // Testing getWallSprite and getGroundSprite. Super similar implementations
    2 usages
    private static final PacManSprites PMS = new PacManSprites();

    no usages   new *
    @Test
    void testGetWallSprite() { assertNotNull(PMS.getWallSprite(),  message: "Didn't return valid wall sprite.\n"); }

    no usages   new *
    @Test
    void testGetGroundSprite() { assertNotNull(PMS.getGroundSprite(),  message: "Didn't return valid ground sprite.\n"); }
}
no usages
Creates a new game
```

Figure 4: Unit test for src/main/java/nl/tudelft/jpacman/sprite/PacManSprites.getGroundSprite()



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

//Need to import playerFactory to create a player
import nl.tudelft.jpacman.level.PlayerFactory;

// We need to use sprites to instantiate a player for the map
import nl.tudelft.jpacman.sprite.AnimatedSprite;
import nl.tudelft.jpacman.sprite.PacManSprites;

// For assertions?
import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertNotNull; // useful for object returns, visit code

// Found this in OccupantTest.java, not sure what it does exactly?
import nl.tudelft.jpacman.sprite.Sprite;
import org.junit.jupiter.api.Test;

// Testing functionality of player class.
new *
public class PlayerTest {
    // We are testing the player's functionality, so it wouldn't make
    // much sense to constantly instantiate player. Make static object
    // Sprites -> Factory -> Player
    1 usage
    private static final PacManSprites SPRITES = new PacManSprites();
    1 usage
    private PlayerFactory fac = new PlayerFactory(SPRITES);
    2 usages
    private Player p = fac.createPacMan(); // Have player now

    // Just start testing stuff now.
    // TASK 2: Test if it's alive
    // OH THATS WHY WE IMPORTED
    new *
    @Test
    void testAlive(){
        // Just assert if it's alive, aka equal to true
        assertThat(p.isAlive()).isEqualTo( expected: true); // Just test if alive
    }

    new *
    @Test
    void testGetSprite(){
        // Check to make sure a sprite is returned
        assertNotNull(p.getSprite());
    }
}
```

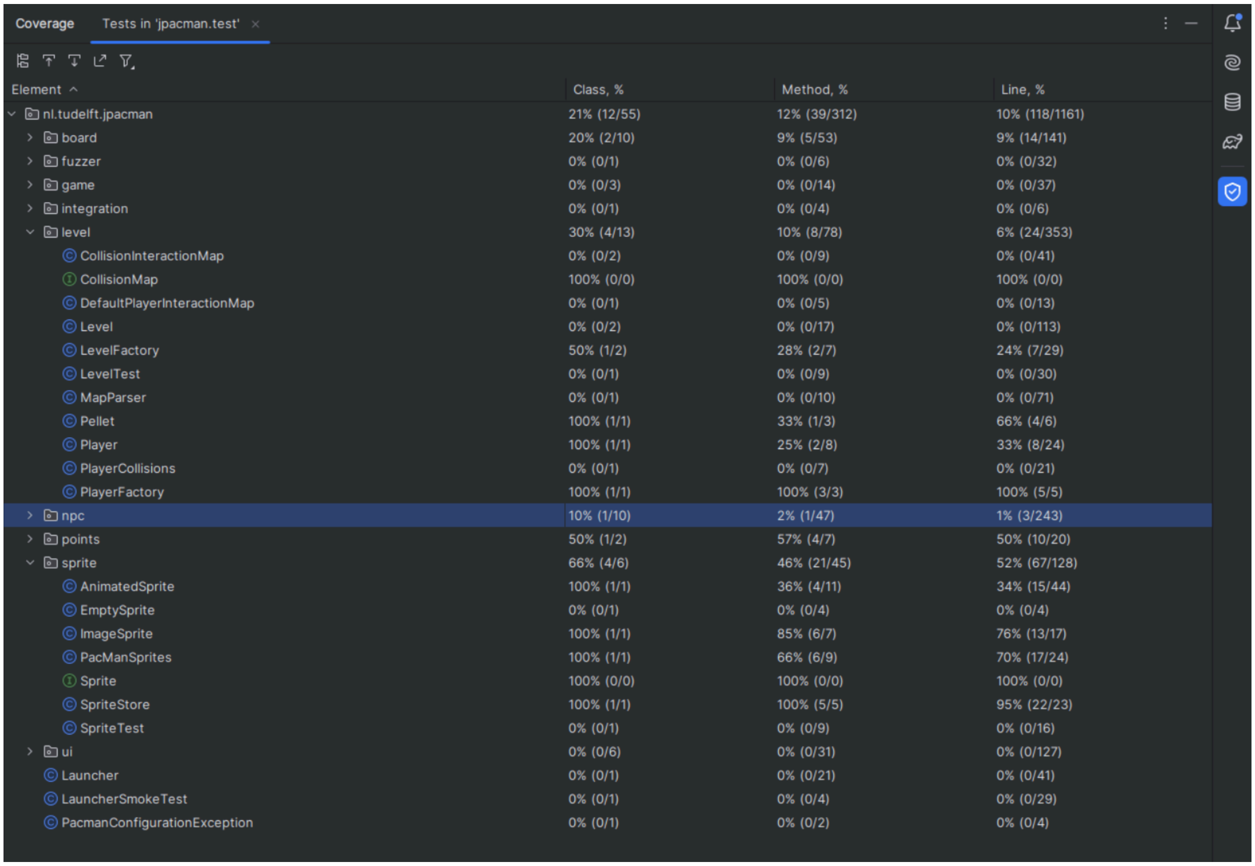
Figure 5: Unit test for src/main/java/nl/tudelft/jpacman/level/Player.getSprite()



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

import nl.tudelft.jpacman.sprite.AnimatedSprite;
import nl.tudelft.jpacman.sprite.PacManSprites;
import nl.tudelft.jpacman.sprite.Sprite;

import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.points.PointCalculatorLoader;

import nl.tudelft.jpacman.level.LevelFactory;

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

public class LevelFactoryTest {

    @Test
    void testCreatePellet(){
        PacManSprites SPRITES = new PacManSprites();
        GhostFactory GF = new GhostFactory(SPRITES);
        PointCalculatorLoader PCL = new PointCalculatorLoader();

        LevelFactory LF = new LevelFactory(SPRITES, GF, PCL.load());

        assertNotNull(LF.createPellet());
    }
}
```

Figure 6: Unit test for src/main/java/nl/tudelft/jpacman/level/LevelFactory.createPellet()

Coverage with all four unit tests implemented:



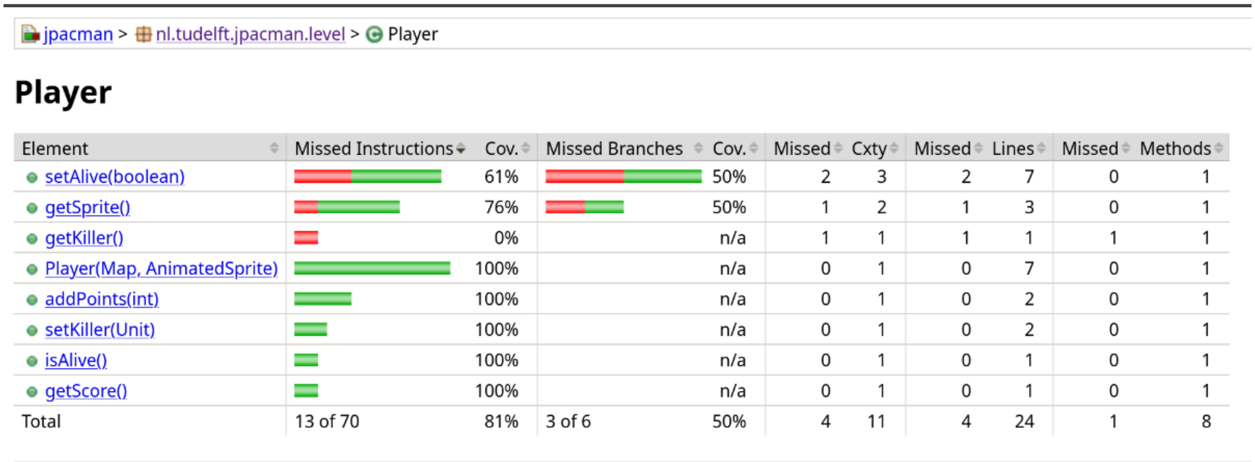
Coverage    Tests in 'jpacman.test'

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| nl.tudelft.jpacman | 21% (12/55) | 12% (39/312) | 10% (118/1161) |
| 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 | 30% (4/13) | 10% (8/78) | 6% (24/353) |
| CollisionInteractionMap | 0% (0/2) | 0% (0/9) | 0% (0/41) |
| CollisionMap | 100% (0/0) | 100% (0/0) | 100% (0/0) |
| DefaultPlayerInteractionMap | 0% (0/1) | 0% (0/5) | 0% (0/13) |
| Level | 0% (0/2) | 0% (0/17) | 0% (0/113) |
| LevelFactory | 50% (1/2) | 28% (2/7) | 24% (7/29) |
| LevelTest | 0% (0/1) | 0% (0/9) | 0% (0/30) |
| MapParser | 0% (0/1) | 0% (0/10) | 0% (0/71) |
| Pellet | 100% (1/1) | 33% (1/3) | 66% (4/6) |
| Player | 100% (1/1) | 25% (2/8) | 33% (8/24) |
| PlayerCollisions | 0% (0/1) | 0% (0/7) | 0% (0/21) |
| PlayerFactory | 100% (1/1) | 100% (3/3) | 100% (5/5) |
| npc | 10% (1/10) | 2% (1/47) | 1% (3/243) |
| points | 50% (1/2) | 57% (4/7) | 50% (10/20) |
| sprite | 66% (4/6) | 46% (21/45) | 52% (67/128) |
| 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) | 70% (17/24) |
| 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) |
| 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) |

Figure 7: IntelliJ code coverage with added unit tests.

After writing the four unit tests, the level and sprite package code coverage went up a few percent from their initial values of 0% in all three categories.

# Task 3: JaCoCo Report on JPacman



jpacman > nl.tudelft.jpacman.level > Player

## Player

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|---|---|---|---|---|---|---|---|---|---|---|
| setAlive(boolean) | 🟥🟩 | 61% | 🟥🟩 | 50% | 2 | 3 | 2 | 7 | 0 | 1 |
| getSprite() | 🟥🟩 | 76% | 🟥🟩 | 50% | 1 | 2 | 1 | 3 | 0 | 1 |
| getKiller() | 🟥 | 0% | | n/a | 1 | 1 | 1 | 1 | 1 | 1 |
| Player(Map, AnimatedSprite) | 🟩 | 100% | | n/a | 0 | 1 | 0 | 7 | 0 | 1 |
| addPoints(int) | 🟩 | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 |
| setKiller(Unit) | 🟩 | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 |
| isAlive() | 🟩 | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| getScore() | 🟩 | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| Total | 13 of 70 | 81% | 3 of 6 | 50% | 4 | 11 | 4 | 24 | 1 | 8 |

Figure 8: JaCoCo report

### Are coverage results similar to Intellij? Why or why not?

The coverages shown by both are not the same, and this is because both of the analysis tools focus on different things. JaCoCo spends more of its efforts focusing on and analyzing the branch and instruction coverage compared to IntelliJ's approach of checking out which lines are hit. Not only that, but JaCoCo covers individual methods as opposed to IntelliJ's higher-level approach, but that's not entirely related to the coverage itself.

### Was the source code visualization useful regarding uncovered branches?

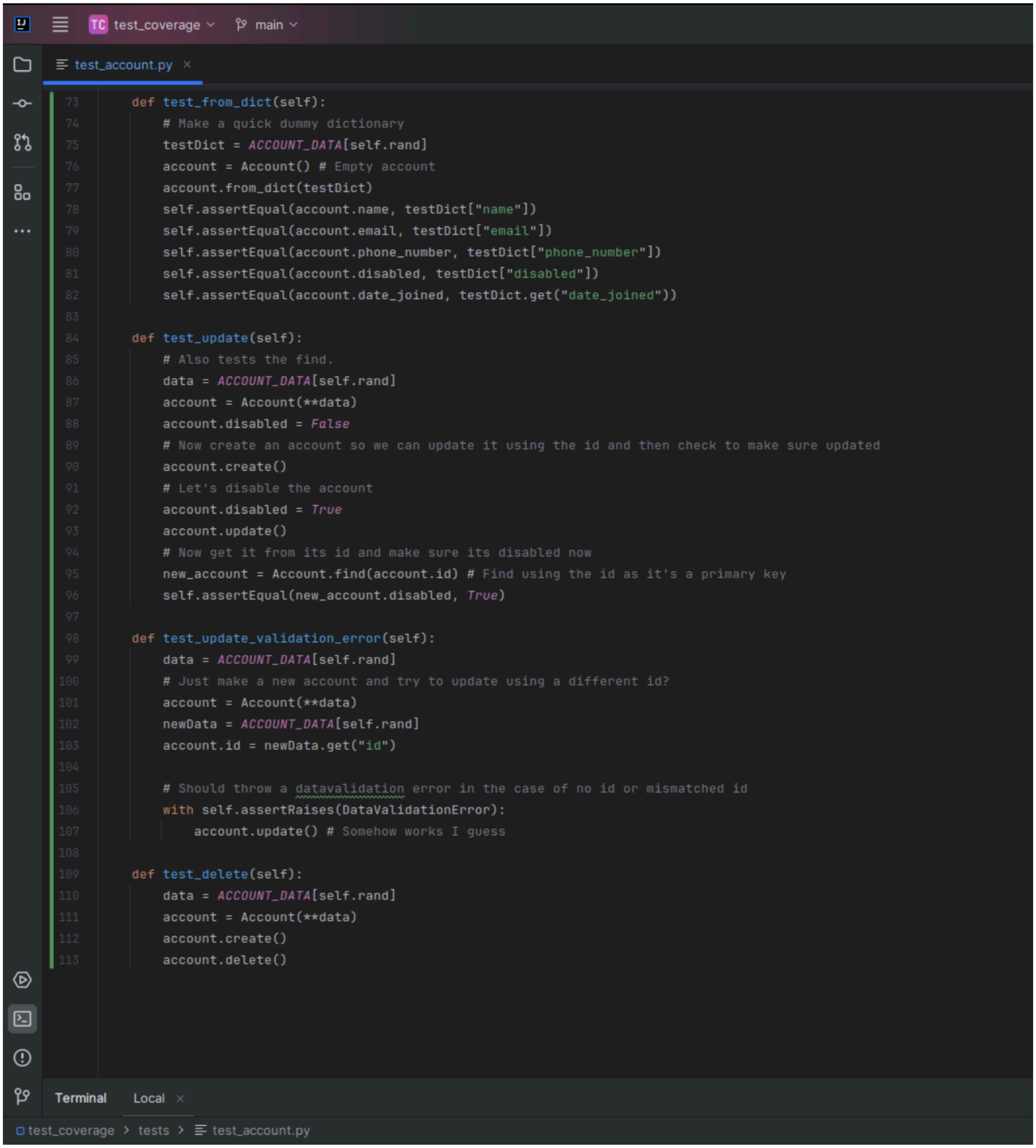
I found it to be rather useful, primarily because I can see more or less the flow of the unit tests as they go through methods. This helps me not only understand what was covered, but also how to design more unit tests to get higher coverage as I can more or less piece together what portions (and ultimately flow of executions) of the code wasn't tested.

### Which visualization did I prefer, and why?

I honestly preferred JaCoCo's because of the above question. Though there are more menus to navigate and ultimately get to a class or whatever it might be that I want to investigate on JaCoCo, the expanded results and more detailed reports on branch and instruction coverage help me to not just understand what was tested, but how to design tests that address any gaps present in said tests. I did like the high-level and simple way of the IntelliJ coverage report, however.

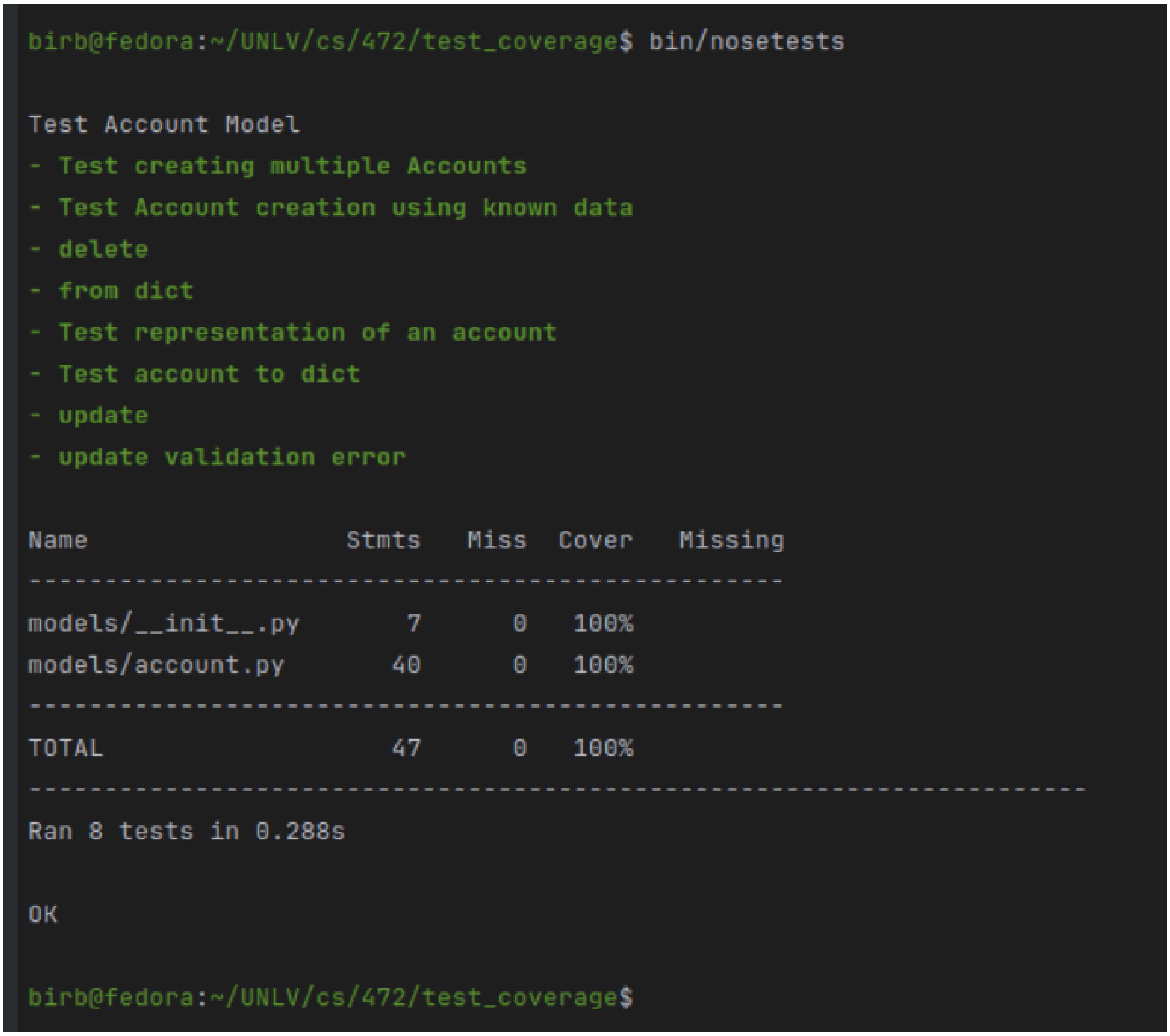
## Task 4: Working with Python Test Coverage

In this portion we start working with python unit testing, in particular implementing tests for account.py such that we can get full coverage as we initially do not.



```python
    def test_from_dict(self):
        # Make a quick dummy dictionary
        testDict = ACCOUNT_DATA[self.rand]
        account = Account() # Empty account
        account.from_dict(testDict)
        self.assertEqual(account.name, testDict["name"])
        self.assertEqual(account.email, testDict["email"])
        self.assertEqual(account.phone_number, testDict["phone_number"])
        self.assertEqual(account.disabled, testDict["disabled"])
        self.assertEqual(account.date_joined, testDict.get("date_joined"))

    def test_update(self):
        # Also tests the find.
        data = ACCOUNT_DATA[self.rand]
        account = Account(**data)
        account.disabled = False
        # Now create an account so we can update it using the id and then check to make sure updated
        account.create()
        # Let's disable the account
        account.disabled = True
        account.update()
        # Now get it from its id and make sure its disabled now
        new_account = Account.find(account.id) # Find using the id as it's a primary key
        self.assertEqual(new_account.disabled, True)

    def test_update_validation_error(self):
        data = ACCOUNT_DATA[self.rand]
        # Just make a new account and try to update using a different id?
        account = Account(**data)
        newData = ACCOUNT_DATA[self.rand]
        account.id = newData.get("id")

        # Should throw a datavalidation error in the case of no id or mismatched id
        with self.assertRaises(DataValidationError):
            account.update() # Somehow works I guess

    def test_delete(self):
        data = ACCOUNT_DATA[self.rand]
        account = Account(**data)
        account.create()
        account.delete()
```

Figure 9: Code snippets for testing account.py



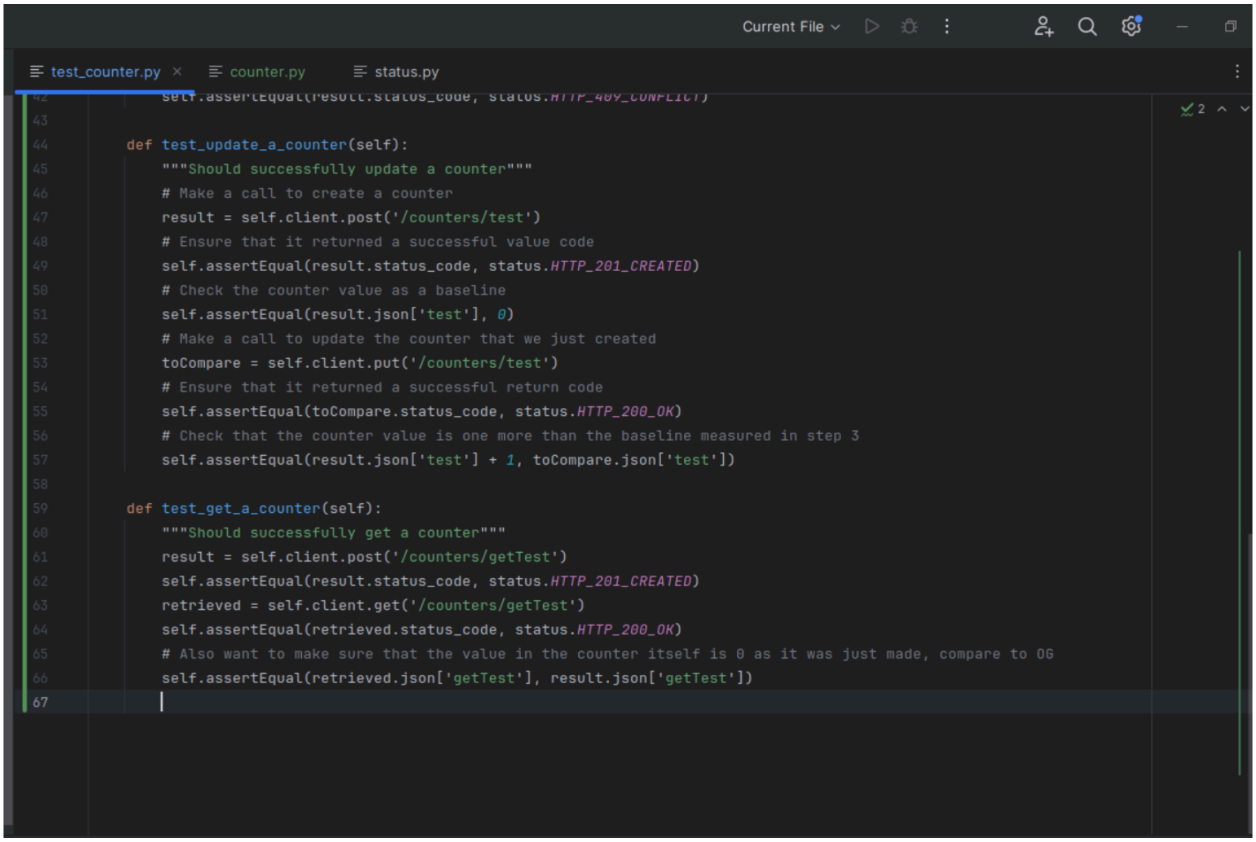
```
birb@fedora:~/UNLV/cs/472/test_coverage$ bin/nosetests

Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data
- delete
- from dict
- Test representation of an account
- Test account to dict
- update
- update validation error

Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models/__init__.py       7      0   100%
models/account.py       40      0   100%
--------------------------------------------------
TOTAL                   47      0   100%
----------------------------------------------------------------------
Ran 8 tests in 0.288s

OK

birb@fedora:~/UNLV/cs/472/test_coverage$
```

Figure 10: Running nosetests with full coverage

## Task 5: Test Driven Development

Red Phase:

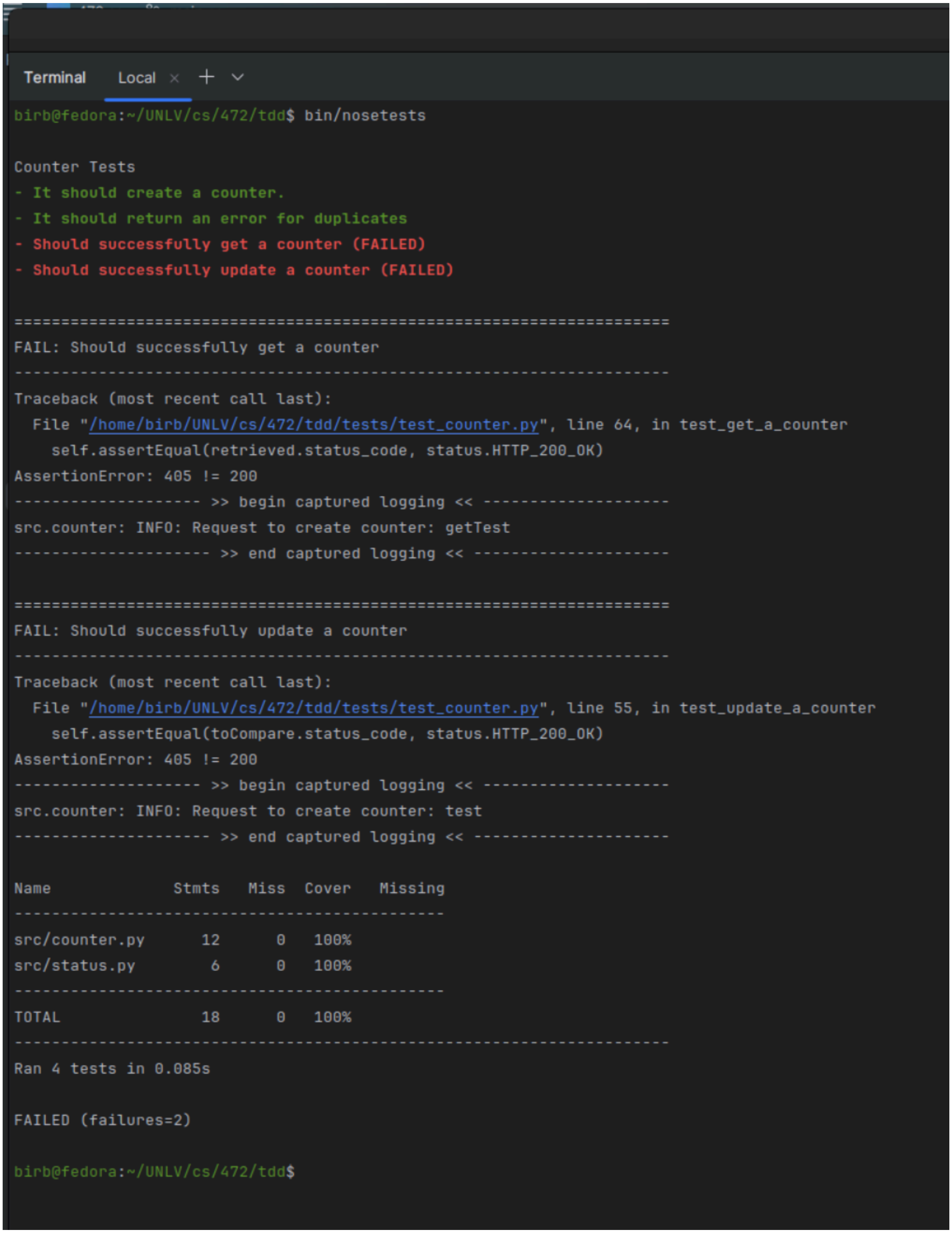
Our goal is to implement a route and method to update a counter, and another to get a counter. As such we write unit tests to check both of those, even before we implement them. This allows us to define expected behavior before we get to implementation and thus we can check as we work.



```python
        self.assertEqual(result.status_code, status.HTTP_409_CONFLICT)

    def test_update_a_counter(self):
        """Should successfully update a counter"""
        # Make a call to create a counter
        result = self.client.post('/counters/test')
        # Ensure that it returned a successful value code
        self.assertEqual(result.status_code, status.HTTP_201_CREATED)
        # Check the counter value as a baseline
        self.assertEqual(result.json['test'], 0)
        # Make a call to update the counter that we just created
        toCompare = self.client.put('/counters/test')
        # Ensure that it returned a successful return code
        self.assertEqual(toCompare.status_code, status.HTTP_200_OK)
        # Check that the counter value is one more than the baseline measured in step 3
        self.assertEqual(result.json['test'] + 1, toCompare.json['test'])

    def test_get_a_counter(self):
        """Should successfully get a counter"""
        result = self.client.post('/counters/getTest')
        self.assertEqual(result.status_code, status.HTTP_201_CREATED)
        retrieved = self.client.get('/counters/getTest')
        self.assertEqual(retrieved.status_code, status.HTTP_200_OK)
        # Also want to make sure that the value in the counter itself is 0 as it was just made, compare to OG
        self.assertEqual(retrieved.json['getTest'], result.json['getTest'])
        
```

Figure 11: Unit tests for ‘PUT’ and ‘GET’

*These will return 405 status codes as the routes are not implemented yet, thus red phase*

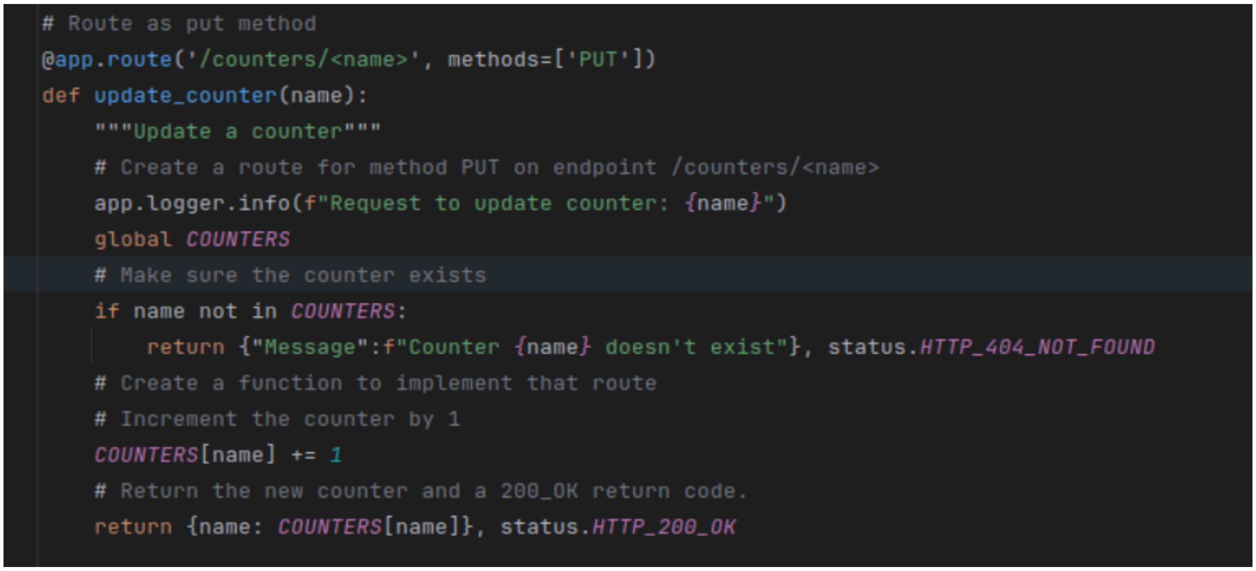


```
birb@fedora:~/UNLV/cs/472/tdd$ bin/nosetests

Counter Tests
- It should create a counter.
- It should return an error for duplicates
- Should successfully get a counter (FAILED)
- Should successfully update a counter (FAILED)

======================================================================
FAIL: Should successfully get a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/birb/UNLV/cs/472/tdd/tests/test_counter.py", line 64, in test_get_a_counter
    self.assertEqual(retrieved.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: getTest
--------------------- >> end captured logging << ---------------------

======================================================================
FAIL: Should successfully update a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/birb/UNLV/cs/472/tdd/tests/test_counter.py", line 55, in test_update_a_counter
    self.assertEqual(toCompare.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: test
--------------------- >> end captured logging << ---------------------

Name             Stmts   Miss  Cover   Missing
----------------------------------------------
src/counter.py      12      0   100%
src/status.py        6      0   100%
----------------------------------------------
TOTAL               18      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.085s

FAILED (failures=2)

birb@fedora:~/UNLV/cs/472/tdd$
```

Figure 12: Testing ‘PUT’ and ‘GET’ without full functionality (exceptions)

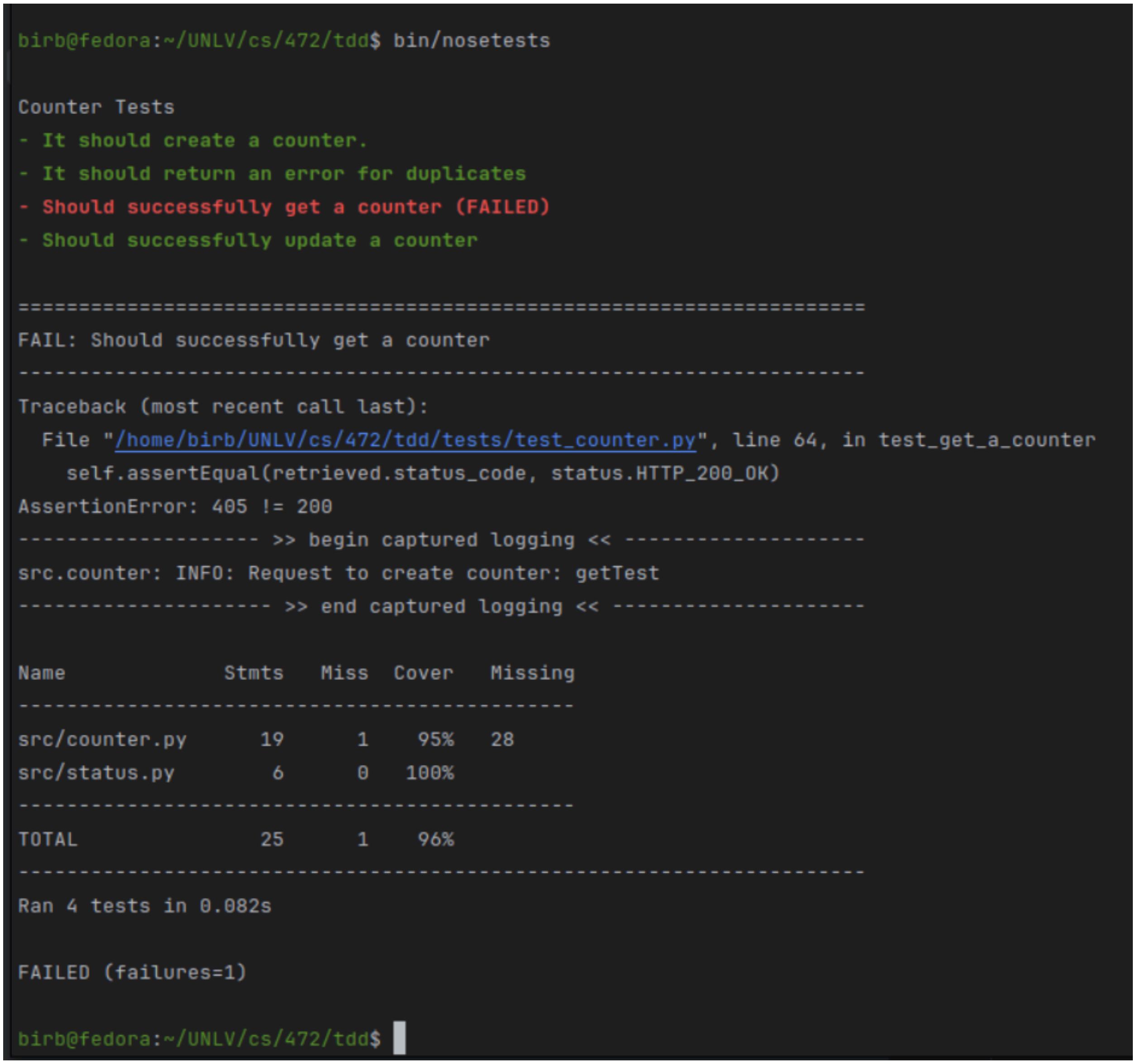
## Green Phase:

Adding functionality for ‘PUT’ and ‘GET’, but without full code coverage in unit tests. As such we implement the REST api calls.



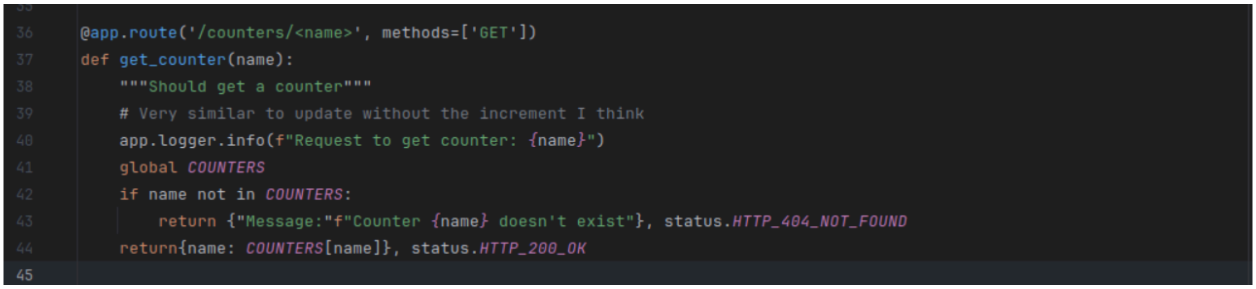
```
# Route as put method
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    # Create a route for method PUT on endpoint /counters/<name>
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    # Make sure the counter exists
    if name not in COUNTERS:
        return {"Message":f"Counter {name} doesn't exist"}, status.HTTP_404_NOT_FOUND
    # Create a function to implement that route
    # Increment the counter by 1
    COUNTERS[name] += 1
    # Return the new counter and a 200_OK return code.
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

Figure 13: Code snippet for ‘PUT’ and updating a counter



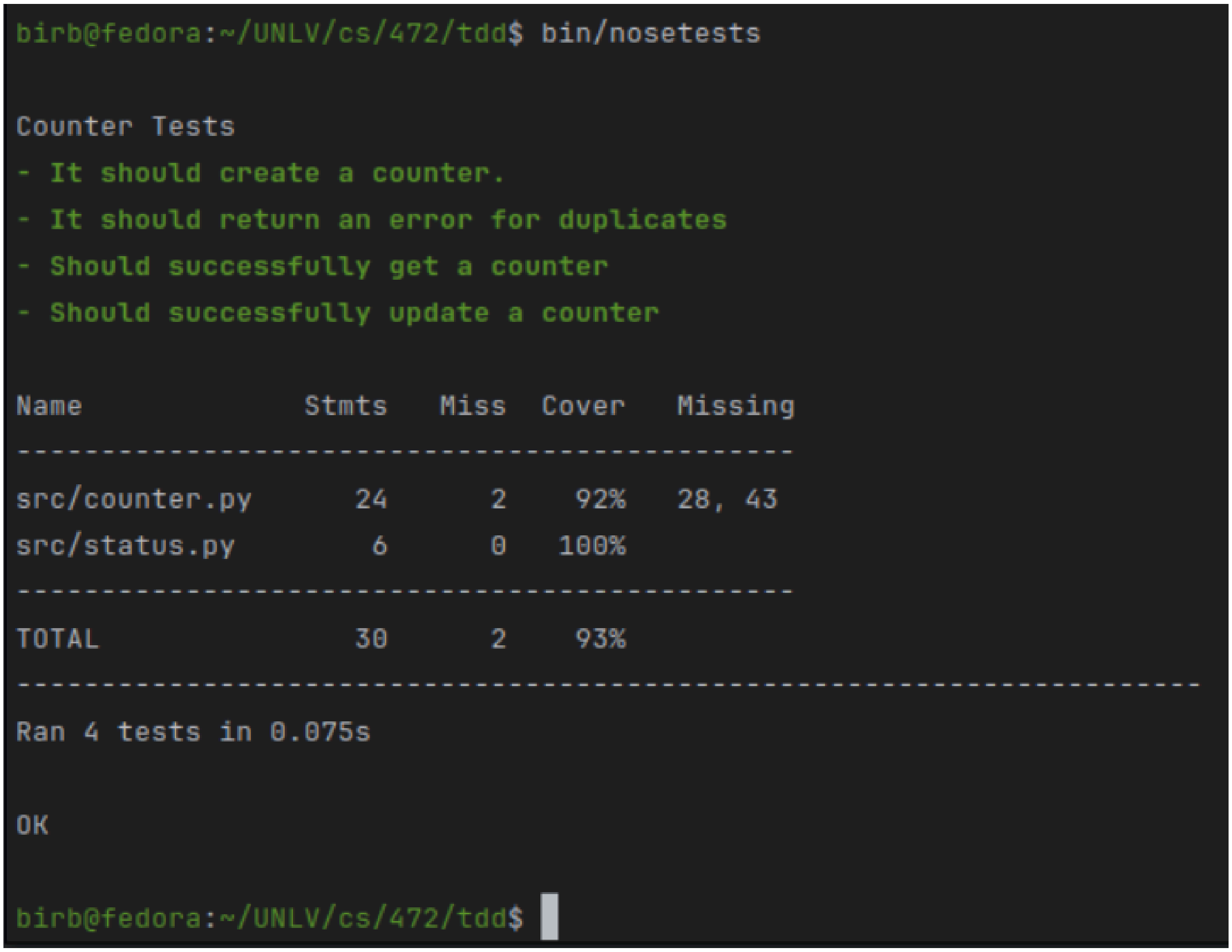
```
birb@fedora:~/UNLV/cs/472/tdd$ bin/nosetests

Counter Tests
- It should create a counter.
- It should return an error for duplicates
- Should successfully get a counter (FAILED)
- Should successfully update a counter

======================================================================
FAIL: Should successfully get a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/birb/UNLV/cs/472/tdd/tests/test_counter.py", line 64, in test_get_a_counter
    self.assertEqual(retrieved.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: getTest
--------------------- >> end captured logging << ---------------------

Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
src/counter.py       19      1    95%   28
src/status.py         6      0   100%
-----------------------------------------------
TOTAL                25      1    96%
----------------------------------------------------------------------
Ran 4 tests in 0.082s

FAILED (failures=1)

birb@fedora:~/UNLV/cs/472/tdd$
```

Figure 14: Green phase for updating a counter



```python
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Should get a counter"""
    # Very similar to update without the increment I think
    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_404_NOT_FOUND
    return{name: COUNTERS[name]}, status.HTTP_200_OK

```

Figure 15: Code snippet for ‘GET’ and getting a counter

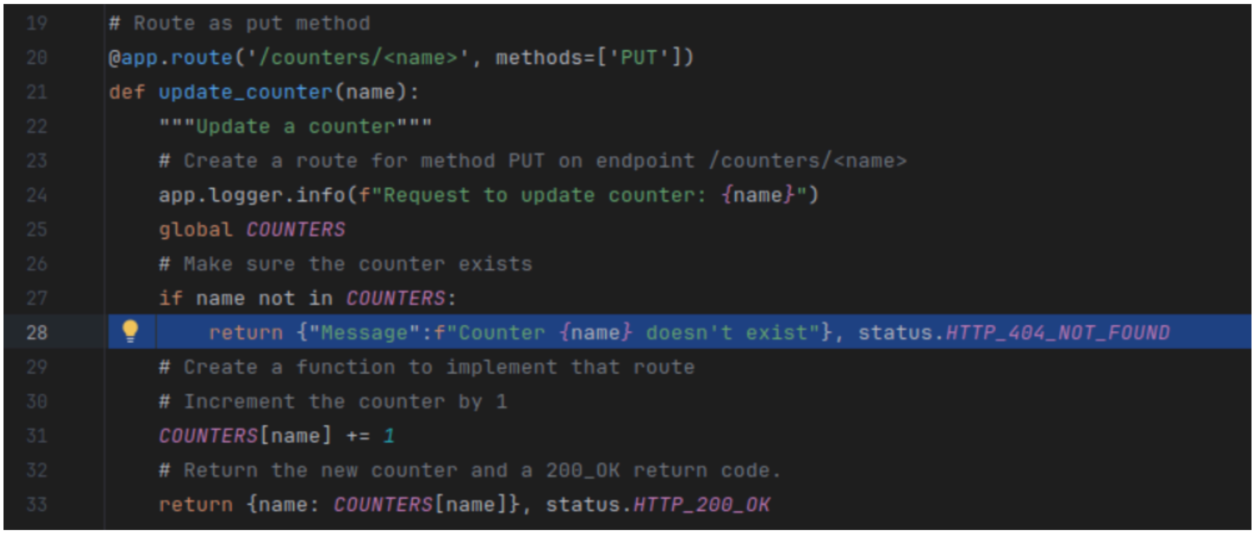


```
birb@fedora:~/UNLV/cs/472/tdd$ bin/nosetests

Counter Tests
- It should create a counter.
- It should return an error for duplicates
- Should successfully get a counter
- Should successfully update a counter

Name                Stmts   Miss  Cover   Missing
----------------------------------------------
src/counter.py         24      2    92%   28, 43
src/status.py           6      0   100%
----------------------------------------------
TOTAL                  30      2    93%
----------------------------------------------------------------------
Ran 4 tests in 0.075s

OK

birb@fedora:~/UNLV/cs/472/tdd$ 
```

Figure 16: Green phase for getting a counter

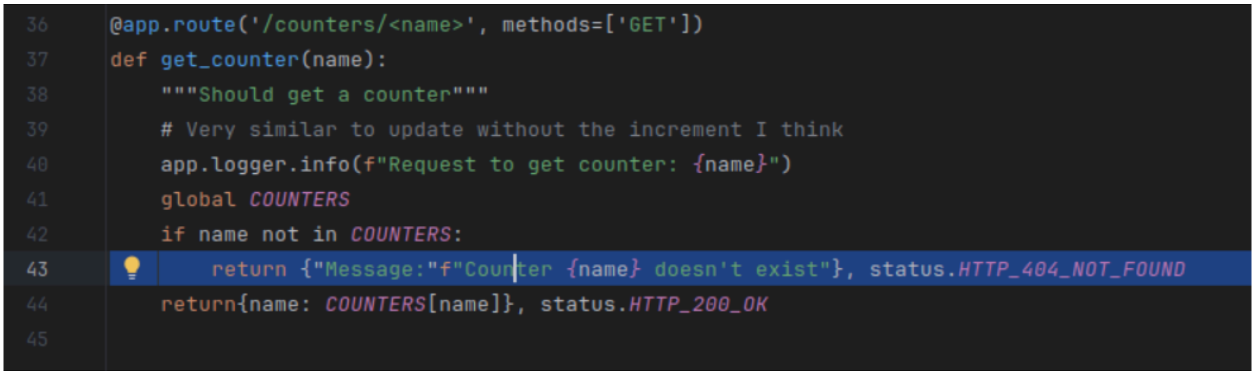
## Refactoring:

We can see in the figure directly above that, even though our implementations of ‘PUT’ and ‘GET’ work as intended and pass our unit tests, we still don’t have full code coverage. As a matter of fact, we managed to miss coverage on the bad cases for each of the two functions, in particular when the counter we attempt to retrieve or update doesn’t exist. As such, we need to implement two more unit tests, each one catered to tackling the aforementioned case for each function. Such tests should simply attempt to get or update a counter that doesn't exist, and then make sure that the status code returned is that of HTTP_404_NOT_FOUND.



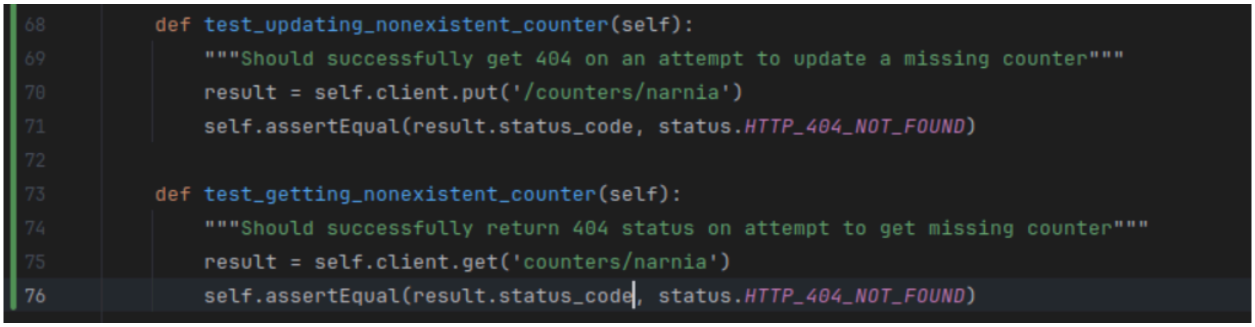
```python
# Route as put method
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    # Create a route for method PUT on endpoint /counters/<name>
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    # Make sure the counter exists
    if name not in COUNTERS:
        return {"Message":f"Counter {name} doesn't exist"}, status.HTTP_404_NOT_FOUND
    # Create a function to implement that route
    # Increment the counter by 1
    COUNTERS[name] += 1
    # Return the new counter and a 200_OK return code.
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

Figure 17: Missing line for ‘PUT’



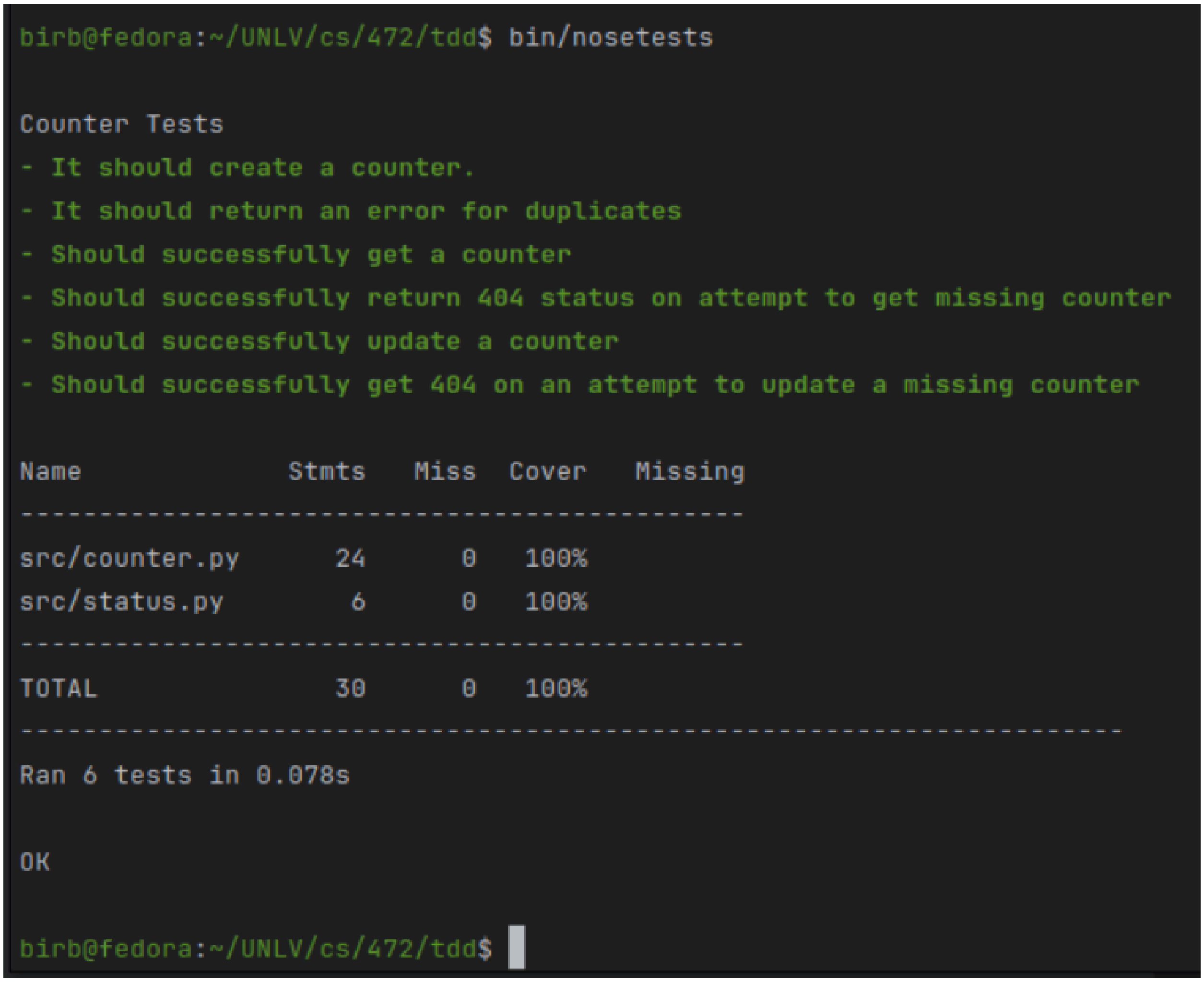
```python
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Should get a counter"""
    # Very similar to update without the increment I think
    app.logger.info(f"Request to get counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message:"f"Counter {name} doesn't exist"}, status.HTTP_404_NOT_FOUND
    return{name: COUNTERS[name]}, status.HTTP_200_OK
```

Figure 18: Missing line for ‘GET’ coverage.



```python
    def test_updating_nonexistent_counter(self):
        """Should successfully get 404 on an attempt to update a missing counter"""
        result = self.client.put('/counters/narnia')
        self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

    def test_getting_nonexistent_counter(self):
        """Should successfully return 404 status on attempt to get missing counter"""
        result = self.client.get('counters/narnia')
        self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

Figure 19: Added unit tests to cover error cases



```
birb@fedora:~/UNLV/cs/472/tdd$ bin/nosetests

Counter Tests
- It should create a counter.
- It should return an error for duplicates
- Should successfully get a counter
- Should successfully return 404 status on attempt to get missing counter
- Should successfully update a counter
- Should successfully get 404 on an attempt to update a missing counter

Name                 Stmts   Miss  Cover   Missing
----------------------------------------------
src/counter.py          24      0   100%
src/status.py            6      0   100%
----------------------------------------------
TOTAL                   30      0   100%
----------------------------------------------------------------------
Ran 6 tests in 0.078s

OK

birb@fedora:~/UNLV/cs/472/tdd$ 
```

Figure 20: Full code coverage after refactored unit testing