Stefano Rubini

Dr. Businge

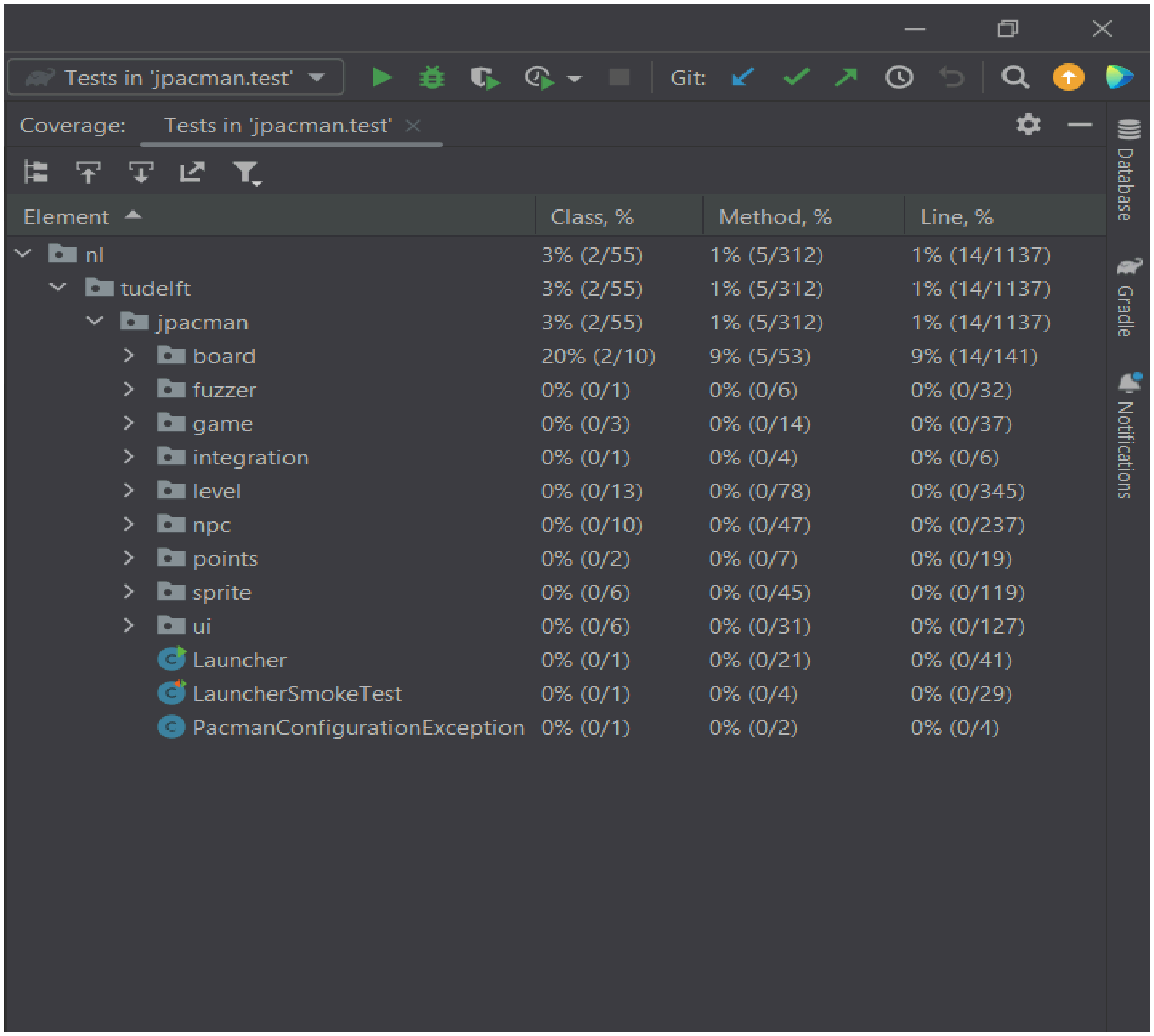
CS 472 1001

February 2, 2024

CS 472 - Software Testing

Link To Fork Repository: https://github.com/StefanoRubini/Barbell

Task 1 - JPacman Test Coverage



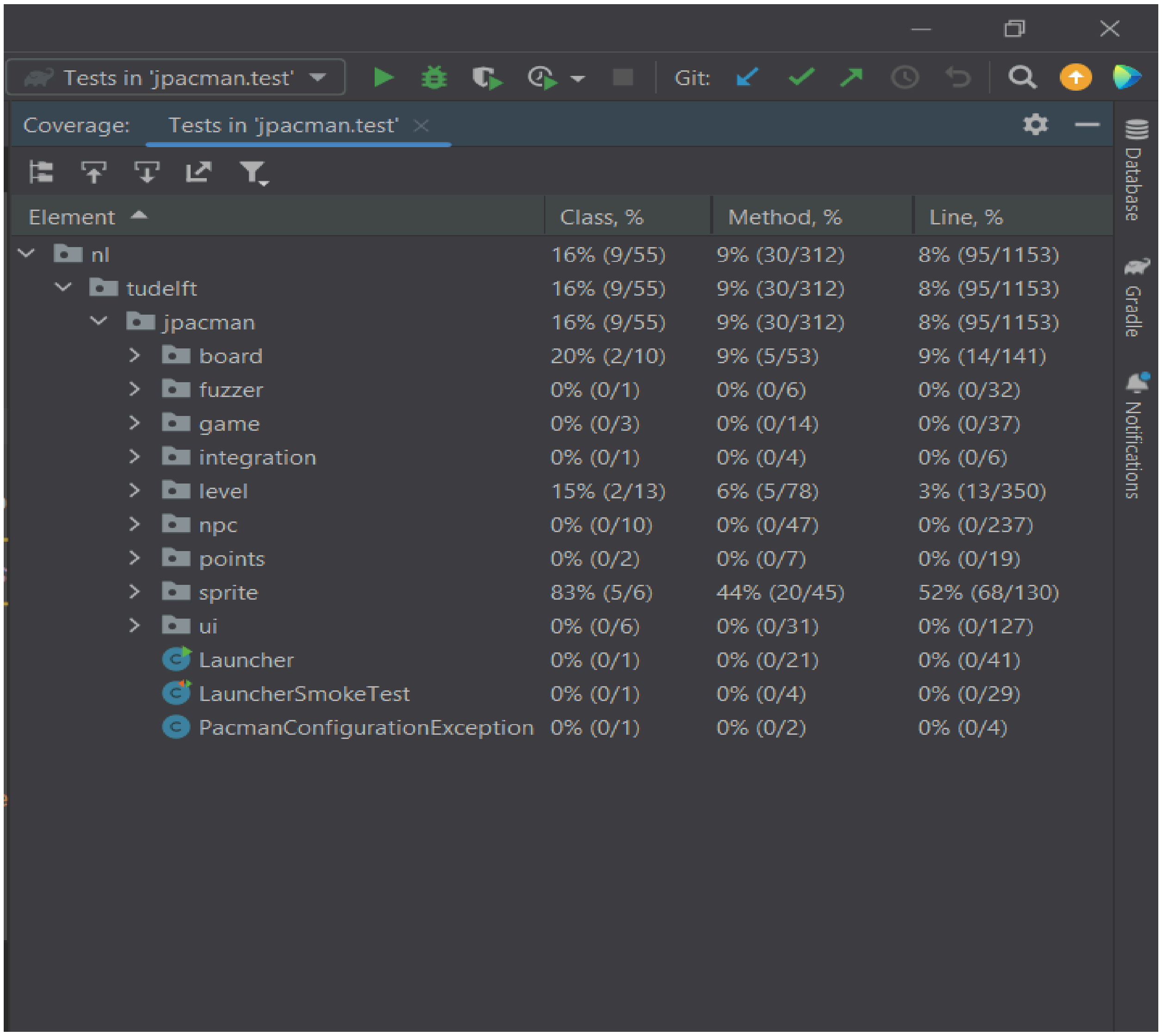
Element	Class, %	Method, %	Line, %
nl	3% (2/55)	1% (5/312)	1% (14/1137)
tudelft	3% (2/55)	1% (5/312)	1% (14/1137)
jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
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)

The coverage results of running "Run 'Tests' in jpacman.test with Coverage".

Question: Is the coverage good enough?

- As we can see from the above coverage results, only 3% of the classes (2/55) have unit tests, and only 1% (5/312) of the methods found within the classes have unit tests. This means that there is immense room for improvement for coverage as the above results demonstrate poor coverage.

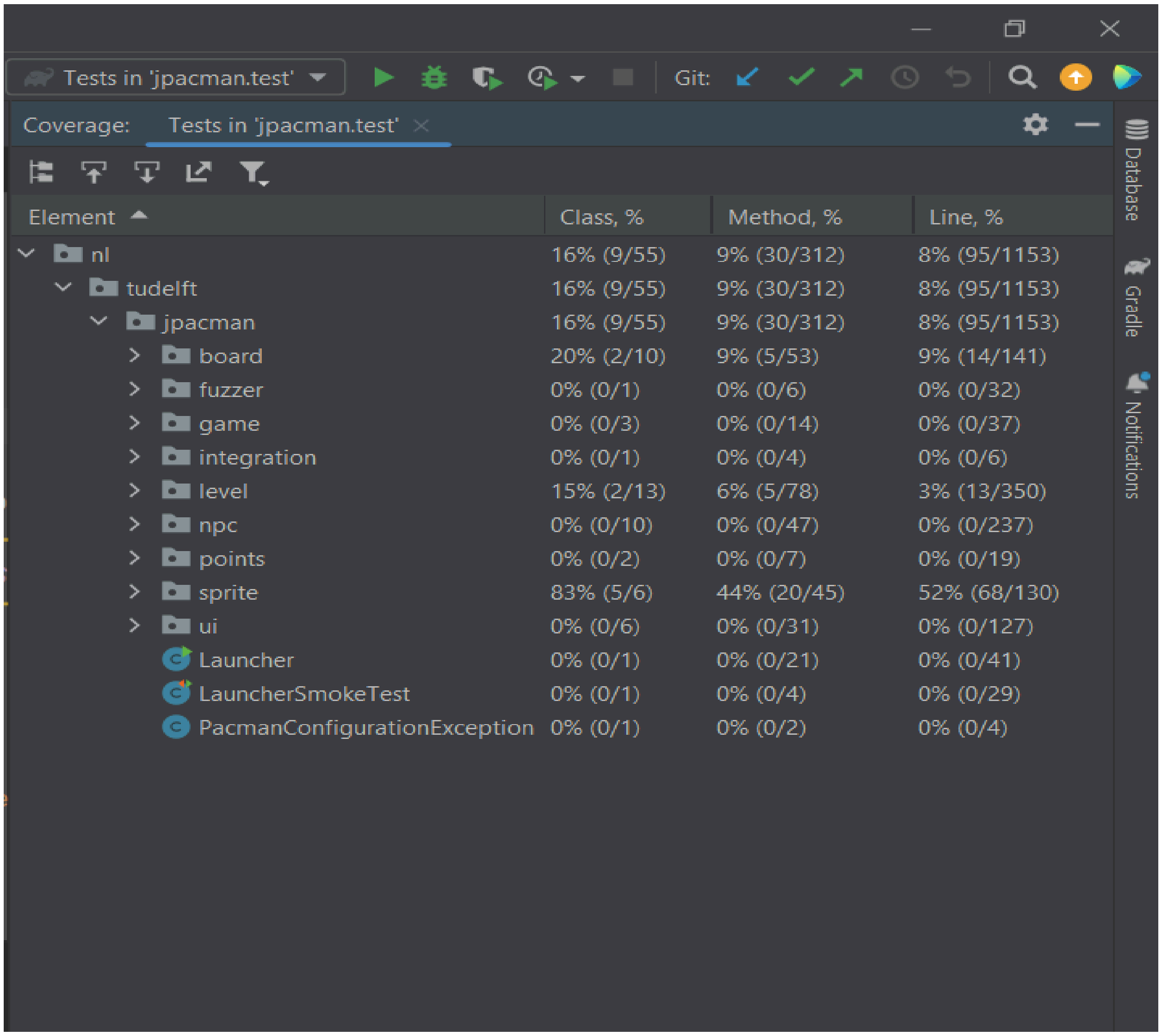
Task 2 - Increasing Coverage on JPacman



Element	Class, %	Method, %	Line, %
nl	16% (9/55)	9% (30/312)	8% (95/1153)
tudelft	16% (9/55)	9% (30/312)	8% (95/1153)
jpacman	16% (9/55)	9% (30/312)	8% (95/1153)
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	83% (5/6)	44% (20/45)	52% (68/130)
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)

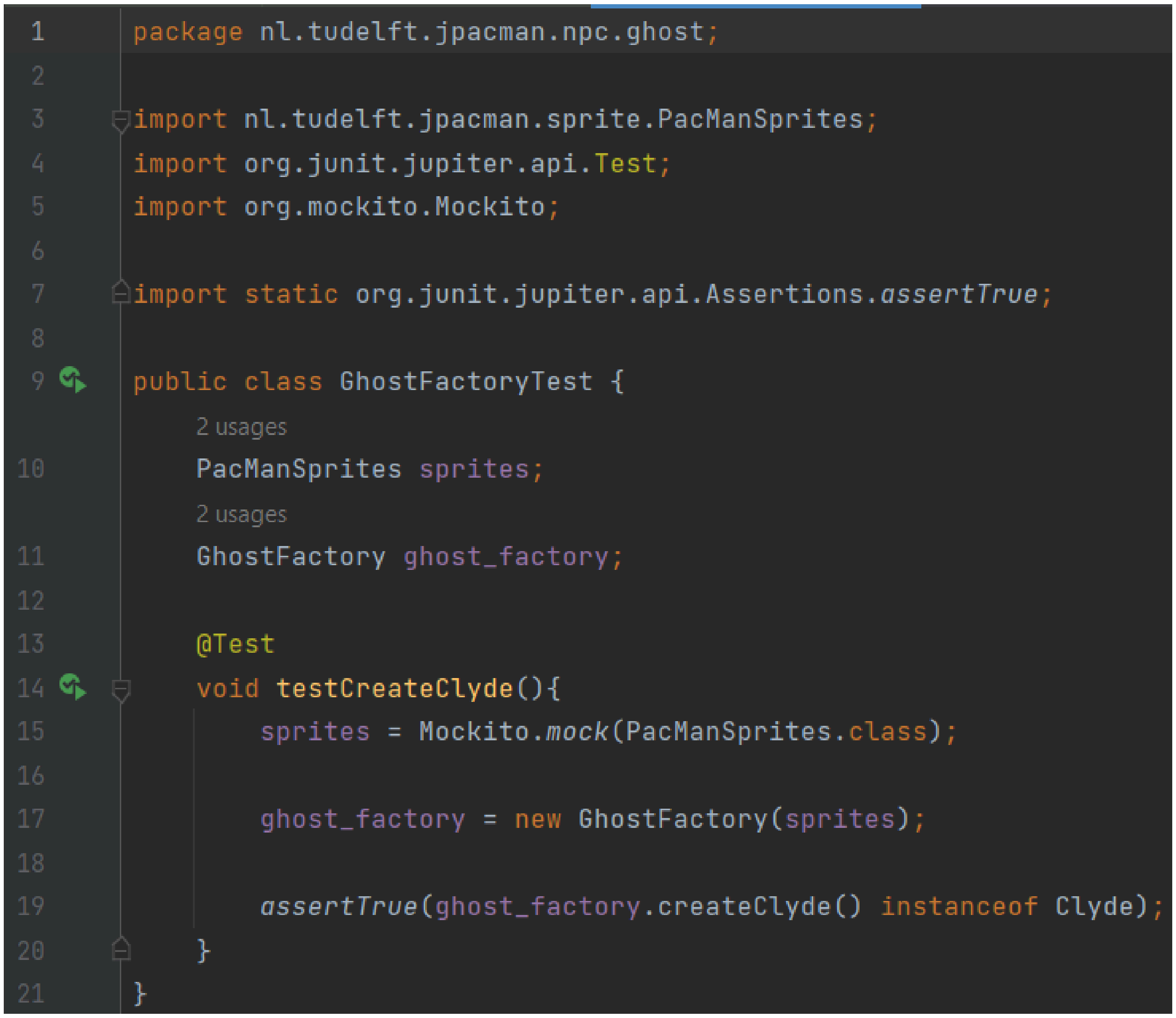
The updated coverage results of running "Run 'Tests' in jpacman.test with Coverage".

Task 2.1 - Identify Three or More Methods in Any Java Classes and Write Unit Tests of Those Methods



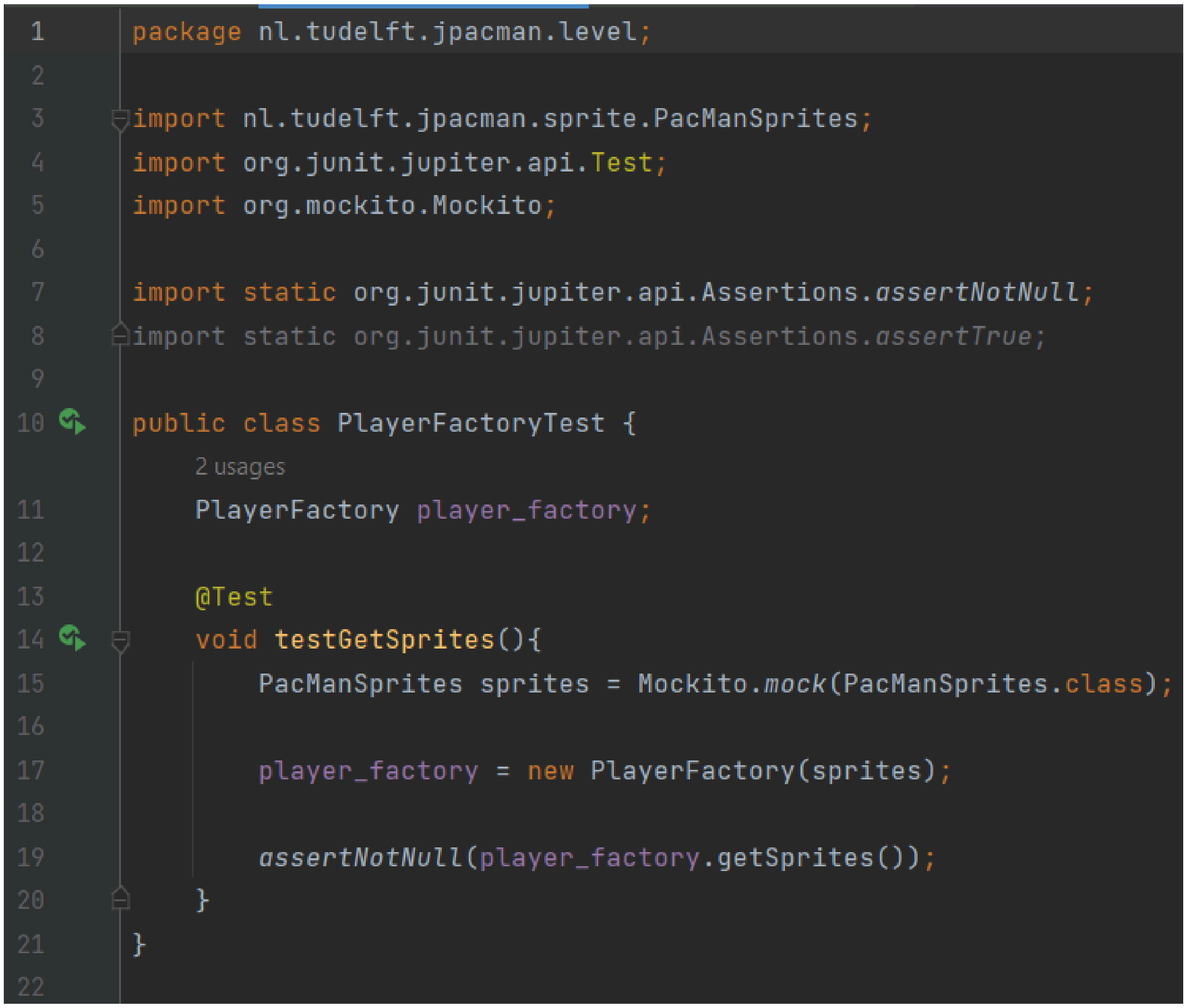
Element	Class, %	Method, %	Line, %
nl	16% (9/55)	9% (30/312)	8% (95/1153)
tudelft	16% (9/55)	9% (30/312)	8% (95/1153)
jpacman	16% (9/55)	9% (30/312)	8% (95/1153)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (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	83% (5/6)	44% (20/45)	52% (68/130)
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)

Coverage results BEFORE identifying three other methods in separate Java classes.



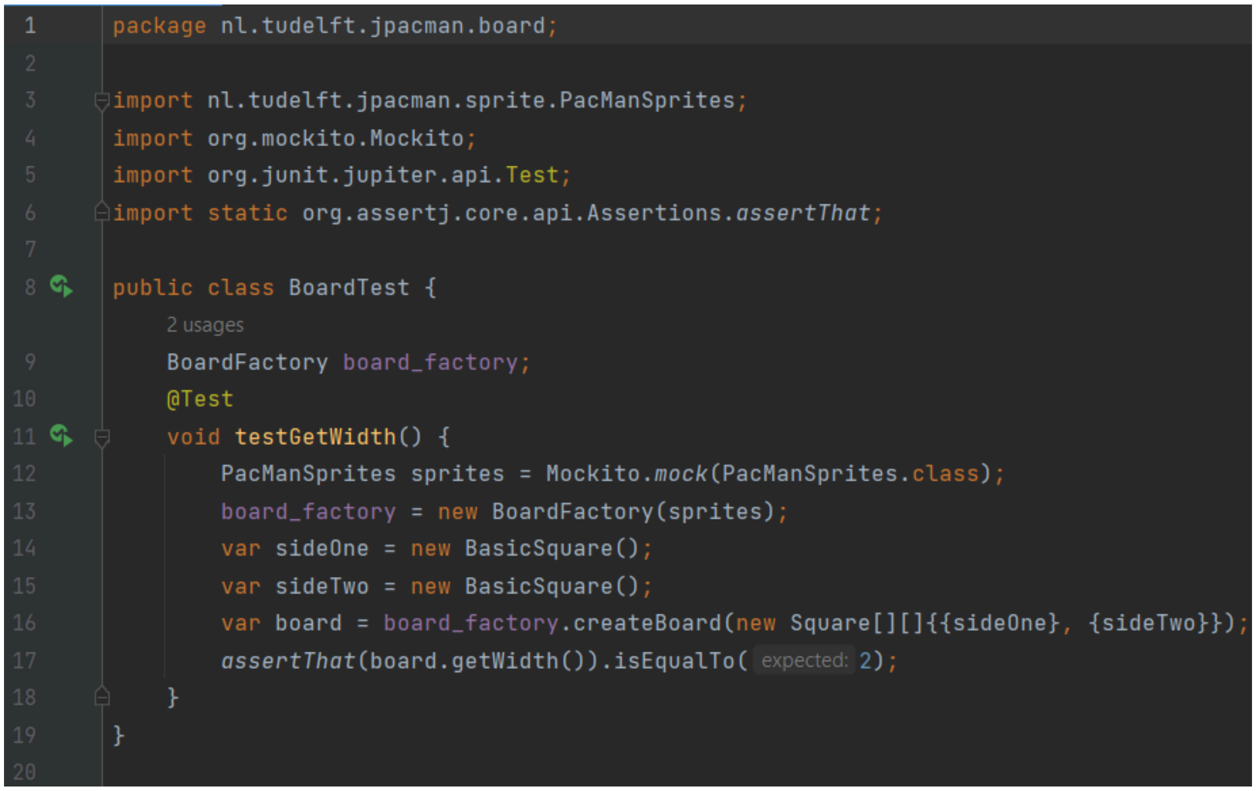
```java
package nl.tudelft.jpacman.npc.ghost;

import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class GhostFactoryTest {
    2 usages
    PacManSprites sprites;
    2 usages
    GhostFactory ghost_factory;

    @Test
    void testCreateClyde(){
        sprites = Mockito.mock(PacManSprites.class);

        ghost_factory = new GhostFactory(sprites);

        assertTrue(ghost_factory.createClyde() instanceof Clyde);
    }
}
```

Unit Test #1 for *src/main/java/nl/tudelft/jpacman/npc/ghost/GhostFactory.createClyde*



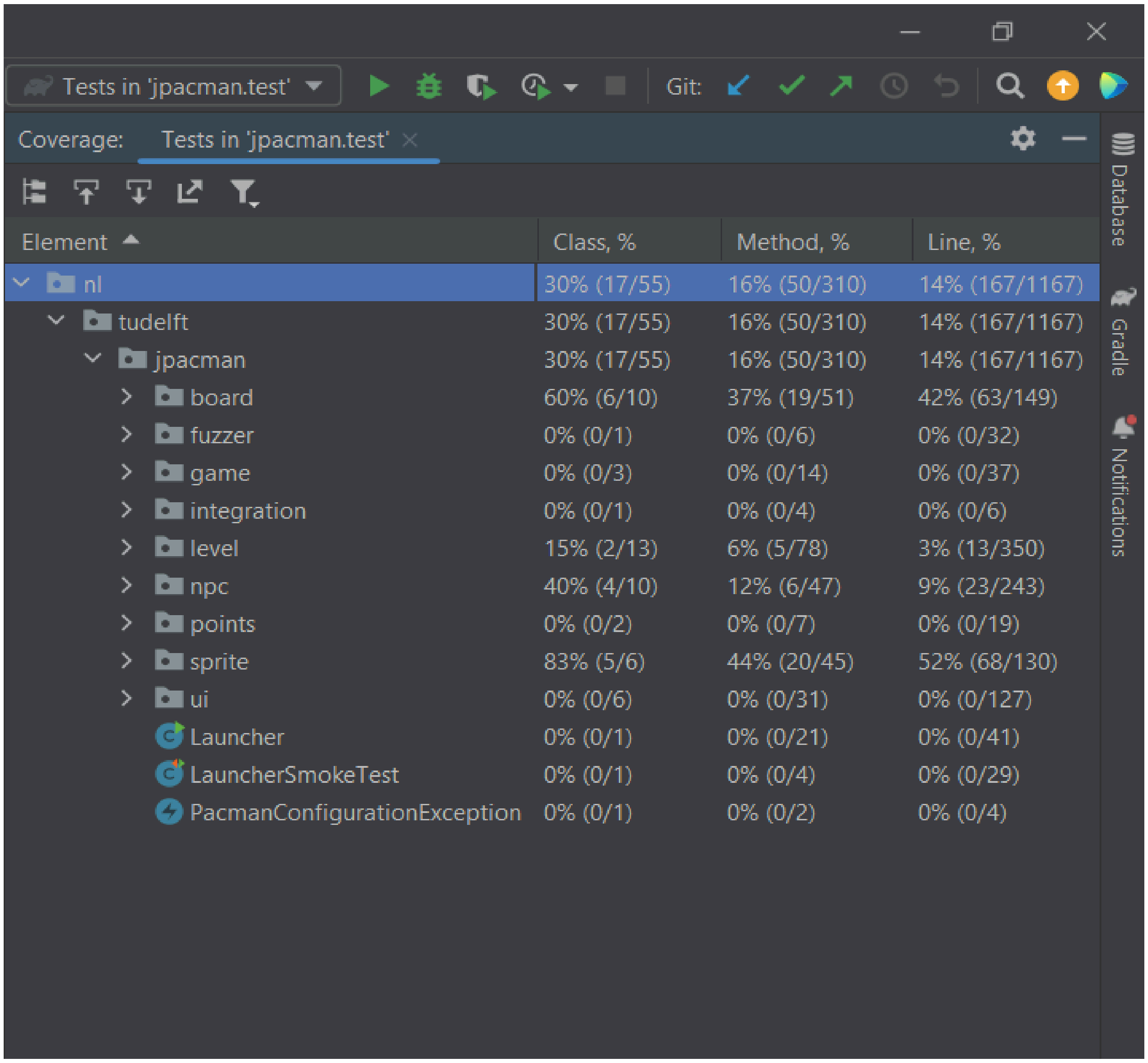
```java
package nl.tudelft.jpacman.level;

import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

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

public class PlayerFactoryTest {
    2 usages
    PlayerFactory player_factory;

    @Test
    void testGetSprites(){
        PacManSprites sprites = Mockito.mock(PacManSprites.class);

        player_factory = new PlayerFactory(sprites);

        assertNotNull(player_factory.getSprites());
    }
}
```

Unit Test #2 for *src/main/java/nl/tudelft/jpacman/level/PlayerFactory.getSprites*



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

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

public class BoardTest {
    2 usages
    BoardFactory board_factory;
    @Test
    void testGetWidth() {
        PacManSprites sprites = Mockito.mock(PacManSprites.class);
        board_factory = new BoardFactory(sprites);
        var sideOne = new BasicSquare();
        var sideTwo = new BasicSquare();
        var board = board_factory.createBoard(new Square[][]{{sideOne}, {sideTwo}});
        assertThat(board.getWidth()).isEqualTo( expected: 2);
    }
}
```

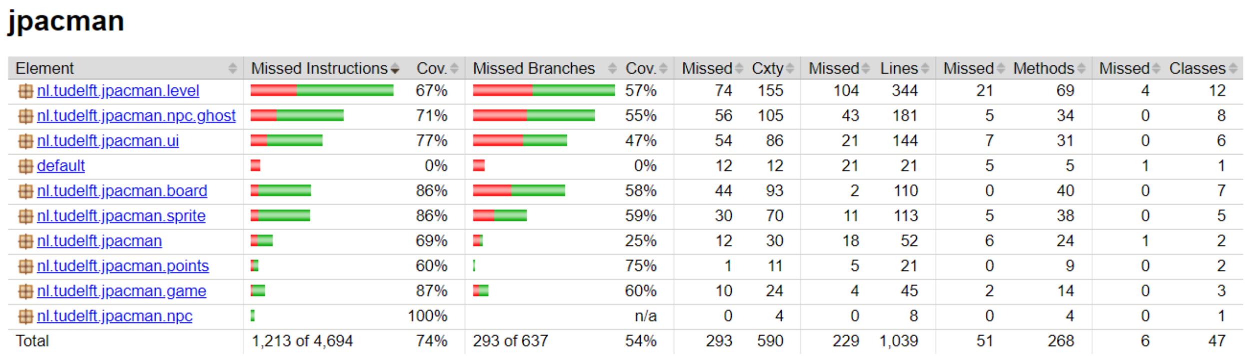
Unit Test #3 for *src/main/java/nl/tudelft/jpacman/board/Board.getWidth*



Element	Class, %	Method, %	Line, %
nl	30% (17/55)	16% (50/310)	14% (167/1167)
tudelft	30% (17/55)	16% (50/310)	14% (167/1167)
jpacman	30% (17/55)	16% (50/310)	14% (167/1167)
board	60% (6/10)	37% (19/51)	42% (63/149)
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	40% (4/10)	12% (6/47)	9% (23/243)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	44% (20/45)	52% (68/130)
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)

Coverage results AFTER identifying three other methods in separate Java classes.

Task 3 - JaCoCo Report on JPacman



jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	74	155	104	344	21	69	4	12
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,213 of 4,694	74%	293 of 637	54%	293	590	229	1,039	51	268	6	47

Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?

- The results that are obtained from both JaCoCo and IntelliJ are different from one another, due to the simple fact that they both focus on different aspects of the project. As we can see above, JaCoCo focuses on missed instructions and missed branches, meanwhile, IntelliJ gives a breakdown of which classes, which methods, and which lines are covered, and it gives a percentage for each of the above categories on how many of the classes, methods and lines are covered.

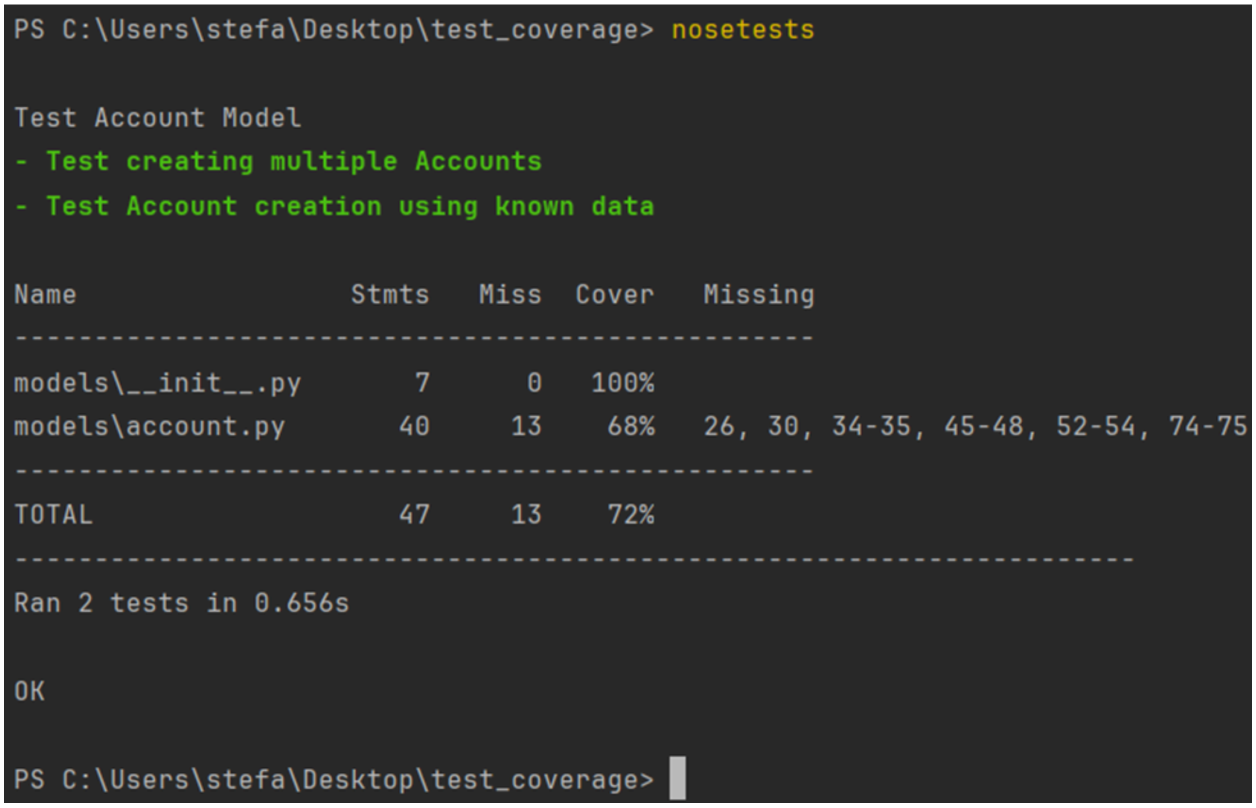
Did you find helpful the source code visualization from JaCoCo on uncovered branches?

- I found the source code visualization from JaCoCo helpful in regards to uncovered branches because it gives us a better understanding of potential areas in our project that may need more attention when debugging any issues that arise and it tells us exactly which branches are uncovered and require our attention.

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

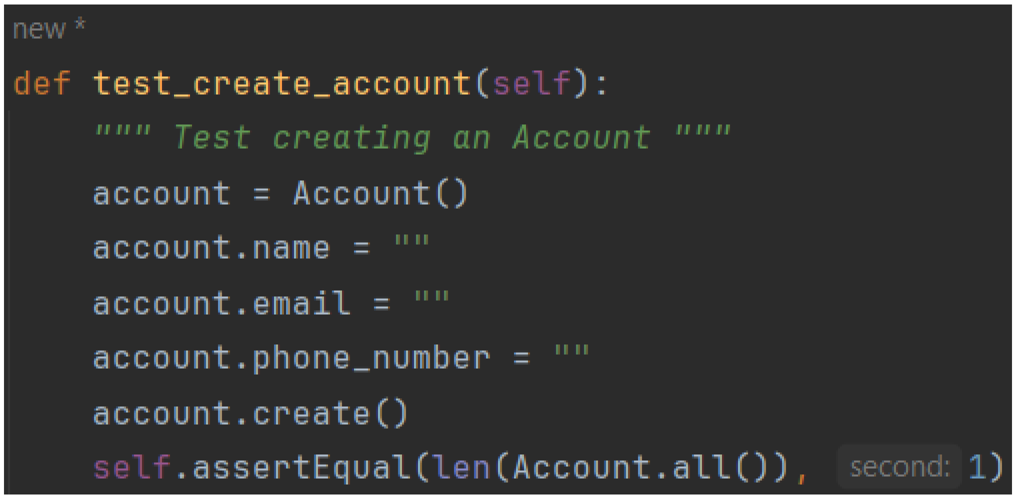
- In terms of project complexity, I would prefer to use JaCoCo's reporting because it would help significantly in bringing attention to any missed instructions and branches where bugs can exist, which helps us to reduce the time needed for debugging and speed up project development. However, when it comes to projects that are not as complex and are not as large, I would prefer to use IntelliJ's reporting because it opens directly in the IDE and provides real-time feedback on coverage every time a program is compiled and executed without the need of opening an external application, unlike JaCoCo, which requires a web browser to display its HTML contents.

Task 4 - Working with Python Test Coverage

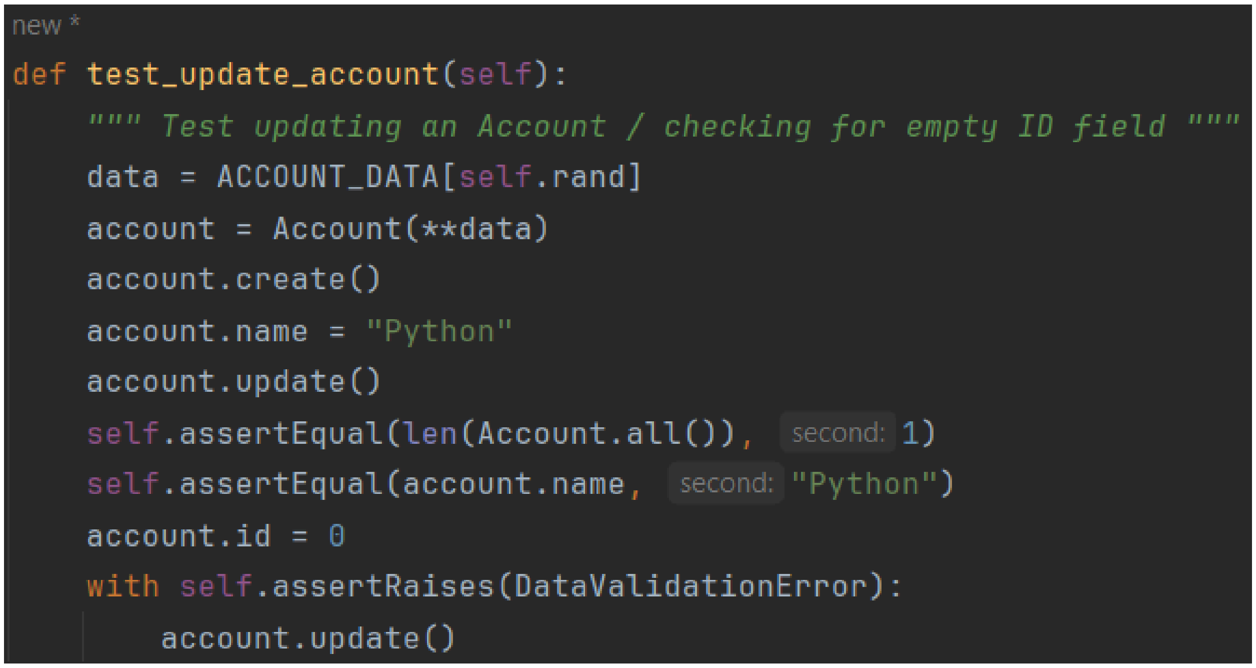


```
PS C:\Users\stefa\Desktop\test_coverage> nosetests

Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data

Name                     Stmts   Miss  Cover   Missing
--------------------------------------------------
models\__init__.py           7      0   100%
models\account.py           40     13    68%   26, 30, 34-35, 45-48, 52-54, 74-75
--------------------------------------------------
TOTAL                       47     13    72%
----------------------------------------------------------------------
Ran 2 tests in 0.656s

OK

PS C:\Users\stefa\Desktop\test_coverage>
```

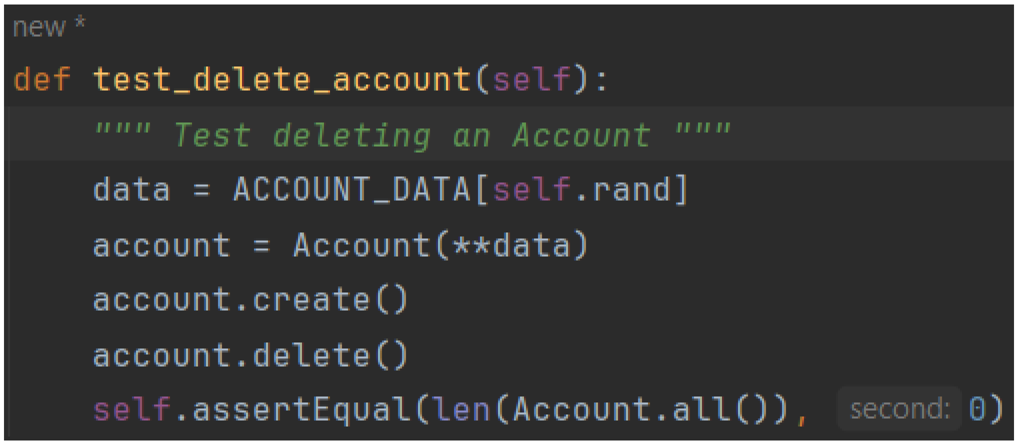
Coverage results of running "nosetests" BEFORE writing unit tests


new *
```
def test_create_account(self):
    """ Test creating an Account """
    account = Account()
    account.name = ""
    account.email = ""
    account.phone_number = ""
    account.create()
    self.assertEqual(len(Account.all()), second: 1)
```

Unit Test #1 for creating an account

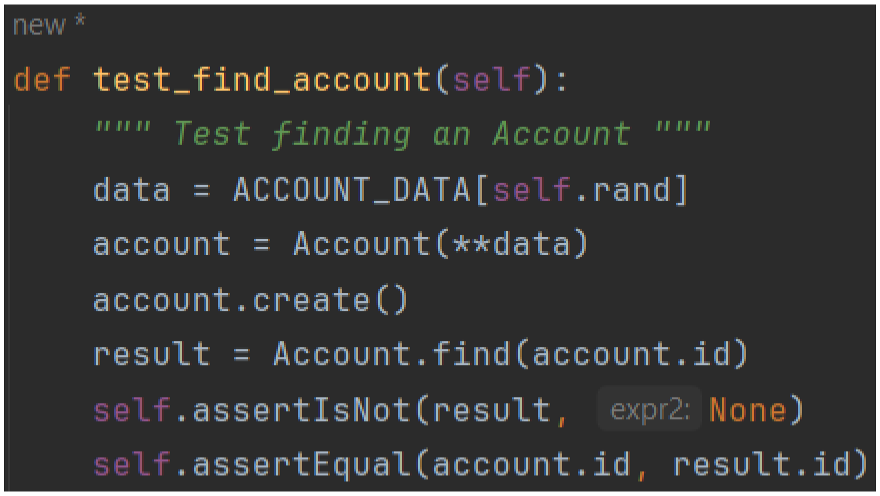

new *
```
def test_update_account(self):
    """ Test updating an Account / checking for empty ID field """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.name = "Python"
    account.update()
    self.assertEqual(len(Account.all()), second: 1)
    self.assertEqual(account.name, second: "Python")
    account.id = 0
    with self.assertRaises(DataValidationError):
        account.update()
```

Unit Test #2 for updating an account and checking if an account has an empty ID field



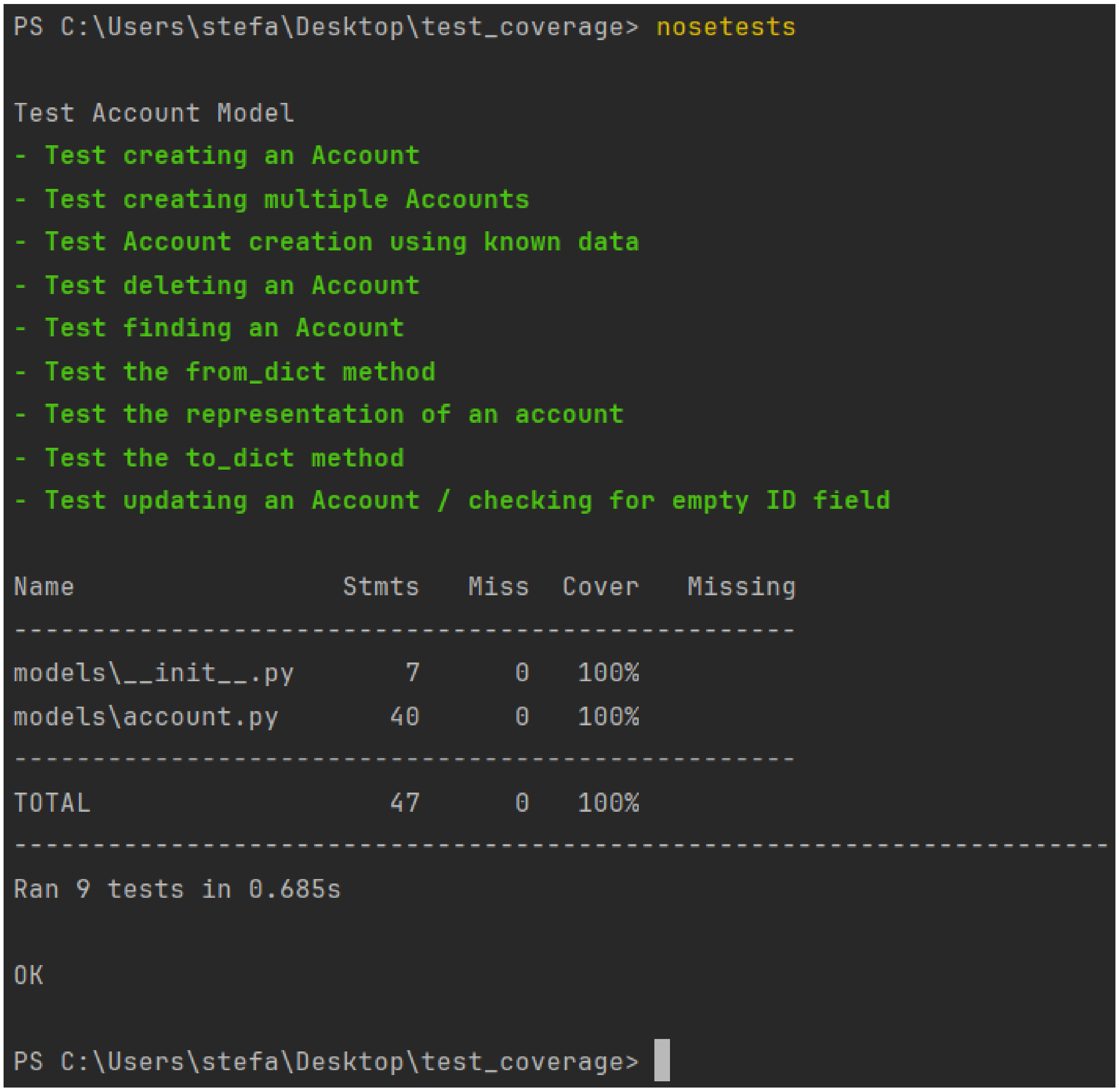
```
new *
def test_delete_account(self):
    """ Test deleting an Account """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.delete()
    self.assertEqual(len(Account.all()),  second: 0)
```

Unit Test #3 for deleting an account



```
new *
def test_find_account(self):
    """ Test finding an Account """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    result = Account.find(account.id)
    self.assertIsNot(result,  expr2: None)
    self.assertEqual(account.id, result.id)
```

Unit Test #4 for finding an account



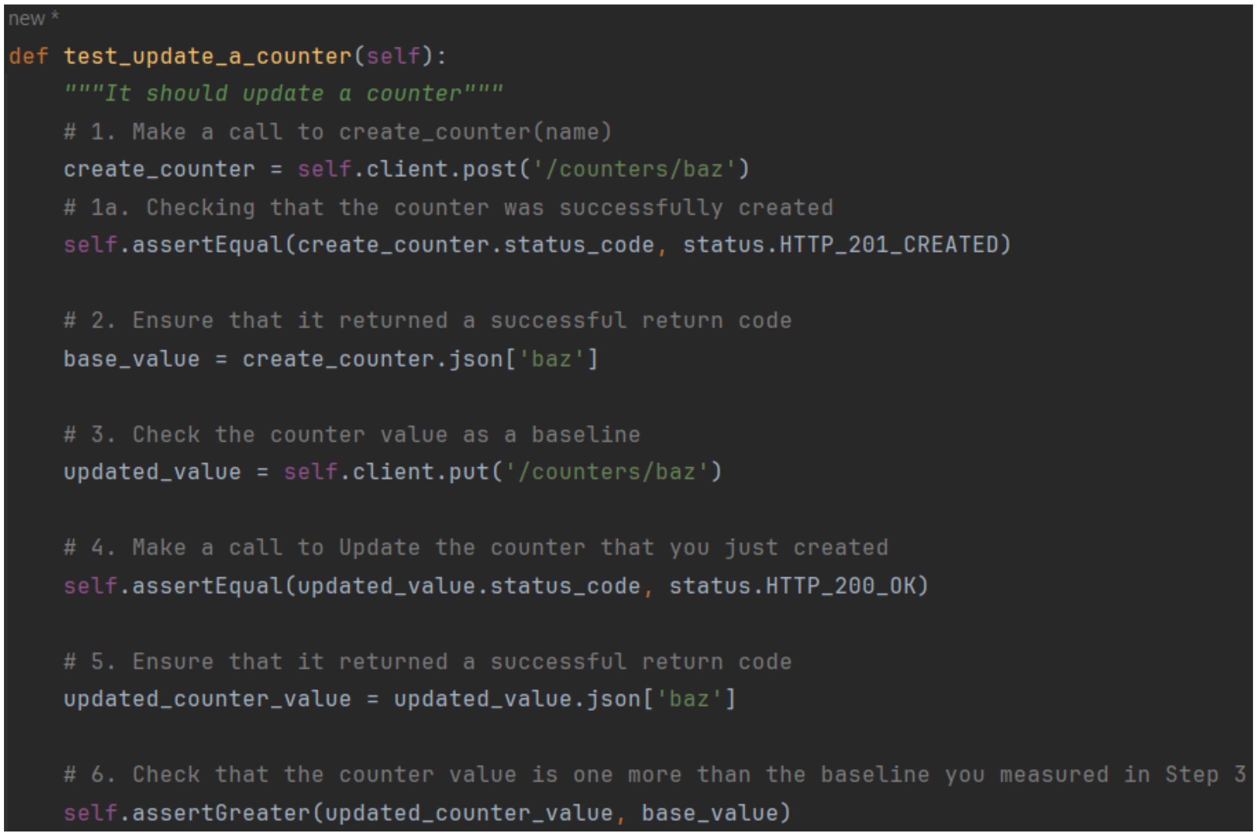
```
PS C:\Users\stefa\Desktop\test_coverage> nosetests

Test Account Model
- Test creating an Account
- Test creating multiple Accounts
- Test Account creation using known data
- Test deleting an Account
- Test finding an Account
- Test the from_dict method
- Test the representation of an account
- Test the to_dict method
- Test updating an Account / checking for empty ID field

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 0.685s

OK

PS C:\Users\stefa\Desktop\test_coverage>
```

Coverage results of running "nosetests" AFTER writing unit tests

Task 5 - TDD

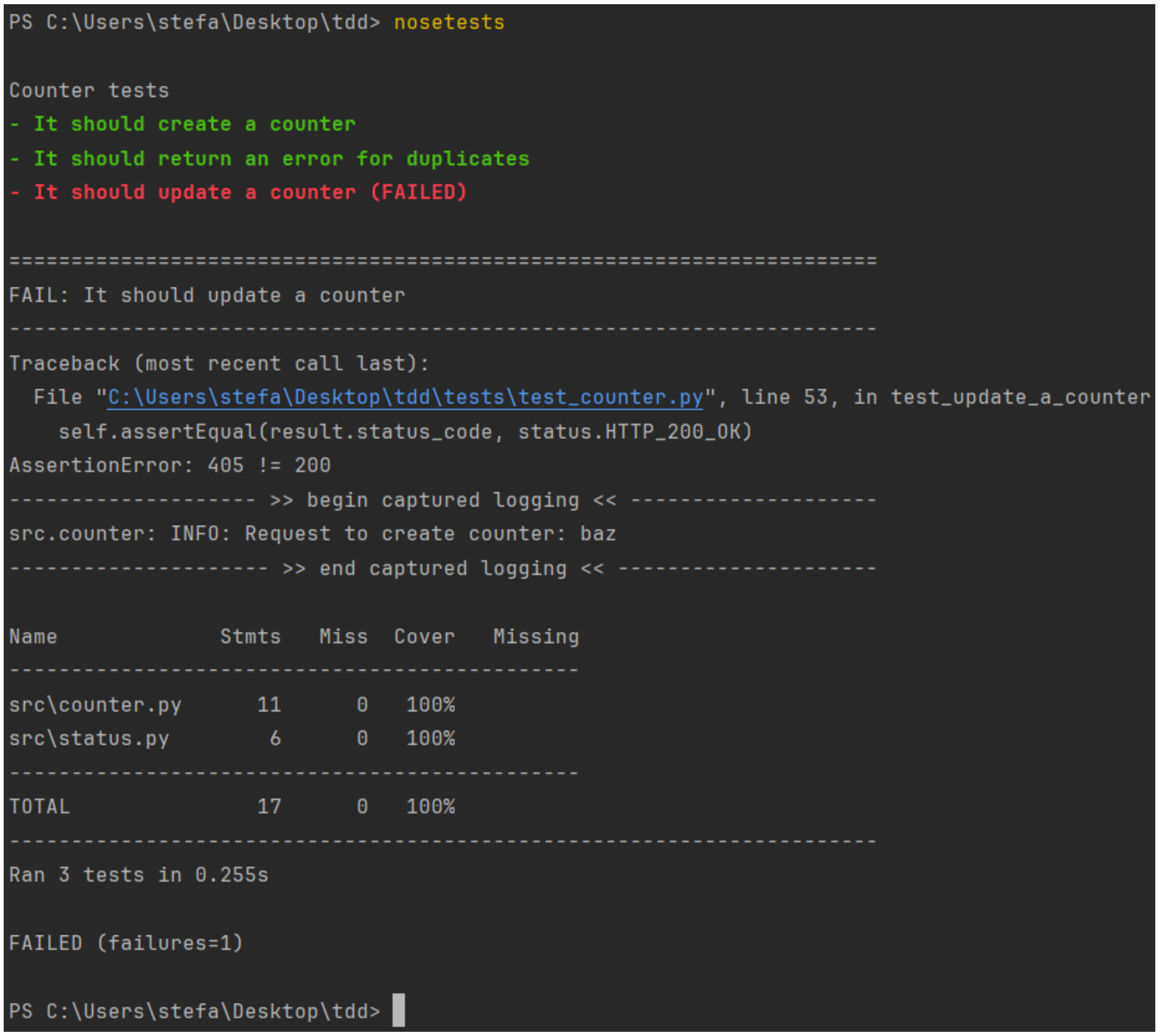
THE RED PHASE OF TEST_UPDATE_A_COUNTER(SELF)

test_update_a_counter(self) code



```
new *
def test_update_a_counter(self):
    """It should update a counter"""
    # 1. Make a call to create_counter(name)
    create_counter = self.client.post('/counters/baz')
    # 1a. Checking that the counter was successfully created
    self.assertEqual(create_counter.status_code, status.HTTP_201_CREATED)

    # 2. Ensure that it returned a successful return code
    base_value = create_counter.json['baz']

    # 3. Check the counter value as a baseline
    updated_value = self.client.put('/counters/baz')

    # 4. Make a call to Update the counter that you just created
    self.assertEqual(updated_value.status_code, status.HTTP_200_OK)

    # 5. Ensure that it returned a successful return code
    updated_counter_value = updated_value.json['baz']

    # 6. Check that the counter value is one more than the baseline you measured in Step 3
    self.assertGreater(updated_counter_value, base_value)
```

From the screenshot below this block of text, we can see that after defining our *test_update_a_counter(self)* test, we are in the RED phase because we are writing a test case for a function that does not yet exist, which makes sense why we are in the red phase because this is the nature of following the coding paradigm of Test Driven Development (TDD), where we first write the test cases for the code we wish we had and then writing the code to make the test cases pass.

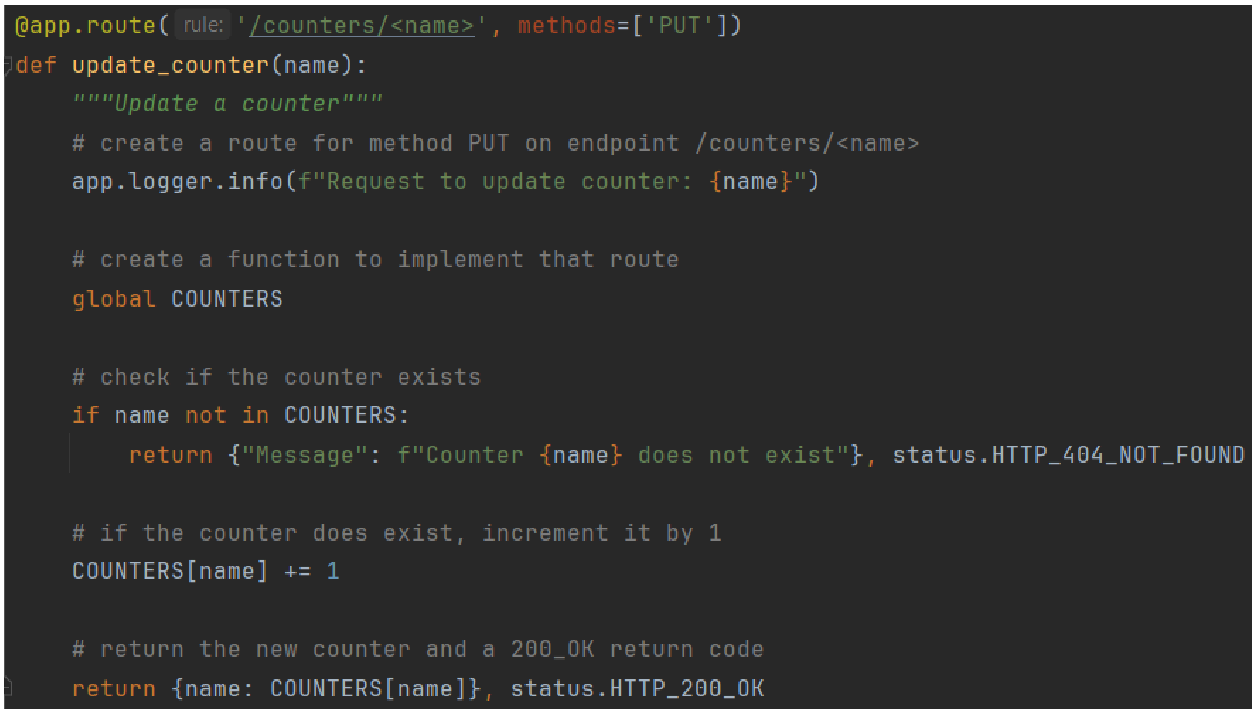


```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should update a counter (FAILED)

======================================================================
FAIL: It should update a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\stefa\Desktop\tdd\tests\test_counter.py", line 53, in test_update_a_counter
    self.assertEqual(result.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: baz
--------------------- >> end captured logging << ---------------------

Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
src\counter.py       11      0   100%
src\status.py         6      0   100%
-----------------------------------------------
TOTAL                17      0   100%
----------------------------------------------------------------------
Ran 3 tests in 0.255s

FAILED (failures=1)

PS C:\Users\stefa\Desktop\tdd>
```

The result of writing a test case (test_update_a_counter(self)) for a block of code that doesn't exist yet (update_counter(name))

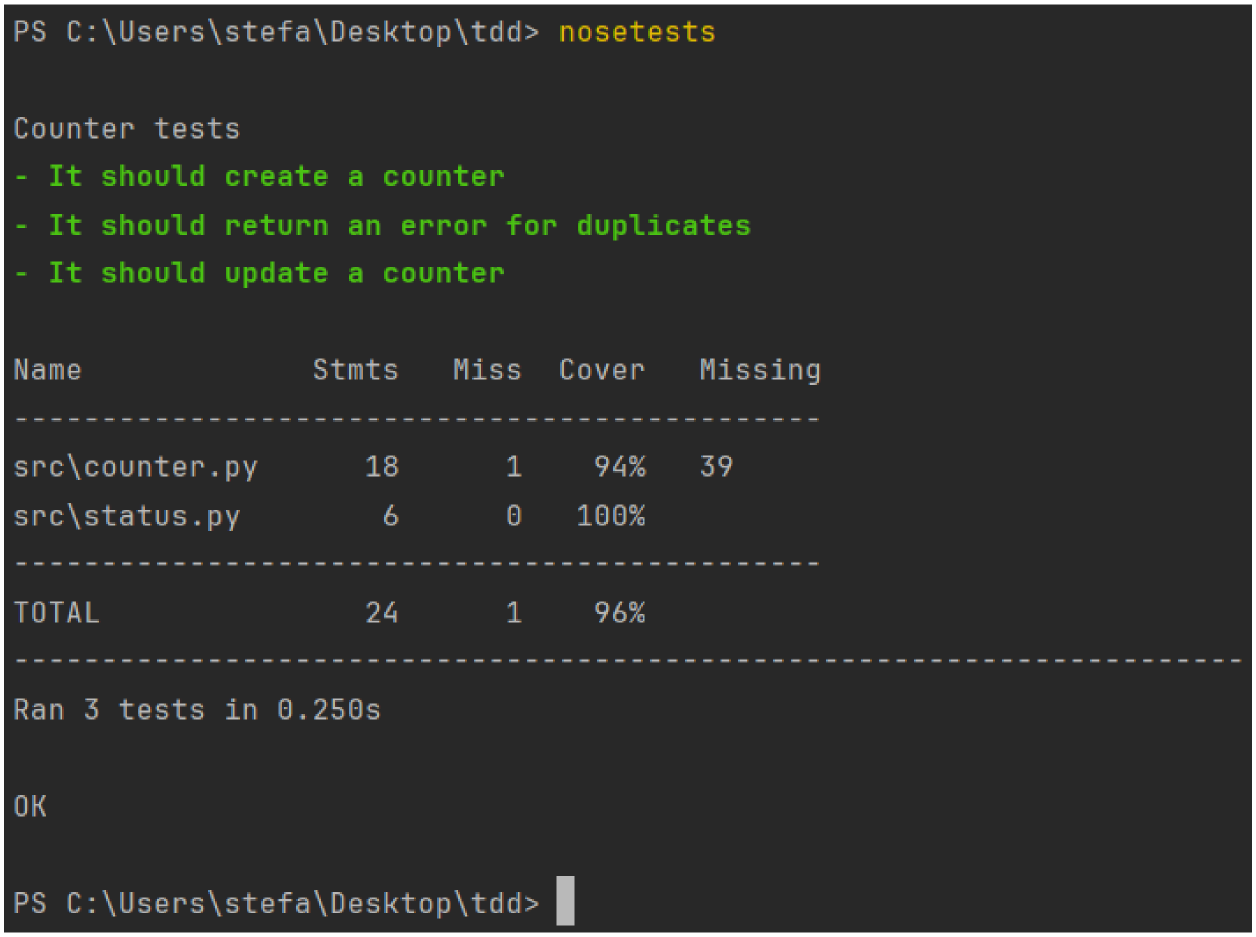
THE GREEN PHASE OF TEST_UPDATE_A_COUNTER(SELF)

update_counter(name) code



```
@app.route( rule: '/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}")

    # create a function to implement that route
    global COUNTERS

    # check if the counter exists
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND

    # if the counter does exist, increment it by 1
    COUNTERS[name] += 1

    # return the new counter and a 200_OK return code
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

From the screenshot below this block of text, we can see that after defining the **update_counter(name)** function for our **test_update_a_counter(self)** test case, we can see that after running “nosetests” we pass all of our current counter tests without any issue.

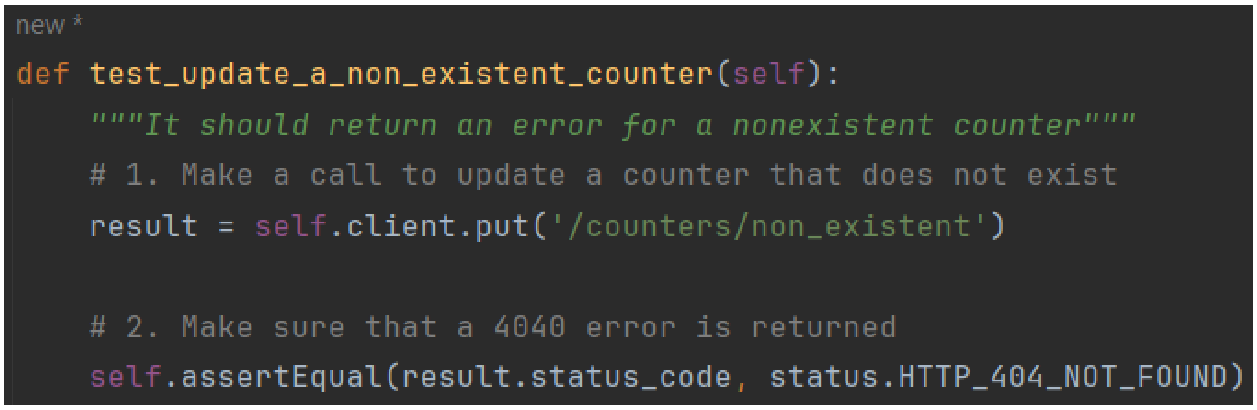


```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should update a counter

Name                Stmts   Miss  Cover   Missing
---------------------------------------------
src\counter.py         18      1    94%   39
src\status.py           6      0   100%
---------------------------------------------
TOTAL                  24      1    96%
----------------------------------------------------------------------
Ran 3 tests in 0.250s

OK

PS C:\Users\stefa\Desktop\tdd>
```

THE REFACTOR PHASE OF TEST_UPDATE_A_COUNTER(SELF)

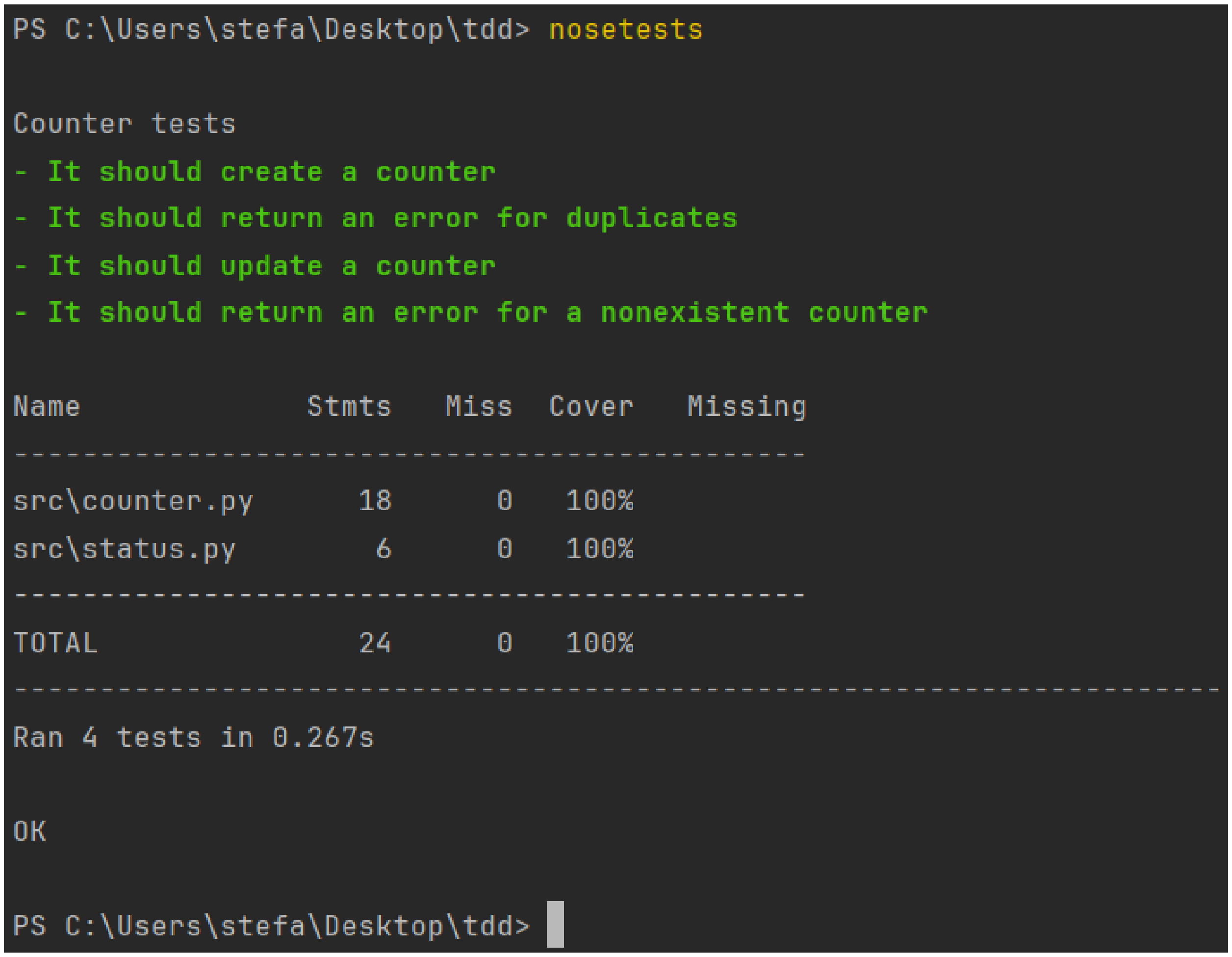
From the aforementioned screenshot above, we can see that we do NOT have 100% coverage for all of the statements in counter.py, as line 39 is the only line that is missing. Therefore, we have to refactor our test code in order for this specific line to be accounted for, and this line happens to be interested in whether or not we test the case in which we try to update a non-existent counter, and we currently do not have a test case for that scenario. Therefore, we go ahead and account for that scenario by adding the following test case test_counter.py:

test_update_a_non_existent_counter(self) code



```
new *
def test_update_a_non_existent_counter(self):
    """It should return an error for a nonexistent counter"""
    # 1. Make a call to update a counter that does not exist
    result = self.client.put('/counters/non_existent')

    # 2. Make sure that a 4040 error is returned
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

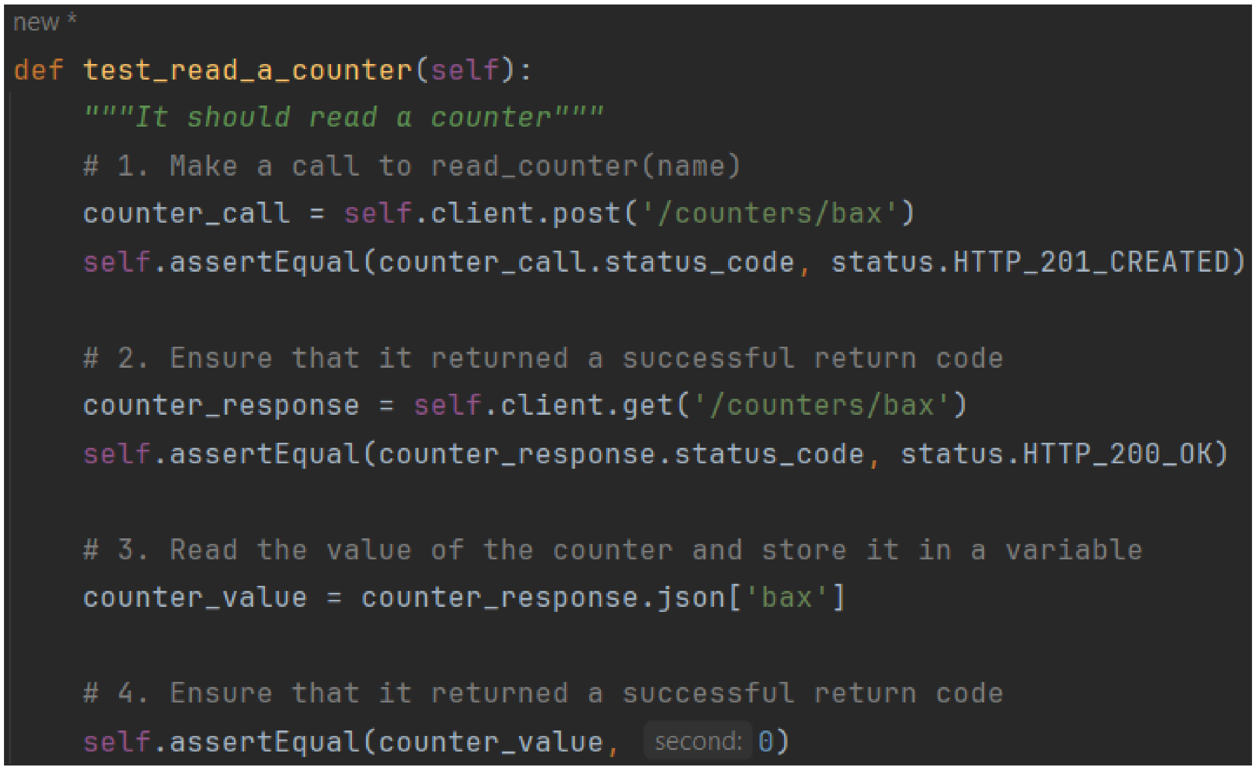
After adding the above test case in test_counter.py, we go ahead and run “nosetests” and we can now see that our code coverage is now 100%, as can be seen in the screenshot attached below:



```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should update a counter
- It should return an error for a nonexistent counter

Name                Stmts   Miss  Cover   Missing
----------------------------------------------
src\counter.py         18      0   100%
src\status.py           6      0   100%
----------------------------------------------
TOTAL                  24      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.267s

OK

PS C:\Users\stefa\Desktop\tdd>
```

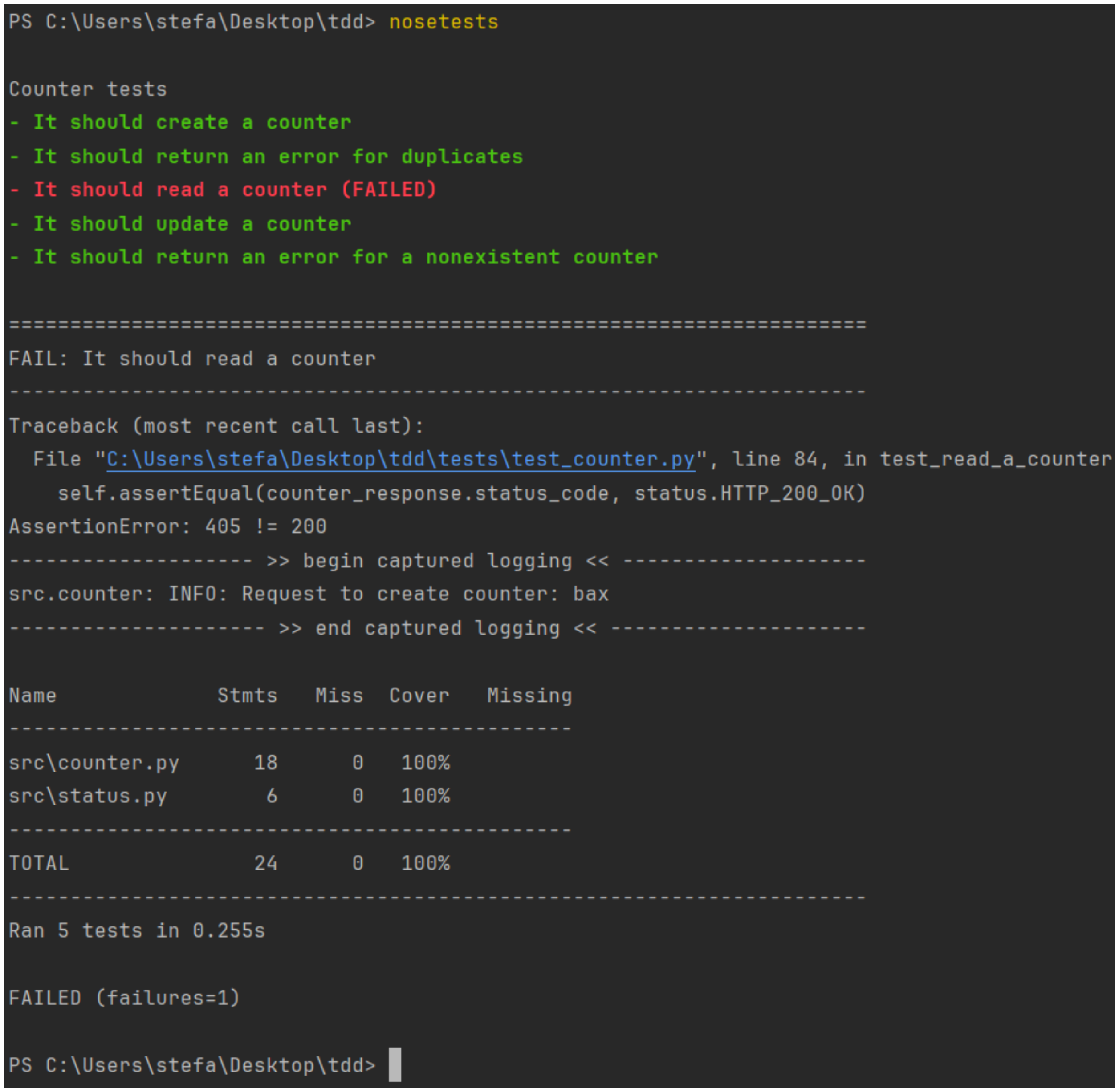
THE RED PHASE OF TEST_READ_A_COUNTER(SELF)

test_read_a_counter(self) code



```
new *
def test_read_a_counter(self):
    """It should read a counter"""
    # 1. Make a call to read_counter(name)
    counter_call = self.client.post('/counters/bax')
    self.assertEqual(counter_call.status_code, status.HTTP_201_CREATED)

    # 2. Ensure that it returned a successful return code
    counter_response = self.client.get('/counters/bax')
    self.assertEqual(counter_response.status_code, status.HTTP_200_OK)

    # 3. Read the value of the counter and store it in a variable
    counter_value = counter_response.json['bax']

    # 4. Ensure that it returned a successful return code
    self.assertEqual(counter_value,  second: 0)
```

From the screenshot below this block of text, we can see that after defining our *test_read_a_counter(self)* test, we are in the RED phase because we are writing a test case for a function that does not yet exist, which makes sense why we are in the red phase because this is the nature of following the coding paradigm of Test Driven Development (TDD), where we first write the test cases for the code we wish we had and then writing the code to make the test cases pass.

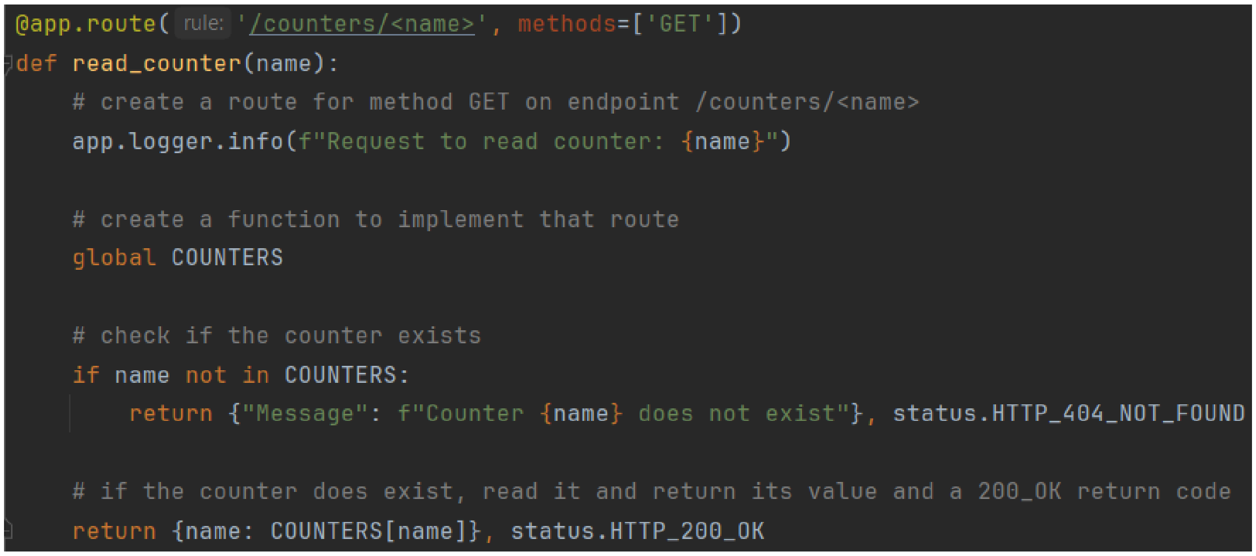


```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should read a counter (FAILED)
- It should update a counter
- It should return an error for a nonexistent counter

======================================================================
FAIL: It should read a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\stefa\Desktop\tdd\tests\test_counter.py", line 84, in test_read_a_counter
    self.assertEqual(counter_response.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: bax
--------------------- >> end captured logging << ---------------------

Name                Stmts   Miss  Cover   Missing
-----------------------------------------------
src\counter.py         18      0   100%
src\status.py           6      0   100%
-----------------------------------------------
TOTAL                  24      0   100%
----------------------------------------------------------------------
Ran 5 tests in 0.255s

FAILED (failures=1)

PS C:\Users\stefa\Desktop\tdd>
```

The result of writing a test case (test_read_a_counter(self)) for a block of code that doesn't exist yet (read_counter(name))

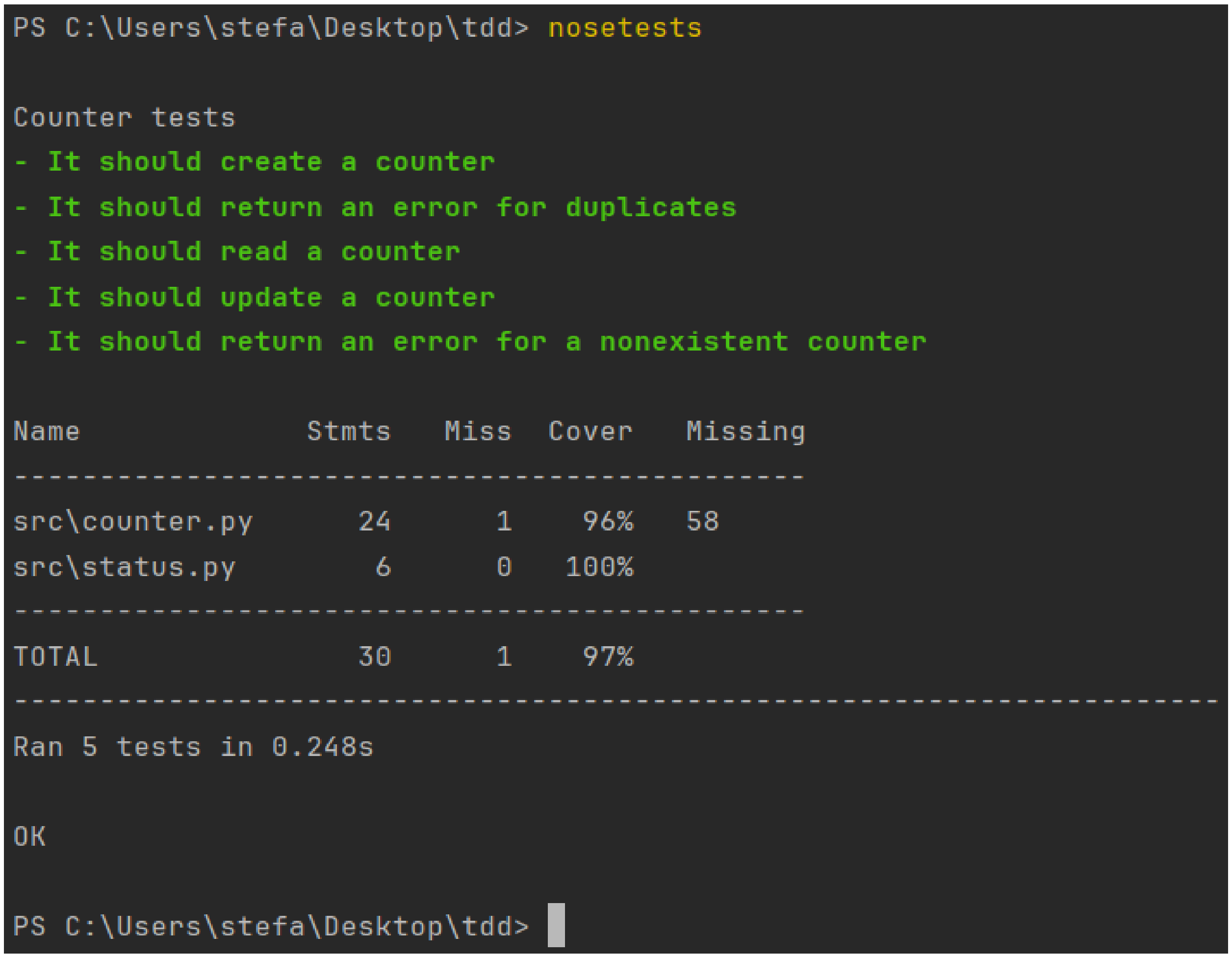
THE GREEN PHASE OF TEST_READ_A_COUNTER(SELF)

read_counter(name) code



```
@app.route( rule: '/counters/<name>', methods=['GET'])
def read_counter(name):
    # create a route for method GET on endpoint /counters/<name>
    app.logger.info(f"Request to read counter: {name}")

    # create a function to implement that route
    global COUNTERS

    # check if the counter exists
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND

    # if the counter does exist, read it and return its value and a 200_OK return code
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

From the screenshot below this block of text, we can see that after defining the **read_counter(name)** function for our **test_read_a_counter(self)** test case, we can see that after running "nosetests" we pass all of our current counter tests without any issue.

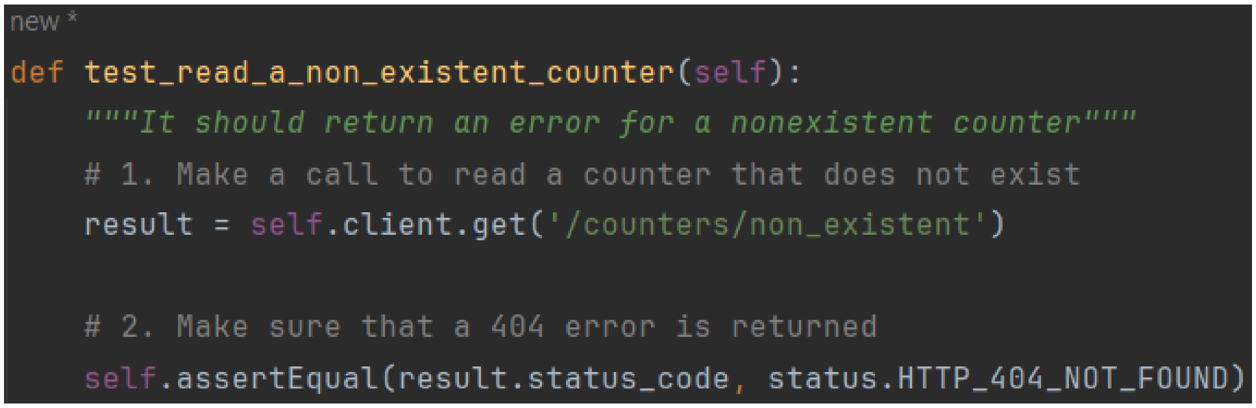


```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should read a counter
- It should update a counter
- It should return an error for a nonexistent counter

Name                 Stmts   Miss  Cover   Missing
----------------------------------------------
src\counter.py          24      1    96%   58
src\status.py            6      0   100%
----------------------------------------------
TOTAL                   30      1    97%
----------------------------------------------------------------------
Ran 5 tests in 0.248s

OK

PS C:\Users\stefa\Desktop\tdd>
```

THE BLUE PHASE OF TEST_READ_A_COUNTER(SELF)

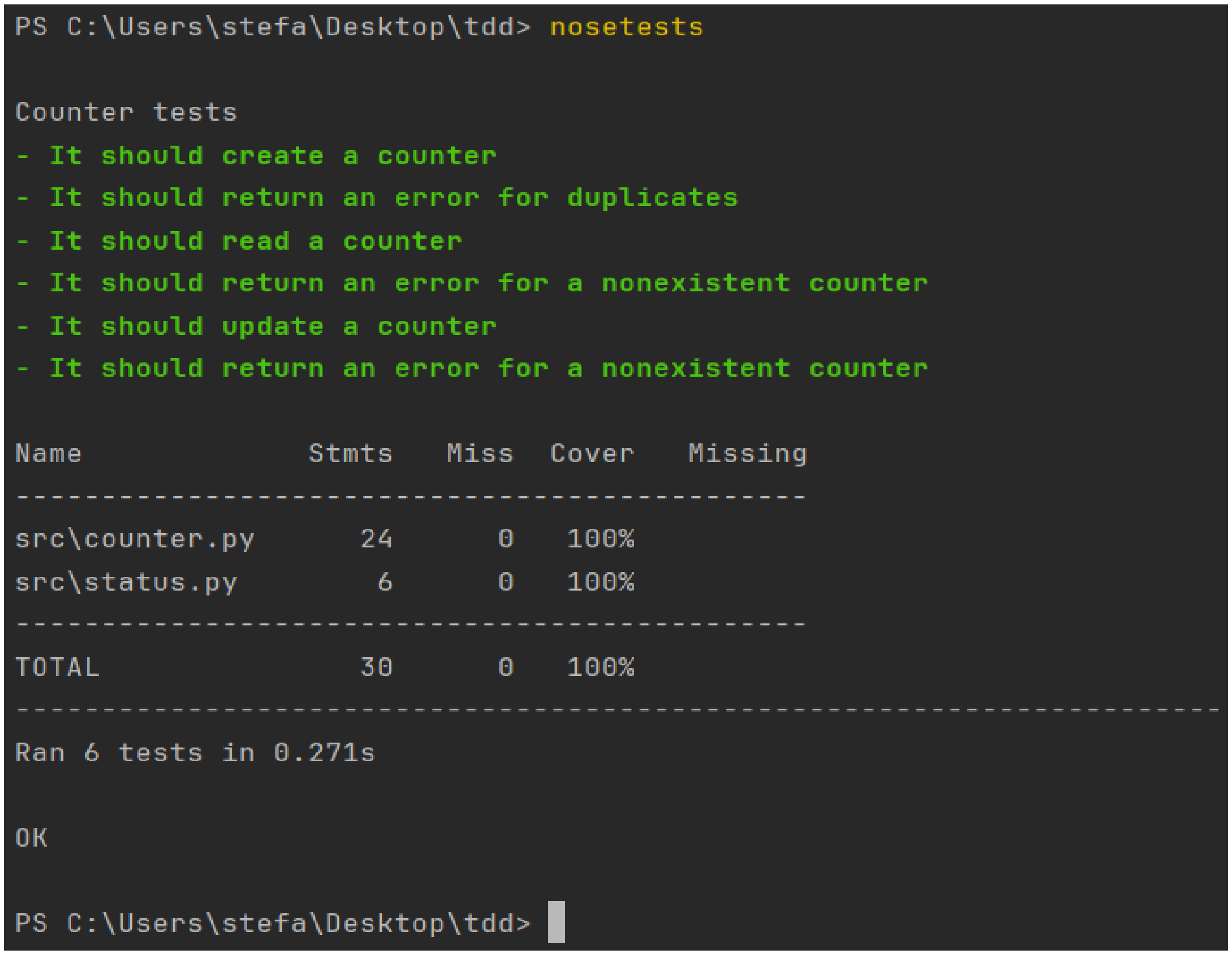
From the aforementioned screenshot above, we can see that we do NOT have 100% coverage for all of the statements in counter.py, as line 58 is the only line that is missing. Therefore, we have to refactor our test code in order for this specific line to be accounted for, and this line happens to be interested in whether or not we test the case in which we try to update a non-existent counter, and we currently do not have a test case for that scenario. Therefore, we go ahead and account for that scenario by adding the following test case test_counter.py:

test_read_a_non_existent_countet(self) code



```
new *
def test_read_a_non_existent_counter(self):
    """It should return an error for a nonexistent counter"""
    # 1. Make a call to read a counter that does not exist
    result = self.client.get('/counters/non_existent')

    # 2. Make sure that a 404 error is returned
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

After adding the above test case in test_counter.py, we go ahead and run "nosetests" and we can now see that our code coverage is now 100%, as can be seen in the screenshot attached below:



```
PS C:\Users\stefa\Desktop\tdd> nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should read a counter
- It should return an error for a nonexistent counter
- It should update a counter
- It should return an error for a nonexistent 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 0.271s

OK

PS C:\Users\stefa\Desktop\tdd>
```