Thomas Bryant | NSHE: 2000193948 | CS472-1002
Software Testing

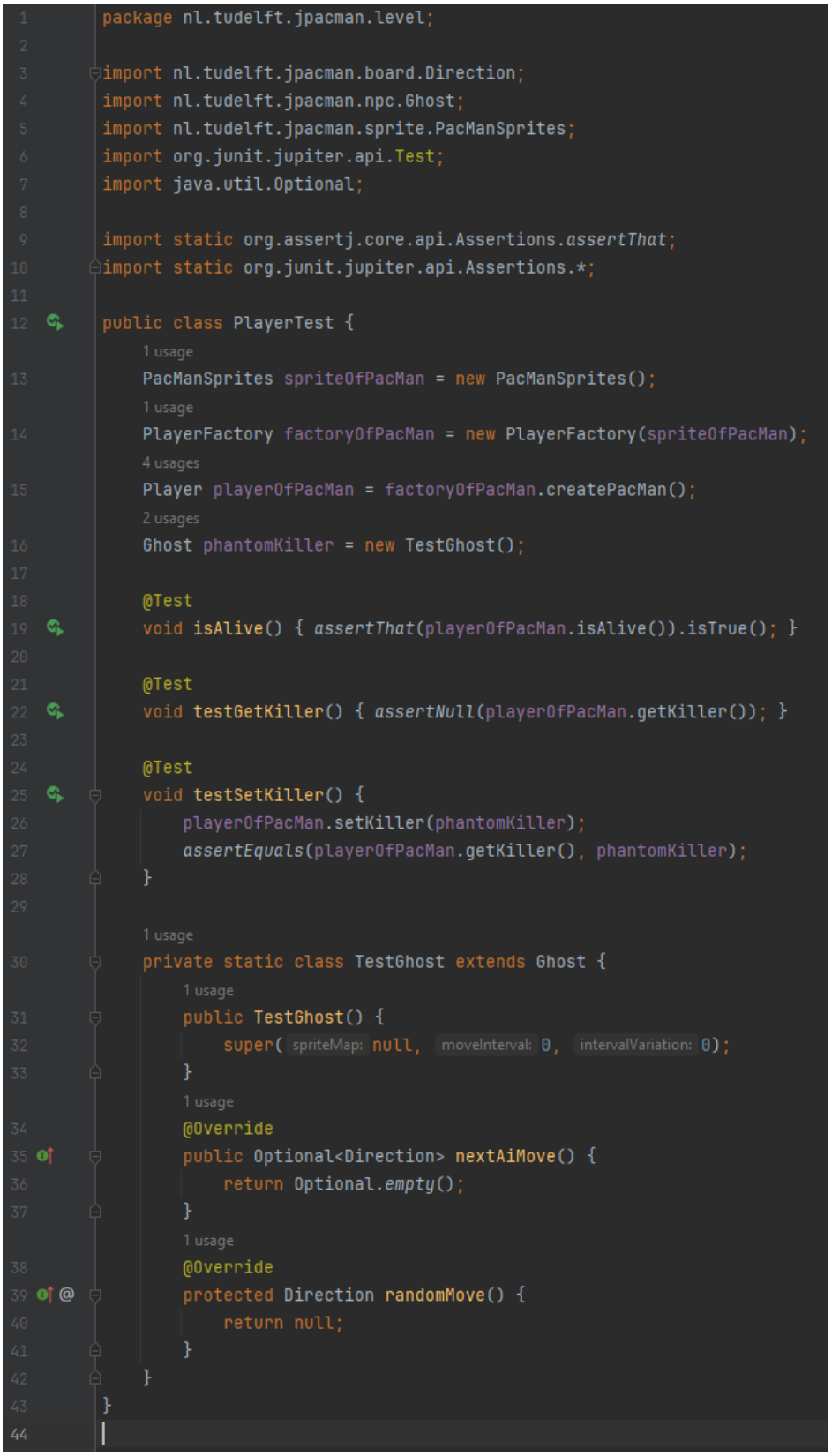
Task 1: Is The Coverage Good Enough?

The initial coverage for this program is abysmal, with only about 3% coverage of classes (2/55), 1% coverage of methods (5/312), and 1% coverage of lines (14/1137). Given that the standard for good coverage is at least 90%, this initial coverage is laughably low - not even mediocre coverage let alone good coverage.

Task 2: Adding More Unit Tests to Increase Coverage

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

import nl.tudelft.jpacman.sprite.Sprite;
import org.junit.jupiter.api.Test;

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

public class BoardTest {
    9 usages
    Square sampleSquare = new Square() {
        @java.lang.Override
        public boolean isAccessibleTo(Unit unit) {
            return false;
        }
        1 usage
        @java.lang.Override
        public Sprite getSprite() {
            return null;
        }
    };
    1 usage
    Square[][] sampleBoard = {
        {sampleSquare, sampleSquare, sampleSquare},
        {sampleSquare, sampleSquare, sampleSquare},
        {sampleSquare, sampleSquare, sampleSquare},
    };
    4 usages
    Board board = new Board(sampleBoard);
    1 usage
    int height = board.getHeight();
    1 usage
    int expectedHeight = 3;

    @Test
    void testGetHeight() { assertEquals(expectedHeight, height); }
}
```

Figure 1: Unit Test for …/`board/Board.getHeight`



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

import nl.tudelft.jpacman.board.Direction;
import nl.tudelft.jpacman.npc.Ghost;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;
import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.*;

public class PlayerTest {
    1 usage
    PacManSprites spriteOfPacMan = new PacManSprites();
    1 usage
    PlayerFactory factoryOfPacMan = new PlayerFactory(spriteOfPacMan);
    4 usages
    Player playerOfPacMan = factoryOfPacMan.createPacMan();
    2 usages
    Ghost phantomKiller = new TestGhost();

    @Test
    void isAlive() { assertThat(playerOfPacMan.isAlive()).isTrue(); }

    @Test
    void testGetKiller() { assertNull(playerOfPacMan.getKiller()); }

    @Test
    void testSetKiller() {
        playerOfPacMan.setKiller(phantomKiller);
        assertEquals(playerOfPacMan.getKiller(), phantomKiller);
    }

    1 usage
    private static class TestGhost extends Ghost {
        1 usage
        public TestGhost() {
            super( spriteMap: null,  moveInterval: 0,  intervalVariation: 0);
        }
        1 usage
        @Override
        public Optional<Direction> nextAiMove() {
            return Optional.empty();
        }
        1 usage
        @Override
        protected Direction randomMove() {
            return null;
        }
    }
}
```

Figure 2: Unit Tests for …/`level/Player.getKiller` and …/`level/Player.setKiller`

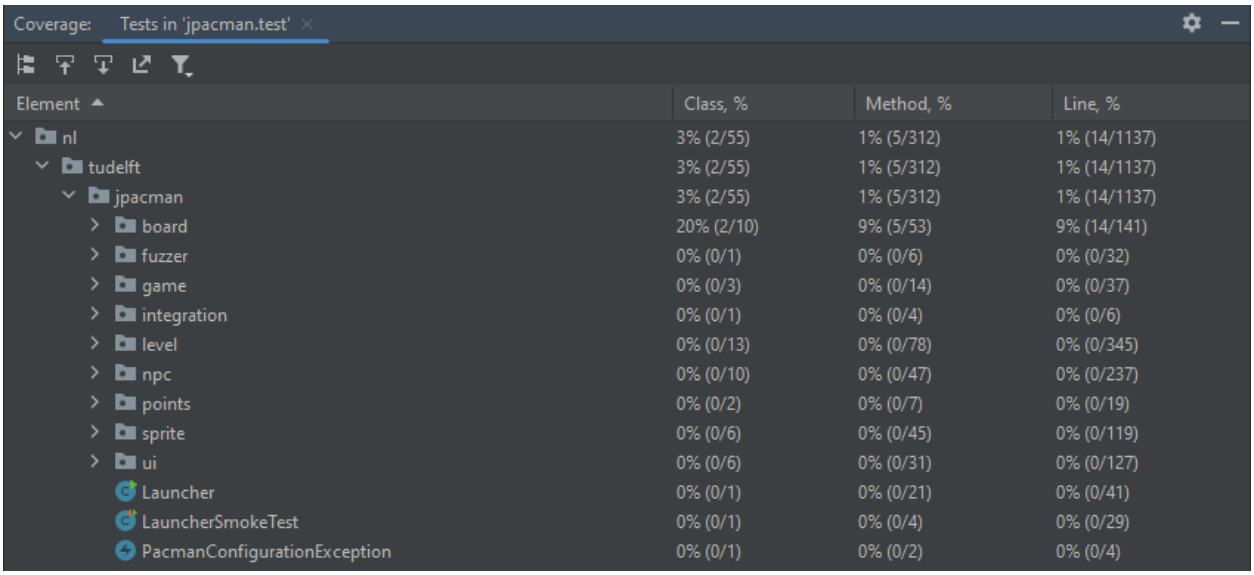


Figure 3: Test Coverage Before Implementing New Tests

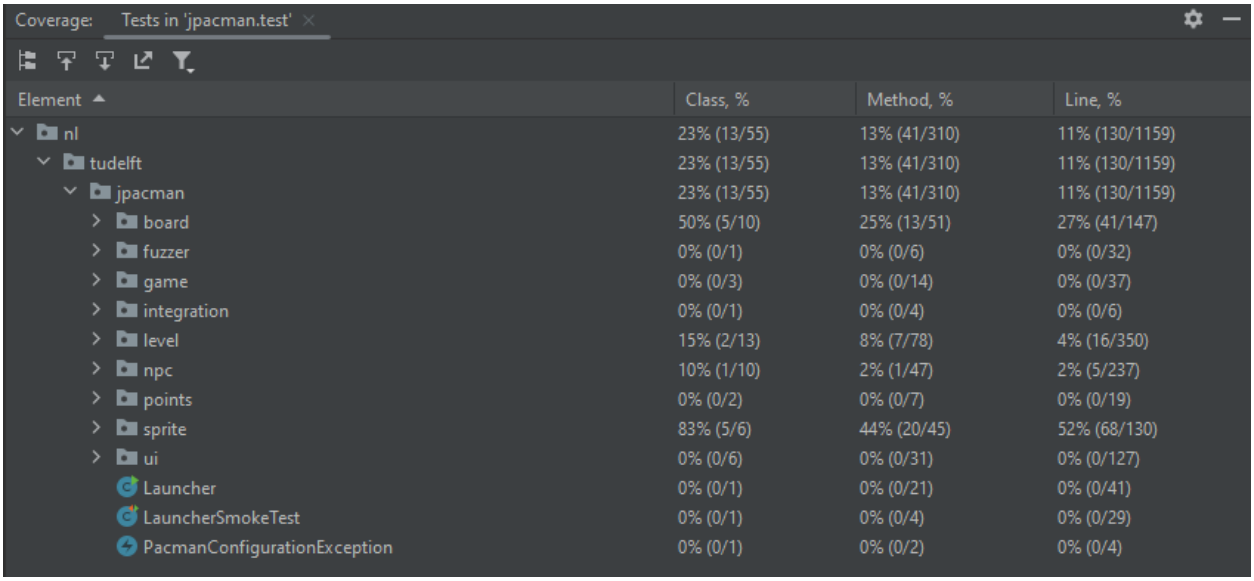
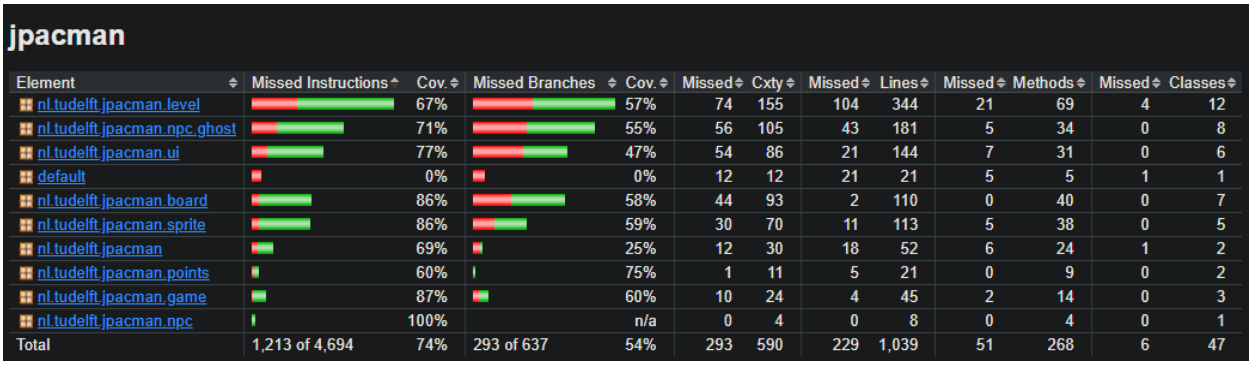


Figure 4: Test Coverage After Implementing New Tests

Task 3: JaCoCo Report On Pacman



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

Figure 5: JaCoCo Report

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 from JaCoCo differ from the results from IntelliJ/Gradle in that while the report from IntelliJ mainly focuses on coverage in terms of classes, methods, and lines of code, JaCoCo provides a more in depth analysis based on instructions and branches (presenting the percentage based on these metrics) along with providing the ratio of missed-to-total classes, methods, etc.

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

I found the visualization from the JaCoCo report to be extremely helpful as not only does it give a clear representation of what was missed in testing, it allows me to dive further into the code and explicitly see which exact lines have been missed during testing. I feel that this will allow me to write more meaningful unit testing to provide maximum coverage.

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

While I do appreciate the built-in coverage window IntelliJ provides to allow one a more streamlined approach to unit test, I much rather prefer the more in depth analysis provided to me by the JaCoCo report – especially with the ability to see highlighted code demonstrating what still requires coverage.

Task 4: Working With Python Test Coverage

```
72
73      def test_from_dict(self):
74          """ Test setting attributes from a dictionary """
75          data = ACCOUNT_DATA[self.rand]
76          account = Account()
77          account.from_dict(data)
78          self.assertEqual(account.name, data["name"])
79          self.assertEqual(account.email, data["email"])
80          self.assertEqual(account.phone_number, data.get("phone_number"))
81          self.assertEqual(account.disabled, data.get("disabled"))
82          self.assertEqual(account.date_joined, data.get("date_joined"))
83
84      def test_update(self):
85          """ Test updating an account """
86          data = ACCOUNT_DATA[self.rand]
87          account = Account(**data)
88          account.create()
89          new_name = "UpdatedName"
90          account.name = new_name
91          account.update()
92          updated_account = Account.find(account.id)
93          self.assertEqual(updated_account.name, new_name)
94
95      def test_update_empty_id(self):
96          """ Test updating an account with an empty ID """
97          data = ACCOUNT_DATA[self.rand]
98          account = Account(**data)
99          with self.assertRaises(DataValidationError):
100             account.update()
101
102     def test_delete(self):
103         """ Test deleting an account """
104         data = ACCOUNT_DATA[self.rand]
105         account = Account(**data)
106         account.create()
107         account.delete()
108         deleted_account = Account.find(account.id)
109         self.assertIsNone(deleted_account)
110
111     def test_find(self):
112         """ Test finding an account by ID """
113         data = ACCOUNT_DATA[self.rand]
114         account = Account(**data)
115         account.create()
116         found_account = Account.find(account.id)
117         self.assertEqual(found_account, account)
118
```

Figure 6: Unit Tests In Python for `test_coverage`

```
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 19.283s

OK
```

Figure 7: 100% Coverage After Implementing Tests

Task 5: TDD

TTD For PUT Method

Starting with the RED phase, I built a testing method following the step-by-step instructions given in the assignment:

Step 1: Make a call to Create a counter.
Step 2: Ensure that it returned a successful return code.
Step 3: Check the counter value as a baseline.
Step 4: Make a call to update the counter that I just created.
Step 5: Ensure that it returns a successful return code.
Step 6: Check that the counter value is one more than the baseline.

```
40      def test_update_a_counter(self):
41          """It should update a counter"""
42          resultCreate = self.client.post('/counters/hoge')                    # Step 1
43          self.assertEqual(resultCreate.status_code, status.HTTP_201_CREATED)  # Step 2
44          baseValue = resultCreate.json['hoge']                                # Step 3
45          resultUpdate = self.client.put('/counters/hoge')                     # Step 4
46          self.assertEqual(resultUpdate.status_code, status.HTTP_200_OK)       # Step 5
47          updatedValue = resultUpdate.json['hoge']
48          self.assertEqual(updatedValue, baseValue + 1)                        # Step 6
49
```

Figure 8: TDD Test for PUT Method

This, of course, causes the test to fail (specifically return the `AssertionError: 405 != 200` error). Referring to the `status.py` file refers this as a `HTTP_405_METHOD_NOT_ALLOWED` error, meaning that PUT has not been implemented, as expected.

```
PS D:\CS472\Ast2.3\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 "D:\CS472\Ast2.3\tdd\tests\test_counter.py", line 46, in test_update_a_counter
    self.assertEqual(resultUpdate.status_code, status.HTTP_200_OK)          # Step 5
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: hoge
--------------------- >> 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 0.181s

FAILED (failures=1)

PS D:\CS472\Ast2.3\tdd>
```

Figure 9: RED Phase showing `Assertion Error: 405 != 200`

With both the error and the testing method written, we are given a good blueprint as to how the PUT method should be implemented, this is reinforced with the next set of step-by-step instructions:

Step 1: Create a route for method `PUT` on endpoint `/counters/<name>`.
Step 2: Create a function to implement that route.
Step 3: Increment the counter by 1.
Step 4: Return the new counter and a `200_OK` return code.

```python
@app.route('/counters/<name>', methods=['PUT'])          # Step 1
def update_counter(name):                                # Step 2
    """Update a counter by incrementing it by 1"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    COUNTERS[name] += 1                                  # Step 3
    return {name: COUNTERS[name]}, status.HTTP_200_OK    # Step 4

```

Figure 10: TDD Implementation for PUT Method

Now in the GREEN phase, with the PUT method implemented, I re-ran the coverage test, and as expected, return with full coverage and the test succeeded.

```
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     16      0   100%
src\status.py       6      0   100%
---------------------------------------------
TOTAL              22      0   100%
----------------------------------------------------------------------
Ran 3 tests in 0.176s

OK

PS D:\CS472\Ast2.3\tdd>
```

Figure 11: GREEN Phase showing full coverage and success

For REFACTORING, I originally wanted to include a conditional in PUT to check for empty counter names and pass a 404 error should no name be found. However, when attempting to include a test for it, I received a 500 error, which is not included in the `status.py` file – leading me to conclude that this cannot be fully tested by only modifying `counter.py` or tested only in `test_counter.py`. After removing these modifications, I was able to reclaim my 100% coverage in GREEN.

TDD For GET Method

Based on the documentation for HTTP methods and REST guidelines, the GET method only *requests to retrieve resources representation/information only – and not modify it in any way*[1]. With that in mind, I built my test to create a new counter and then call it with GET to see if I get a successful status – in this case a `200` status.

```
50      def test_read_a_counter(self):
51          """It should read a counter"""
52          resultCreate = self.client.post('/counters/fuga')
53          self.assertEqual(resultCreate.status_code, status.HTTP_201_CREATED)
54          resultRead = self.client.get('/counters/fuga')
55          self.assertEqual(resultRead.status_code, status.HTTP_200_OK)
56          self.assertEqual(resultRead.json['fuga'], 0)
57
```

Figure 12: TDD Test For GET Method

As before in the previous RED phase, this RED phase also generated an expected error – specifically the `405` error indicating that the method has not been implemented.

```
PS D:\CS472\Ast2.3\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

======================================================================
FAIL: It should read a counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\CS472\Ast2.3\tdd\tests\test_counter.py", line 55, in test_read_a_counter
    self.assertEqual(resultRead.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
-------------------- >> begin captured logging << --------------------
src.counter: INFO: Request to create counter: fuga
-------------------- >> end captured logging << --------------------

Name             Stmts   Miss  Cover   Missing
----------------------------------------------
src\counter.py      16      0   100%
src\status.py        6      0   100%
----------------------------------------------
TOTAL               22      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.183s
```

Figure 13: RED Phase showing `Assertion Error: 405 != 200`

For the GREEN Phase, I took this error and the previously implemented GET test to implement my method for GET.

```
30  @app.route('/counters/<name>', methods=['GET'])
31  def read_counter(name):
32      """Read a counter"""
33      app.logger.info(f"Request to read counter: {name}")
34      global COUNTERS
35      return {name: COUNTERS[name]}, status.HTTP_200_OK
36
```

Figure 14: TDD Implementation for GET Method

Once implemented, I reran the test and it shows full coverage and returned successfully

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

Name             Stmts   Miss  Cover   Missing
----------------------------------------------
src\counter.py      20      0   100%
src\status.py        6      0   100%
----------------------------------------------
TOTAL               26      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.181s

OK

PS D:\CS472\Ast2.3\tdd>
```

Figure 15: GREEN Phase showing full coverage and success

For REFACTORING, I implemented the check to determine if a bad request has been passed, which if it did, then an appropriate `404` error should return indicating that the name does not exist. After making the change in both the test and implementation for GET, I was able to maintain successful testing and full coverage.

```
50      def test_read_a_counter(self):
51          """It should read a counter"""
52          resultCreate = self.client.post('/counters/fuga')
53          self.assertEqual(resultCreate.status_code, status.HTTP_201_CREATED)
54          resultRead = self.client.get('/counters/fuga')
55          self.assertEqual(resultRead.status_code, status.HTTP_200_OK)
56          self.assertEqual(resultRead.json['fuga'], 0)
57          badResult = self.client.get('/counters/doesnotexist')
58          self.assertEqual(badResult.status_code, status.HTTP_404_NOT_FOUND)
59
```

Figure 16: REFACTOR Test for GET to test for 404 error

```
30  @app.route('/counters/<name>', methods=['GET'])
31  def read_counter(name):
32      """Read a counter"""
33      app.logger.info(f"Request to read counter: {name}")
34      global COUNTERS
35      if name not in COUNTERS:
36          return {"Message":f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
37      return {name: COUNTERS[name]}, status.HTTP_200_OK
38
```

Figure 17: REFACTOR Implementation for GET to check for name not found

```
PS D:\CS472\Ast2.3\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

Name             Stmts   Miss  Cover   Missing
----------------------------------------------
src\counter.py      22      0   100%
src\status.py        6      0   100%
----------------------------------------------
TOTAL               28      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.187s
```

Figure 18: REFACTOR for GET to verify test was successfully and full coverage

References

[1] Gupta, L., & Gupta, L. (2023, November 4). *HTTP methods*. REST API Tutorial. https://restfulapi.net/http-methods/