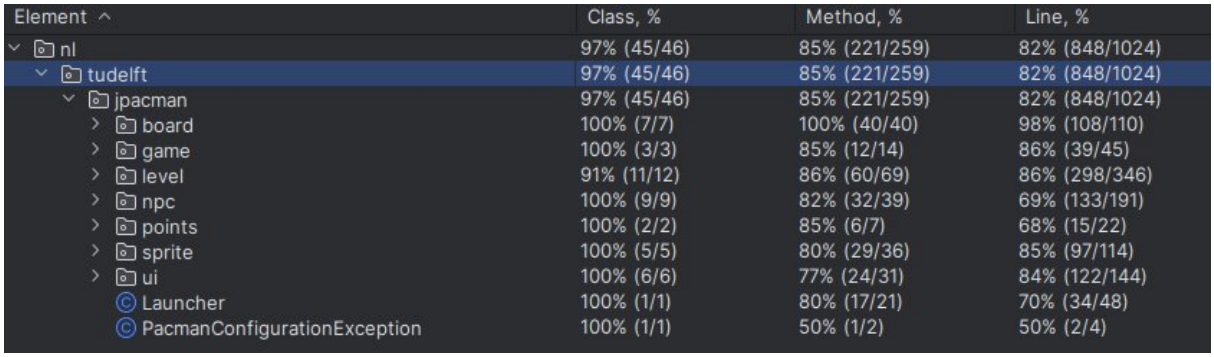
Dr Businge, Project Coordinator
Garrett Prentice, Project Manager
**RE: Code Coverage & Evaluation (JaCoCo v IntelliJ)**

I believe that the coverage results from IntelliJ are better for uses that prefer integrated environments. When I use IntelliJ, I use the built-in code coverage, as it provides me with all that I need and allows me to jump directly to editable source code. JaCoCo is a great substitute for users that do not use IntelliJ; VS Code, or Neovim.

**Similarity**
I have similar results between both JaCoCo and IntelliJ's coverage reporting values. However, IntelliJ gives me a line based breakdown of the code. JaCoCo gives me a functional breakdown of the code. For example, in Figure 1 and 2, IntelliJ reports by line, method, and class. These are things that only require touching to change. But JaCoCo coverage is more focused on code branching and combination of input. For this reason, JaCoCo might be better for quality of test.

*Figure 1: IntelliJ coverage report for JPacman repo*



| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| nl | 97% (45/46) | 85% (221/259) | 82% (848/1024) |
| tudelft | 97% (45/46) | 85% (221/259) | 82% (848/1024) |
| jpacman | 97% (45/46) | 85% (221/259) | 82% (848/1024) |
| board | 100% (7/7) | 100% (40/40) | 98% (108/110) |
| game | 100% (3/3) | 85% (12/14) | 86% (39/45) |
| level | 91% (11/12) | 86% (60/69) | 86% (298/346) |
| npc | 100% (9/9) | 82% (32/39) | 69% (133/191) |
| points | 100% (2/2) | 85% (6/7) | 68% (15/22) |
| sprite | 100% (5/5) | 80% (29/36) | 85% (97/114) |
| ui | 100% (6/6) | 77% (24/31) | 84% (122/144) |
| Launcher | 100% (1/1) | 80% (17/21) | 70% (34/48) |
| PacmanConfigurationException | 100% (1/1) | 50% (1/2) | 50% (2/4) |

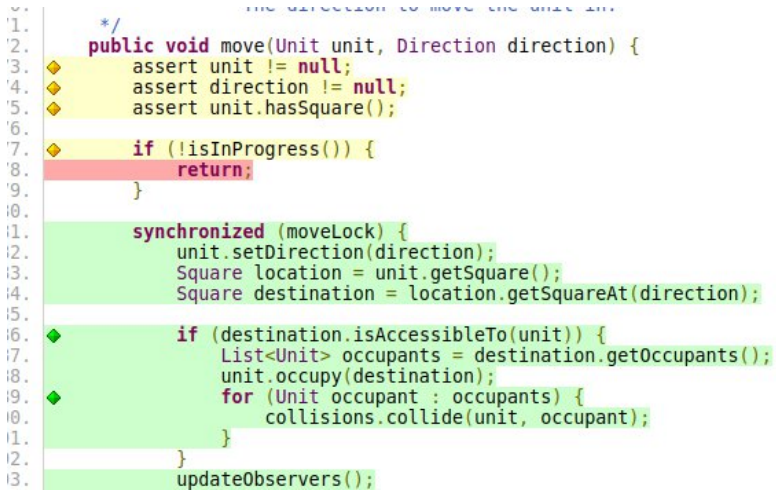
*Figure 2: JaCoCo coverage report for JPacman repo*

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nl.tudelft.jpacman.npc.ghost | | 71% | | 55% | 56 | 105 | 43 | 181 | 5 | 34 | 0 | 8 |
| nl.tudelft.jpacman.level | | 85% | | 69% | 51 | 155 | 37 | 344 | 7 | 69 | 1 | 12 |
| 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 | | 71% | | 25% | 11 | 30 | 16 | 52 | 5 | 24 | 0 | 2 |
| nl.tudelft.jpacman.points | | 60% | | 75% | 1 | 11 | 5 | 21 | 0 | 9 | 0 | 2 |
| nl.tudelft.jpacman.game | | 89% | | 60% | 9 | 24 | 2 | 45 | 1 | 14 | 0 | 3 |
| nl.tudelft.jpacman.npc | | 100% | | n/a | 0 | 4 | 0 | 8 | 0 | 4 | 0 | 1 |
| Total | 963 of 4,694 | 79% | 273 of 637 | 57% | 268 | 590 | 158 | 1,039 | 35 | 268 | 2 | 47 |

As shown above, JaCoCo reports a lower code coverage than IntelliJ. In JaCoCo, it reports only 79% for missed instructions, which is closest to line coverage in IntelliJ, which is reported as 82%. For a full reference to the changes that were made to the JPacman repository, see my fork of the GitHub repository. (https://github.com/S1robe/jpacman)

**Usefulness**
I believe that seeing the source code in the browser is the most useful feature of JaCoCo. I can match this directly to my test cases without opening the project. For example, in Figure 3, I can navigate to a specific section of code that was not covered and then write test cases to cover it.

*Figure 3: JaCoCo coverage of Level.java#move()*



```
    */
    public void move(Unit unit, Direction direction) {
        assert unit != null;
        assert direction != null;
        assert unit.hasSquare();

        if (!isInProgress()) {
            return;
        }

        synchronized (moveLock) {
            unit.setDirection(direction);
            Square location = unit.getSquare();
            Square destination = location.getSquareAt(direction);

            if (destination.isAccessibleTo(unit)) {
                List<Unit> occupants = destination.getOccupants();
                unit.occupy(destination);
                for (Unit occupant : occupants) {
                    collisions.collide(unit, occupant);
                }
            }
            updateObservers();
```

As shown above, the code in yellow is assertions and branches that were checked. The section shown in red is the branch that has not yet been covered, and the green is what has been covered.

**Preference**

I have taken this course at a different school, we used JaCoCo, however, when paired with IntelliJ it is not as powerful. But, when developing a report to senior developer or even as the project lead, it can make my life easy. Receiving a zip file with these test case reports is sometimes easier than going through the process of opening GitHub or IntelliJ to view test results.

Now that I use Neovim, a text editor IDE, it is much lighter weight, but test cases are harder to view. JaCoCo bundled with Gradle would be a solution to this problem without needing to integrate with a plugin or write any code. From a developer standing, I prefer IntelliJ. From a project manager perspective, I prefer JaCoCo.

**JPacman Test Coverage**

The following code snippets are from the JPacman repository. It is a pacman clone that is aimed at teaching how to unit test various parts of a program. Below is an acceptable test of the movement, map parser, and collision detection.

*Figure 4: Test of Movement in JPacman*

```
    @Test
    void testPlayerMove(){
        Player pacman = launcher.getGame().getPlayers().get(0);
        Square pacmanStartSpot = pacman.getSquare();
        launcher.getGame().getLevel().move(pacman, Direction.NORTH);
// Never moved cause upward collision
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
        launcher.getGame().getLevel().move(pacman, Direction.SOUTH);
// Never moved cause downward collision
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());

        launcher.getGame().getLevel().move(pacman, Direction.EAST);
        launcher.getGame().getLevel().move(pacman, Direction.WEST);
// mvoed right so left should be same
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
        launcher.getGame().getLevel().move(pacman, Direction.WEST);
        launcher.getGame().getLevel().move(pacman, Direction.EAST);
// moved right back to start spot should be same as start.
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
    }
```

As shown above in figure 4, the player's movement is being test on the standard map. In this configuration, there are walls directly above and below, but not the sides. Since player exact position is not exposed, but the square that the player is on us. As a result, we can compare the square the player starts on *'pacmanStartSpot'* and compare it to the player's current position '*pacman.getSquare()*'. Because there are walls directly above and below the player, moving up (*north*) or down (*south*) should not change the player's position. Similarly, moving left (*west*) or right (*east*) will change the player's position.

*Figure 5: Test of Map Parser in JPacman*

```
//Load simple map, should load and throw no error
@Test
void testLoadSimpleMap() {
    Assertions.assertThatCode(() -> mp.parseMap("/simplemap.txt"))
        .doesNotThrowAnyException();
}

//Empty maps are not allowd, should throw error
@Test
void testLoadEmptyMap(){
    Assertions.assertThatThrownBy(() -> mp.parseMap("/emptyMap.txt"))
        .isInstanceOf(PacmanConfigurationException.class);
}

//A different non-std map, should not error
@Test
void testLoadCertainDeathMap() {
    Assertions.assertThatCode(() -> mp.parseMap("/certainDeathMap.txt"))
        .doesNotThrowAnyException();
}

//Map doesnt exist, should error.
@Test
void testNonExistentMap(){
    Assertions.assertThatThrownBy(() -> mp.parseMap("/IdontExistMap.txt"))
        .isInstanceOf(PacmanConfigurationException.class);
}

//Map doesnt have a player should not error.
@Test
void testMapMissingPlayer(){
    Assertions.assertThatCode(() -> mp.parseMap("/missingPlayer.txt"))
        .doesNotThrowAnyException();
}
```

Above, in figure 5, is a test of the map parser used to load and set the position of the player and NPC's (ghosts). The map if empty should produce an exception, as it should if the map does not exist. Test cases 2 and 4, in figure 5, show how this can be tested for using JUnit. The remaining tests account for situations where the map is non-standard, and is missing a player. Test cases 1, 3, and 5, are intentional behavior.

*Figure 6: Test of Collision with walls, coins, and a ghost*

```
    @Test
    void collideCoinThenWallThenGhost(){
 // Should collide with coin
        customLevel.move(pacman, Direction.WEST);
// coins add 10 to score, since this is the first one, score should be 10
        Assertions.assertThat(pacman.getScore()).isEqualTo(10);

        Square wall = pacman.squaresAheadOf(1);
        customLevel.move(pacman, Direction.WEST);
 // Square should be the same, it is unchanged.
        Assertions.assertThat(pacman.squaresAheadOf(1)).isEqualTo(wall);

//continue marching right, to the ghost.
        ScheduledExecutorService timer = Executors.newScheduledThreadPool(1);
        timer.scheduleAtFixedRate(() -> {
            if (pacman.isAlive()){
                customLevel.move(pacman, Direction.EAST);
            } else {
                timer.shutdown();
            }
        }, 0, 100, TimeUnit.MILLISECONDS);

// This will fail if the task is interrupted.
        Assertions.assertThatCode(() -> {
            if (timer.awaitTermination(1, TimeUnit.SECONDS))
                timer.shutdown();
            }).doesNotThrowAnyException();

// End of game should be caused by interaction with ghost.
        Assertions.assertThat(pacman.getKiller()).isInstanceOf(Ghost.class);
    }
```

The final test was of the collisions between the player (Pac-Man) and walls, coins, and ghosts. First, the player moves left, which causes the player in this map to pick up a coin. Coins add 10 to the player's score, so the score should be 10 if a coin was picked up. The player is also against the left wall, and so attempting to move left should be prevented by the wall. Then the player is moved right-ward until eventually bumping into a ghost causing the game to end, which verifies that the collision occurred.

**Python Test Coverage**

This segment of the report is regarding the ‘test_coverage’ repository provided to teach testing practices in python using ‘nose tests’. In this example, a model account is provided which will be tested. Below are snippets of each various test cases from the project. Included is a test of all the non-class methods provided within models/account.py: to_dict, from_dict, create, update, delete. For a full reference to the changes that were made to the repository, see my fork of the GitHub repository. (https://github.com/S1robe/CS472-test_coverage)

*Figure 7: Test of Account#create()*

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

The method Account#create is responsible for creating an account. By default the Account database is empty, as shown in figure 7. After this method is executed there should be exactly one (1) account in it.

*Figure 8: Test of Account#__repr__()*

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

The method Account#__repr__ is responsible for generating a representation of an account. In this case, shown in figure 8, the proper response when called, is the type (Account), and its name, which as tested above in figure 8, should return “<Account ‘Foo’>”.

*Figure 9: Test of Account#to_dict()*

```
def test_to_dict(self):
    """ Test account to dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    result = account.to_dict()
    self.assertEqual(account.name, result["name"])
    self.assertEqual(account.email, result["email"])
    self.assertEqual(account.phone_number, result["phone_number"])
    self.assertEqual(account.disabled, result["disabled"])
    self.assertEqual(account.date_joined, result["date_joined"])
```

The method Account#to_dict is responsible for turning an account into a dictionary. By comparing the results of the conversion to the original we are able to find they are the same.

*Figure 10: Test of Account#from_dict()*

```
def test_from_dict(self):
    """Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    result = account.to_dict()
    resultAccount = Account()
    resultAccount.from_dict(result)
    self.assertEqual(account.name, resultAccount.name)
    self.assertEqual(account.email, resultAccount.email)
    self.assertEqual(account.phone_number, resultAccount.phone_number)
    self.assertEqual(account.disabled, resultAccount.disabled)
    self.assertEqual(account.date_joined, resultAccount.date_joined)
```

The method Account#from_dict() is responsible for turning an account stored as a dictionary back into an Account object. By comparing an account that was made from this method with another that was created prior, we can prove that the two accounts are equal.

*Figure 11: Test of Account#update()*

```
    # Modify some fields
    result.disabled = not account.disabled
    result.name = "Test" + account.name
    newName = result.name
    result.id = randnum
    # Committ the changes
    result.update()
    # Reload the account
    result = Account.query.get(randnum)
    # Check for changes.
    self.assertEqual(result.name, newName)
    self.assertEqual(result.name, account.name)
    self.assertNotEqual(oldname, newName)
    self.assertEqual(account.disabled, result.disabled)
```

*Figure 12: Test of Account#update() fail*

```
def test_update_fail(self):
    """Test account update DataValidationError"""
    account = Account(**ACCOUNT_DATA[self.rand])
    account.name = "NewName"
    with self.assertRaises(DataValidationError):
        account.update()
```

The method Account#update is responsible for committing changes to an account. It will fail if the ID is not set. Figures 11 and 12 display the successful test and the fail test. In the first test, the ID used to retrieve the account is saved so that it can be updated later.

*Figure 13: Test of Account#delete()*

```
    randnum = self.rand
    account = Account.query.get(randnum)
    account.delete()
    self.assertIsNone(Account.query.get(randnum))
```

The method Account#delete will delete the account it is called on by removing it from the database and deleting its reference. As shown in figure 13, the account must not be in the database after it is deleted.

*Figure 14: Test of Account#find()*

```
    randNum = self.rand
    account = Account.find(randNum)
    result = ACCOUNT_DATA[(randNum-1)]

    self.assertEqual(account.name, result["name"])
    self.assertEqual(account.email, result["email"])
    self.assertEqual(account.phone_number, result["phone_number"])
    self.assertEqual(account.disabled, result["disabled"])
```

The method Account#find will attempt to find the account specified. It is important to note that the dictionary that the accounts are loaded from initially starts at 0 and so the reference numbers are off by 1. To adjust for this, we subtract 1 from the reference number ‘randnum’ to use the correct reference number.

*Figure 15: Test coverage report*

```
Test creating multiple Accounts ... ok
Test Account creation using known data ... ok
Test account delete ... ok
Test find account ... ok
Test account from dict ... ok
Test the representation of an account ... ok
Test account to dict ... ok
Test account update ... ok
Test account update DataValidationError ... ok

Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models/__init__.py       6      0   100%
models/account.py       40      0   100%
--------------------------------------------------
TOTAL                   46      0   100%
--------------------------------------------------
Ran 9 tests in 0.229s

OK
```

Shown above in figure 15 are the results of executing the tests shown previously in figures 7 through 14. In this case, 100% test coverage was achieved. This does not necessarily mean the code is bug-free, but does indicate the code meets the specifications of the tests.

**Test Driven Development**

This section of the report contains how the test driven development process could go. Test driven development begins by making tests that then define the behavior of the program. By writing tests and then writing code to fulfill those tests, the program is tested during development, which improves code quality. Below is each method and then how it was tested. For a full reference to the changes that were made to the repository, see my fork of the GitHub repository. (https://github.com/S1robe/CS472-tdd)

In order to demonstrate the red-green-refractor (blue) pattern that is emphasized for test driven development, this section will be in logical order of: red, before the code exists; green, after the code exists; refactor, condensing common cases.

**Red Phase #1**

Figure 16: Test Case #1

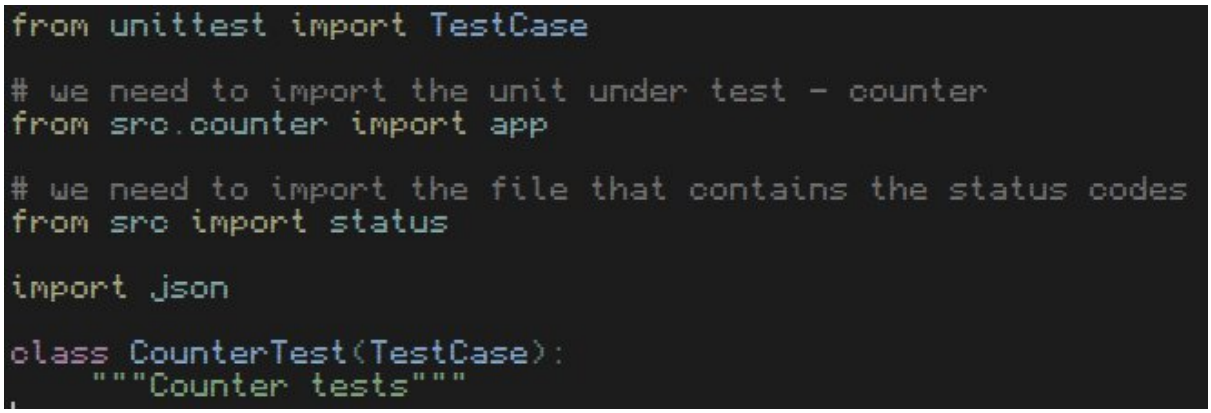
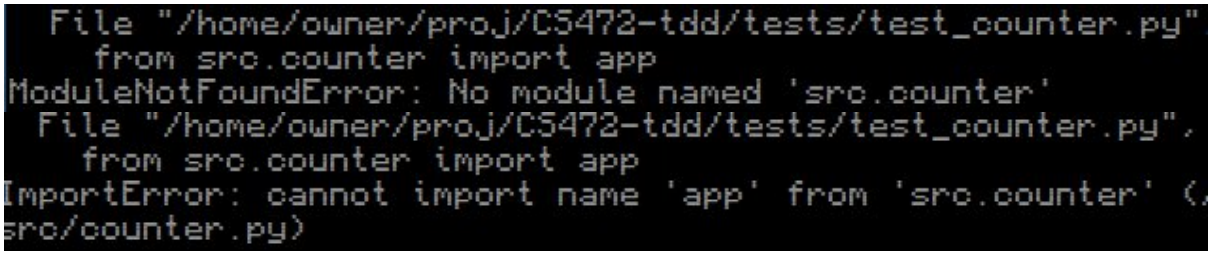



Figure 17: Red Phase #1




Shown above in figure 17, the project enters the red phase. There is no module called “counter” and so importing from “src.counter” produces a “ModuleNotFoundError”. This is resolved by making the file “counter.py”. The tests will still fail, so the phase does not change.

**Green Phase #1**

Figure 18: app creation, counter.py #1


Shown above in figure 18, creating the ‘app’ global variable to the file “counter.py” will address the initial red phase errors.

*Figure 19: Green Phase #1*

```
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
src/counter.py          2      0   100%
src/status.py           6      0   100%
-------------------------------------------------
TOTAL                   8      0   100%
-------------------------------------------------
Ran 0 tests in 0.080s

OK
```

Shown above in figure 19, the first green phase can be seen. The tests shown before in Figure 16 pass with the addition of the code in figure 18. Because the all tests have passed this is the green phase.

**Red Phase #2**

Figure 20: Red Phase #2

```
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
AssertionError: 404 != 201
```

Shown above is the second red phase encountered because of the test case shown below in Figure 21. The program specification expects a HTTP code of 201, not 404.

Figure 21: Test Case #2

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

Shown above is the test case that brings the project back to the red phase. The code above for “client.post” has no reference for the counter “/counters/foo” because it does not exist yet. So it fails with a error 404, instead of a 201, like shown in figure 20.

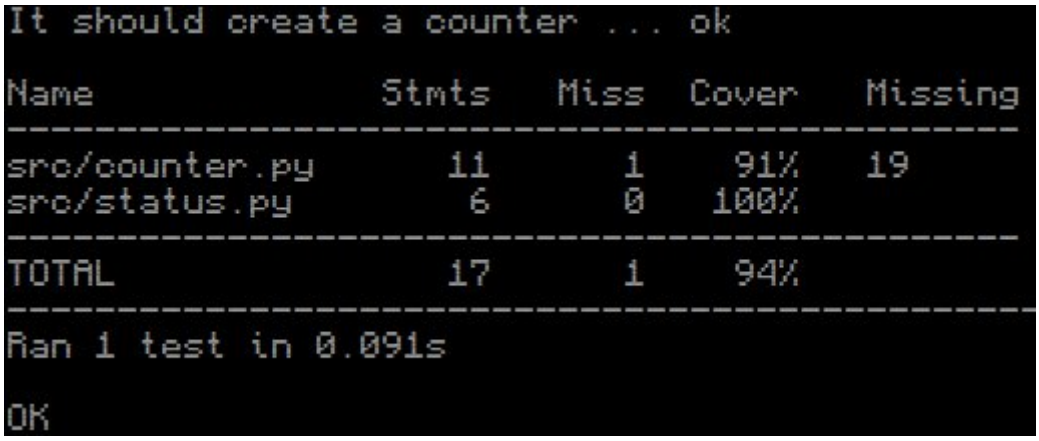
Figure 22: counter.py #2

```
@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
```

The method create_counter as shown in figure 22 should create a counter when invoked. It requires that the full name be given in the form “/counters/<name>”.

### Green Phase #2

Figure 23: Green Phase #2




As shown above, all tests have passed, so this is a green phase. However, the project is no longer at 100% coverage. This code follows the specifications of the current test suite, but is incomplete. More tests must be created until this code is 100% covered.

### Refactor Phase #1

Figure 24: Refactor Phase #1

```
def setUp(self):
    self.client = app.test_client()

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

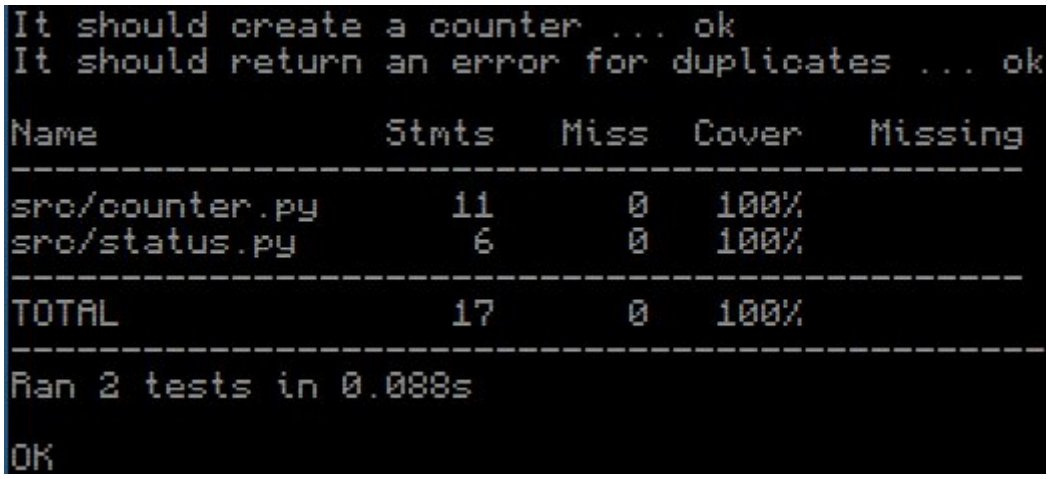
Figure 25: New test case for coverage

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

Shown above, in figure 25, is the new test created that covers the if condition shown in the middle of figure 22. This covers the duplicate counter state. However, during the creation of this test, we can also refactor our code to reduce duplicate lines. The line in figure 24, “setUp(self)” is required for all test cases because each test must have an app. This setup method will do this automatically before every test case.

### Green Phase #3

Figure 26: Green Phase #3




Shown above, in figure 26, is all test cases passing after refactoring. This is another green phase with 100% coverage. Now we start over and create more tests in accordance with the API.

### Red Phase #4

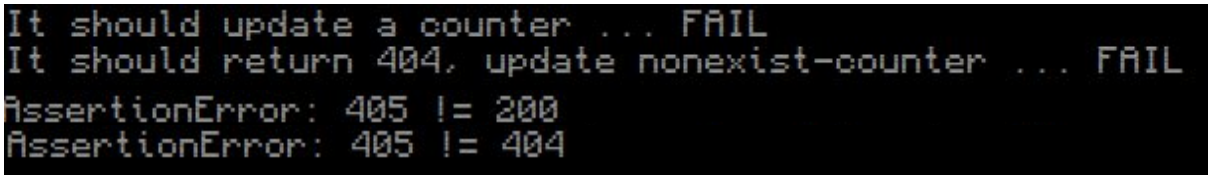
Figure 27: Test Cases 3 & 4

```
def test_update_a_counter(self):
    """It should update a counter"""
    name = "cntr"
    fqdn = "/counters/" + name
    result = self.client.post(fqdn)  # Create
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    baseline = result.get_json()[name]      # Result
    result = self.client.put(fqdn)  # Update
    self.assertEqual(result.status_code, status.HTTP_200_OK) # 200
    self.assertEqual(baseline+1, result.get_json()[name])

def test_update_a_counter_fail(self):
    """It should return 404, update nonexist-counter"""
    result = self.client.put("/counters/IDontExist")  # Update
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

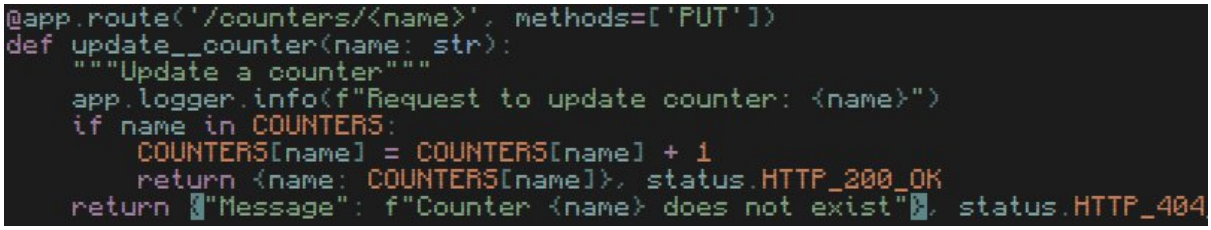
Shown above are the two new test cases that put the project back into the red phase of development. These test cases define how a counter should behave when an HTTP PUT request is made to the counter. It should increment by 1.

Figure 28: Red Phase #4




Shown above in figure 28, are the test cases failing signifying that the project is in the red phase of development.

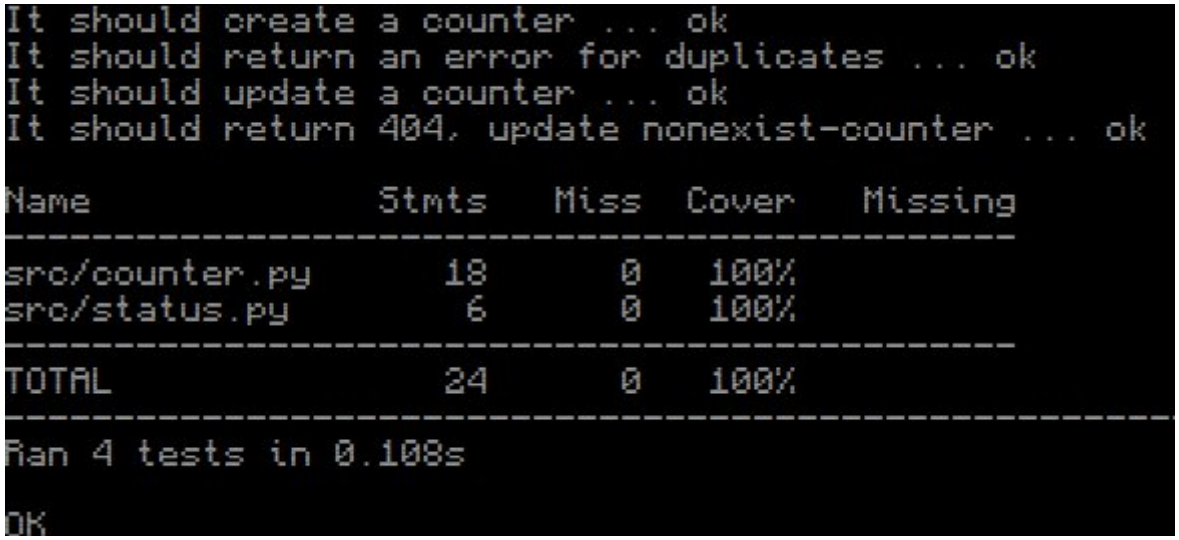
Figure 29: Update a counter, counter.py #3




Shown above is the code produced to address the test cases created in figure 27. The code returns a 200_OK response when the counter exists and can be written to, and responds with an appropriate 404 error when the counter does not exist.

**Green Phase #4**

Figure 30: Green Phase #4




Shown above, in figure 30 is all test cases passing in response to adding the code shown in figure 29. Therefore this code is complete and fulfills the requirements of the test cases. The cycle repeats again, moving back to the red phase after adding new test cases shown below in figure 31.

Figure 31: Test Cases 5 & 6

```
def test_get_a_counter(self):
    """It should read the counter"""
    result = self.client.post("/counters/cnt")
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = self.client.get("/counters/cnt")
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(0, result.get_json()["cnt"])

def test_get_a_counter_fail(self):
    """It should 404 when reading non-existent counter"""
    result = self.client.get("/counters/IDontExist")
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

Shown above are test cases 5 and 6, which define how a counter should behave when “gotten”. This means that when an HTTP GET request is made the counter’s value should be returned. If the counter does not exist then the app should return a 404_NOT_FOUND error.

Figure 32: Red Phase #5

```
It should read the counter ... FAIL
It should 404 when reading non-existent counter ... FAIL
AssertionError: 405 != 200
AssertionError: 405 != 404
```

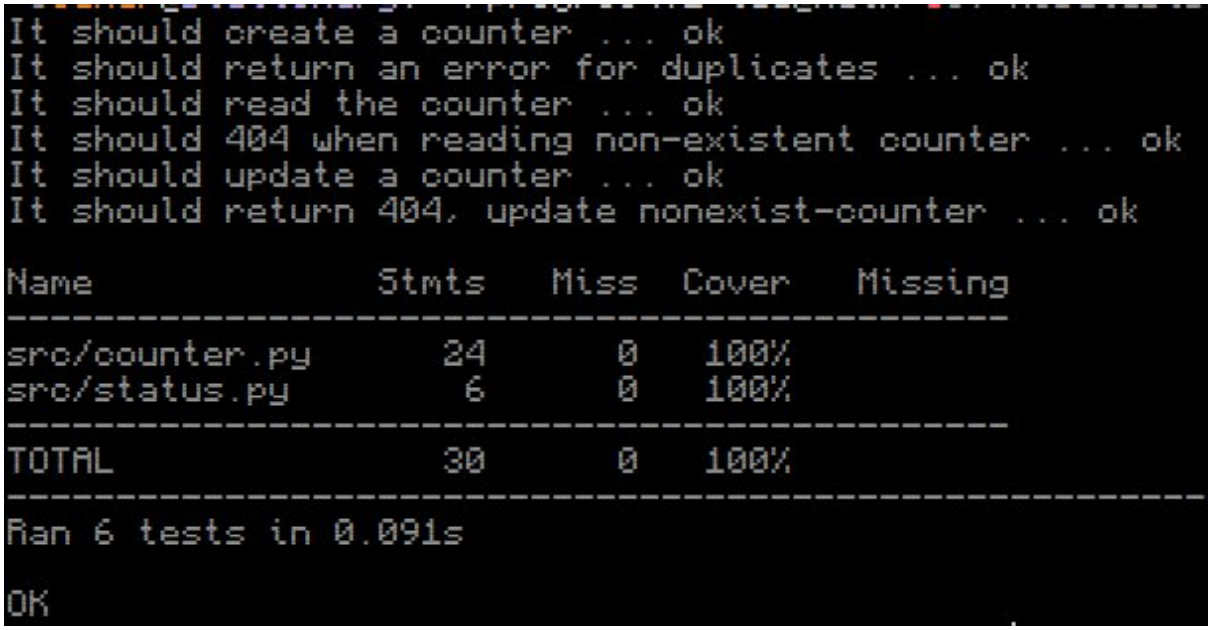
Shown above are the results of the test and its failures, signifying that the project is now, again, in the red phase of development.

Figure 33: Get a counter, counter.py #4

```
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name: str):
    """Get a counter value"""
    app.logger.info(f"Request counter value for: {name}")
    if name in COUNTERS:
        return {name: COUNTERS[name]}, status.HTTP_200_OK
    return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_
```

Shown above is the code that is based off the test cases shown in figure 31. As mentioned before, if the counter exists the value will be returned with a 200_OK status per the REST API. If the counter does not exist, then an appropriate error message is returned with a 404_NOT_FOUND error.

Figure 34: Green Phase #5- Final




Finally, shown above in figure 34, the project enters the final green phase with 100% coverage. This implies that the code was well tested because it was entirely based off test cases that were created with the design requirements in mind.