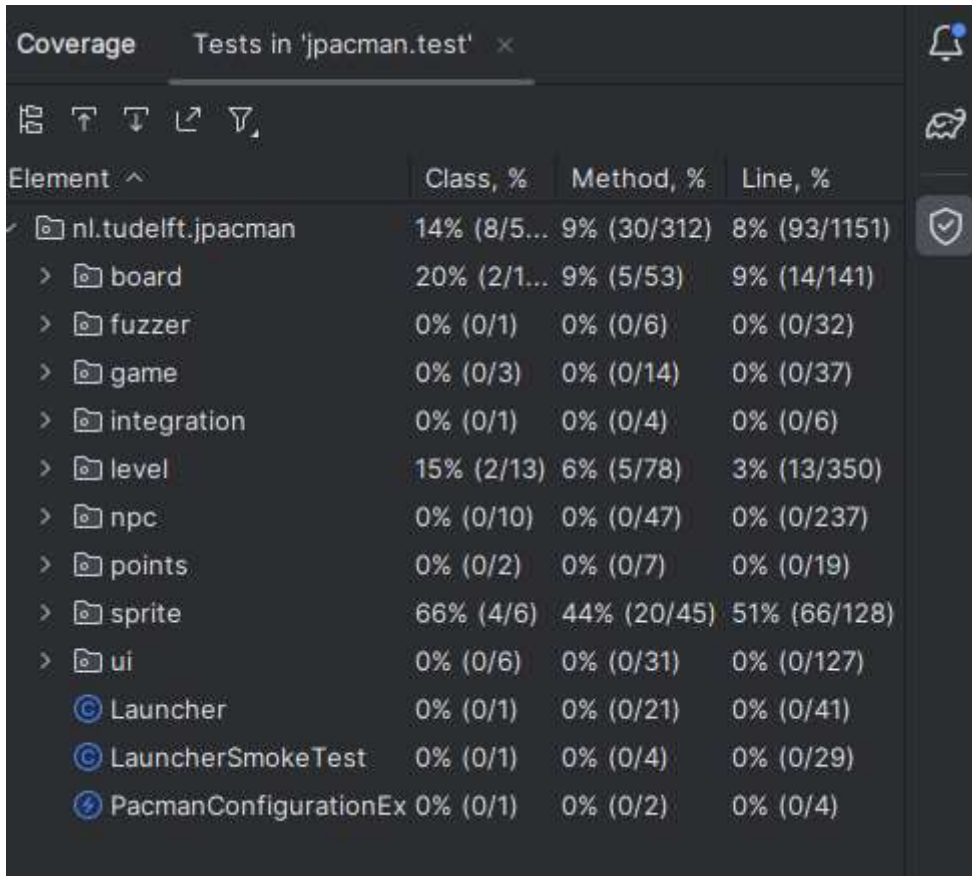


# Unit Testing

## No Unit Tests Provided



The screenshot shows the 'Coverage' window in IntelliJ IDEA for the test 'Tests in 'jpacman.test''. The table displays coverage metrics for various elements in the 'nl.tudelft.jpacman' package. The metrics include Class, Method, and Line coverage percentages and counts.

Element ^	Class, %	Method, %	Line, %
✓ nl.tudelft.jpacman	14% (8/5...)	9% (30/312)	8% (93/1151)
> nl.tudelft.jpacman.board	20% (2/1...)	9% (5/53)	9% (14/141)
> nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
> nl.tudelft.jpacman.game	0% (0/3)	0% (0/14)	0% (0/37)
> nl.tudelft.jpacman.integration	0% (0/1)	0% (0/4)	0% (0/6)
> nl.tudelft.jpacman.level	15% (2/13)	6% (5/78)	3% (13/350)
> nl.tudelft.jpacman.npc	0% (0/10)	0% (0/47)	0% (0/237)
> nl.tudelft.jpacman.points	0% (0/2)	0% (0/7)	0% (0/19)
> nl.tudelft.jpacman.sprite	66% (4/6)	44% (20/45)	51% (66/128)
> nl.tudelft.jpacman.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)
⚡ PacmanConfigurationEx	0% (0/1)	0% (0/2)	0% (0/4)

When no unit tests (excluding PlayerTest and tests which were included in the repo) are provided, we can see in the picture above that our coverage is quite low. Through the inclusion of our 3 unit tests, we will see changes in these percentages. It is important to note that though we will see change, the change will not show significantly through the percentages, as the 3 unit tests only scrape a small percentage of the total methods included in the jpacman repo.

The three methods I decided to implement tests for are:

- src/main/java/nl/tudelft/jpacman/level/Pellet.getValue
- src/main/java/nl/tudelft/jpacman/level/LevelFactory.createGhost
- src/main/java/nl/tudelft/jpacman/board/Square.link

Provided here is the link to my forked repository: <https://github.com/tylereg03/SeniorDesign>

# PelletValueTest

level	15% (2/...	6% (5/78)	3% (13/35...	level	30% (4...	11% (9/78)	7% (25/3...
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/41)	CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/41)
CollisionMap	100% (...)	100% (0/0)	100% (0/0)	CollisionMap	100% (...)	100% (0/0)	100% (0/0)
DefaultPlayerInteractio	0% (0/1)	0% (0/5)	0% (0/13)	DefaultPlayerInteraction	0% (0/1)	0% (0/5)	0% (0/13)
Level	0% (0/2)	0% (0/17)	0% (0/113)	Level	0% (0/2)	0% (0/17)	0% (0/113)
LevelFactory	0% (0/2)	0% (0/7)	0% (0/27)	LevelFactory	50% (1...	28% (2/7)	24% (7/29)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)	LevelTest	0% (0/1)	0% (0/9)	0% (0/30)
MapParser	0% (0/1)	0% (0/10)	0% (0/71)	MapParser	0% (0/1)	0% (0/10)	0% (0/71)
Pellet	0% (0/1)	0% (0/3)	0% (0/5)	Pellet	100% (...)	66% (2/3)	83% (5/6)
Player	100% (1...	25% (2/8)	33% (8/24)	Player	100% (...)	25% (2/8)	33% (8/24)
PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)	PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)
PlayerFactory	100% (1...	100% (3/3)	100% (5/5)	PlayerFactory	100% (...)	100% (3/3)	100% (5/5)

Before

After

*PelletValueTest* is by far the simplest unit test out of the three. Its tested method's directory is found at ***src/main/java/nl/tudelft/jpacman/level/Pellet.getValue***, and the test's main goal is to assert that when a new Pellet object is created, it's point value is equivalent to the one outlined in the LevelFactory class, which is set to be 10, using *assertEquals*. Below is a snippet of *testPelletValue()*.

```
@Test
void testPelletValue() {
    assertEquals(pellet.getValue(), expectedPelletValue);
}
1
```

With the new test put in place, we can see in the picture below that our test percentages have gone up.

Coverage Tests in 'jpacman.test' x			
Element ^	Class, %	Method, %	Line, %
nl.tudelft.jpacman	20% (11/...	11% (36/31...	9% (109/11...

<sup>1</sup> For the entire logic of PelletValueTest, refer to the Code Blocks for JPacman

# GhostCreationTest

Before	After
<div><div>level</div><div><div>CollisionInteractionMap</div><div>CollisionMap</div><div>DefaultPlayerInteractionMap</div><div>Level</div><div>LevelFactory</div><div>LevelTest</div><div>MapParser</div><div>Pellet</div><div>Player</div><div>PlayerCollisions</div><div>PlayerFactory</div></div><div><div>30% ... 11% (9... 7% (25...</div><div>0% (... 0% (0/9) 0% (0/...</div><div>100%... 100% (... 100% (...</div><div>0% (... 0% (0/5) 0% (0/...</div><div>0% (... 0% (0/... 0% (0/...</div><div>50% ... 28% (... 24% (7...</div><div>0% (... 0% (0/9) 0% (0/...</div><div>0% (... 0% (0/... 0% (0/...</div><div>100%... 66% (... 83% (...</div><div>100%... 25% (... 33% (...</div><div>0% (... 0% (0/7) 0% (0/...</div><div>100%... 100% (... 100% (...</div></div></div>	<div><div>level</div><div><div>CollisionInteractionMap</div><div>CollisionMap</div><div>DefaultPlayerInteractionMap</div><div>Level</div><div>LevelFactory</div><div>LevelTest</div><div>MapParser</div><div>Pellet</div><div>Player</div><div>PlayerCollisions</div><div>PlayerFactory</div></div><div><div>30% ... 12% (1... 9% (3...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 100% ... 100% ...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>50% ... 42% (... 48% (...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 66% (... 83% (...</div><div>100... 25% (... 33% (...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 100% ... 100% ...</div></div></div>

Before After

*GhostCreationTest* is the second unit test that was worked on. Its tested method's directory is found at ***src/main/java/nl/tudelft/jpacman/level/LevelFactory.createGhost***, and the test's main goal is to confirm that the ghosts are created correctly and sequentially (**Blinky**, **Inky**, **Pinky**, **Clyde**). *LevelFactory* has a set order for creating the ghosts, so we will create a new ghost object for each of the ghosts mentioned, and compare the order and the type of our ghost object to the ghosts created in *levelFactory*. Below is a snippet of *testGhostCreation()*.

```
// Blinky Test
ghost = levelFactory.createGhost();
if (ghost.getClass() == blinky.getClass()) {
    sameType = true;
}
assertTrue(sameType);
sameType = false;
```

2

Similar tests are executed for the remaining three ghosts. We can also see that in the picture below, our test percentages have again risen.

Coverage	Tests in 'jpacman.test' x
<div><div>nl.tudelft.jpacman</div><div>30% ... 16% (5... 13% (1...</div></div>	<div><div>30% ... 12% (1... 9% (3...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 100% ... 100% ...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>50% ... 42% (... 48% (...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 66% (... 83% (...</div><div>100... 25% (... 33% (...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 100% ... 100% ...</div></div>

<sup>2</sup> For the entire logic of *GhostCreationTest*, refer to the Code Blocks for JPacman

# LinkTest

Before	After
<div><div>board</div><div><div>Board</div><div>BoardFactory</div><div>BoardFactoryTest</div><div>BoardTest</div><div>Direction</div><div>Square</div><div>SquareTest</div><div>Unit</div></div><div><div>20% ... 9% (5/...</div><div>0% (... 0% (0/7) 0% (0/...</div><div>0% (... 0% (0/6) 0% (0/...</div><div>0% (... 0% (0/3) 0% (0/3)</div><div>100%... 75% (3... 90% (1...</div><div>0% (... 0% (0/8) 0% (0/...</div><div>0% (... 0% (0/4) 0% (0/...</div><div>100%... 20% (... 13% (4...</div></div></div>	<div><div>board</div><div><div>Board</div><div>BoardFactory</div><div>BoardFactoryTest</div><div>BoardTest</div><div>Direction</div><div>Square</div><div>SquareTest</div><div>Unit</div></div><div><div>30% ... 18% (1... 18% (...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 75% (... 90% (...</div><div>100... 62% (... 52% (...</div><div>0% (... 0% (0... 0% (0...</div><div>100... 20% (... 13% (...</div></div></div>

*LinkTest* is the third and the final unit test that was worked on. Its tested method's directory is found at ***src/main/java/nl/tudelft/jpacman/board/Square.link***, and the test's main goal is to confirm that the link method not only links two given square objects together (named s1 and s2 in our case), but that it also links them in the correct direction specified. Square has a handy method named `getSquareAt` which is useful in confirming that the link was successful, as `getSquareAt` returns a square adjacent to the current square. Below is a snippet of *testLink()*.

```
// East Test
s1.link(s2, Direction.EAST);
assertEquals(s1.getSquareAt(Direction.EAST), s2);
```

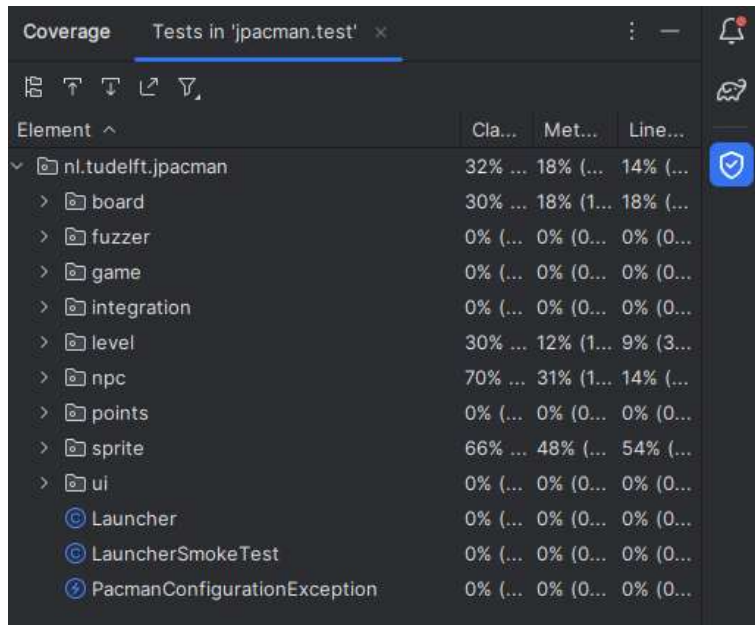
3

Similar tests are executed for the remaining three directions. We can also see that in the picture below, our test percentages have risen.

Coverage	Tests in 'jpacman.test' x	
<div><div>nl.tudelft.jpacman</div><div><div>Board</div><div>BoardFactory</div><div>BoardFactoryTest</div><div>BoardTest</div><div>Direction</div><div>Square</div><div>SquareTest</div><div>Unit</div></div><div><div>32% ... 18% (... 14% (...</div></div></div>		

<sup>3</sup> For the entire logic of LinkTest, refer to the Code Blocks for JPacman

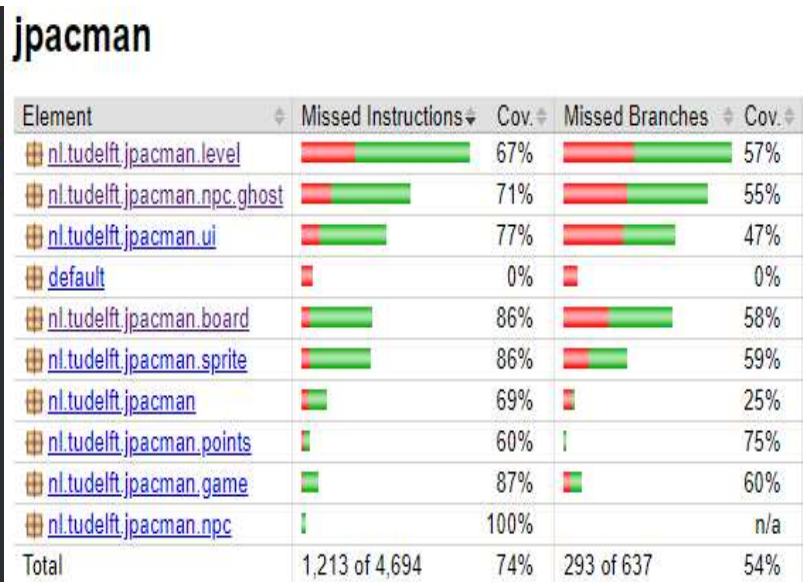
# JaCoCo



The screenshot shows the 'Coverage' tool window in IntelliJ IDEA, displaying test results for 'jpacman.test'. The window is divided into a tree view on the left and a table on the right. The tree view shows the project structure with folders for 'board', 'fuzzer', 'game', 'integration', 'level', 'npc', 'points', 'sprite', and 'ui'. The table on the right shows the coverage for each element, including 'Cla...', 'Met...', and 'Line...'.

Element	Cla...	Met...	Line...
nl.tudelft.jpacman	32% ...	18% (...)	14% (...)
board	30% ...	18% (1...	18% (...)
fuzzer	0% (...)	0% (0...	0% (0...
game	0% (...)	0% (0...	0% (0...
integration	0% (...)	0% (0...	0% (0...
level	30% ...	12% (1...	9% (3...
npc	70% ...	31% (1...	14% (...)
points	0% (...)	0% (0...	0% (0...
sprite	66% ...	48% (...)	54% (...)
ui	0% (...)	0% (0...	0% (0...
Launcher	0% (...)	0% (0...	0% (0...
LauncherSmokeTest	0% (...)	0% (0...	0% (0...
PacmanConfigurationException	0% (...)	0% (0...	0% (0...

Gradle



The screenshot shows the JaCoCo coverage report for 'jpacman'. It displays a table with columns for 'Element', 'Missed Instructions', 'Cov.', 'Missed Branches', and 'Cov.'. The table lists various elements and their coverage percentages, along with visual progress bars.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
nl.tudelft.jpacman.level		67%		57%
nl.tudelft.jpacman.npc.ghost		71%		55%
nl.tudelft.jpacman.ui		77%		47%
default		0%		0%
nl.tudelft.jpacman.board		86%		58%
nl.tudelft.jpacman.sprite		86%		59%
nl.tudelft.jpacman		69%		25%
nl.tudelft.jpacman.points		60%		75%
nl.tudelft.jpacman.game		87%		60%
nl.tudelft.jpacman.npc		100%		n/a
Total	1,213 of 4,694	74%	293 of 637	54%

JaCoCo

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

Despite the fact that both of these reports are generated from the exact same test cases, the coverage results between *JaCoCo* and *IntelliJ* are not similar. This can be attributed to the fact that *IntelliJ* measures if tests cover certain classes, methods, and lines, while *JaCoCo* generates its coverage based on a ratio of total tested lines on any given element/branch, compared with untested lines on the same element/branch (given that the lines are testable). That being said, though these coverage reports do not look similar at face value, these are both correct in their evaluation of test coverage.



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

I found the source code visualization from *JaCoCo* on uncovered branches extraordinarily helpful. This is because when writing a test for any given method, it is almost impossible to anticipate how many lines your assertions will cover simply based on intuition alone. Though experience would certainly make it easier to determine which assertions will test which lines, *JaCoCo*'s source code visualizations highlight exactly which lines were not tested by the tests conducted. This made it easy to see the blind spots in my test cases, and made my overall coverage higher.

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

It is difficult to fully determine which visualization I preferred, as both had their strengths and their drawbacks. That being said, I preferred to look at *JaCoCo*'s report. I believe that *JaCoCo*'s report was more visually understandable, as it would not only walk you through how many lines you missed, but also would give you a direct visual of which lines you missed. *IntelliJ* was in my opinion less intuitive visually, however, it was convenient in its immediate access within the IDE we were programming the tests in, allowing me to test any given method multiple times with ease. Overall, *JaCoCo* had a more visually appealing, telling, complete report for testers to examine, making it the preferable choice for me.

# Code Blocks for JPacman

## PelletValueTest

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

import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.points.DefaultPointCalculator;
import nl.tudelft.jpacman.points.PointCalculator;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;

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

public class PelletValueTest {

    static int expectedPelletValue = 10;
    PacManSprites sprite = new PacManSprites();
    GhostFactory ghostFactory = new GhostFactory(sprite);
    PointCalculator pointCalculator = new DefaultPointCalculator();
    LevelFactory levelFactory = new LevelFactory(sprite, ghostFactory, pointCalculator);
    Pellet pellet = levelFactory.createPellet();

    /**
     * Pellet's getValue method simply returns the value of the pellet.
     * The main difficulty of setting up this test is simply instantiating all the objects
     * necessary in order to properly make a pellet object.
     * According to LevelFactory, the desired pellet value should be 10, and thus, we
defined
     * a variable named "expectedPelletValue" in order to confirm that the newly made pellet
is
     * indeed worth the correct amount of points.
     */
    @Test
    void testPelletValue() {
        assertEquals(pellet.getValue(), expectedPelletValue);
    }
}
```

# GhostCreationTest

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

import nl.tudelft.jpacman.npc.Ghost;
import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.points.DefaultPointCalculator;
import nl.tudelft.jpacman.points.PointCalculator;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;

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

public class GhostCreationTest {
    PacManSprites sprite = new PacManSprites();
    GhostFactory ghostFactory = new GhostFactory(sprite);
    Ghost blinky = ghostFactory.createBlinky();
    Ghost inky = ghostFactory.createInky();
    Ghost pinky = ghostFactory.createPinky();
    Ghost clyde = ghostFactory.createClyde();

    PointCalculator pointCalculator = new DefaultPointCalculator();
    LevelFactory levelFactory = new LevelFactory(sprite, ghostFactory,
pointCalculator);

    /**
     * LevelFactory's createGhost method creates the ghosts in the order of
     * 1. Blinky
     * 2. Inky
     * 3. Pinky
     * 4. Clyde
     * We will use createGhost 4 times and ensure that not only were the ghosts
created,
     * but also that they were created in the correct order.
     */
    @Test
    void testGhostCreation() {

        Ghost ghost;
        boolean sameType;

        // Initialize the sameType boolean to false
        sameType = false;

        // Blinky Test
        ghost = levelFactory.createGhost();
```



```

        if (ghost.getClass() == blinky.getClass()) {
            sameType = true;
        }
        assertTrue(sameType);
        sameType = false;

        // Inky Test
        ghost = levelFactory.createGhost();
        if (ghost.getClass() == inky.getClass()) {
            sameType = true;
        }
        assertTrue(sameType);
        sameType = false;

        // Pinky Test
        ghost = levelFactory.createGhost();
        if (ghost.getClass() == pinky.getClass()) {
            sameType = true;
        }
        assertTrue(sameType);
        sameType = false;

        // Clyde Test
        ghost = levelFactory.createGhost();
        if (ghost.getClass() == clyde.getClass()) {
            sameType = true;
        }
        assertTrue(sameType);
    }
}

```

# LinkTest

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

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

public class LinkTest {
    Square s1 = new BasicSquare();
    Square s2 = new BasicSquare();

    /**
     * Square's link method links one square to another, specified by the 2 squares
     desired to be
     * linked together, along with which direction you want the link to occur. We
     will test
     * the link on each direction, going in the order:
     * 1. EAST
     * 2. WEST
     * 3. NORTH
     * 4. SOUTH
     * We will link 4 times, asserting that the two squares, s1 and s2, are indeed
     linked together
     * in the direction provided, using the handy getSquareAt method.
     */
    @Test
    void testLink() {

        // East Test
        s1.link(s2, Direction.EAST);
        assertEquals(s1.getSquareAt(Direction.EAST), s2);

        // West Test
        s1.link(s2, Direction.WEST);
        assertEquals(s1.getSquareAt(Direction.WEST), s2);

        // North Test
        s1.link(s2, Direction.NORTH);
        assertEquals(s1.getSquareAt(Direction.NORTH), s2);

        // South Test
        s1.link(s2, Direction.SOUTH);
        assertEquals(s1.getSquareAt(Direction.SOUTH), s2);
    }
}
```

# Python Test Coverage

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 0.645s

OK

In the picture above, we can see that the test cases provided below provided 100% coverage. I will only provide code for lines **34-35**, **45-48**, **52-54**, and **74-75**, as the other test implementations were provided in the assignment documentation.

```
def test_from_dict(self):
    # 34 - 35
    account = Account()

    # Fill in test data for account
    data = {
        "name": "Foo",
        "email": "Foo@foo.com",
        "phone_number": "555-555-5555",
        "disabled": False,
        "date_joined": datetime.datetime(1900, 1, 1)
    }

    # Create the account with the data provided
    account.from_dict(data)

    # Assert that each of the fields inputted match with the data provided
    self.assertEqual(account.name, "Foo")
    self.assertEqual(account.email, "Foo@foo.com")
    self.assertEqual(account.phone_number, "555-555-5555")
    self.assertFalse(account.disabled)
    self.assertEqual(account.date_joined, datetime.datetime(1900, 1, 1))

def test_update(self):
    # 45 - 48
    account = Account()

    # Initialize the account name and then create the account
    account.name = "Foo"
    account.create()
```

```

# Change the account name to something new and update it
account.name = "Bar"
account.update()

# Assert that the account name is updated
self.assertNotEqual(account.name, "Foo")
self.assertEqual(account.name, "Bar")

account.id = None

# Tests the specific exception raised when the ID is empty
with self.assertRaises(DataValidationError) as context:
    account.update()

# Simply asserts that the correct error message was raised
self.assertIn("Update called with empty ID field", str(context.exception))

def test_delete(self):
    # 52 - 54
    account = Account()

    # Initialize the account name and then create the account
    account.name = "Foo"
    account.create()

    # Delete the account, and then attempt to retrieve the id
    account.delete()
    deleted_account = Account.query.get(account.id)

    # Assert that the account no longer exists
    self.assertIsNone(deleted_account)

def test_find(self):
    # 74 - 75
    account = Account()

    # Initialize the account name and then create the account
    account.name = "Foo"
    account.create()

    # Use the tested method to find the account id
    found_account = Account.find(account.id)

    # Assert that the account exists, and that the name is correct
    self.assertIsNotNone(found_account)
    self.assertEqual(found_account.name, account.name)

```

# TDD

Name	Stmts	Miss	Cover	Missing
src\counter.py	24	0	100%	
src\status.py	6	0	100%	
TOTAL	30	0	100%	

Ran 4 tests in 0.295s

OK

In the picture above, we can see that the test cases provided below provided 100% coverage. We first had to create the `test_update_a_counter(self)` method. The implementation of this method included:

- Creating a counter
- Asserting that it was created properly using the return code **201**
- Checking the counter value baseline (which should be 0)
- Updating the counter (which should make its value one higher, in our case 1)
- Asserting that the counter was updated properly using the return code **200**
- And finally ensuring that the method properly recognizes when a counter is attempting to be updated when it does not exist using the return code **404**

Once this is run in nosetests, we will enter a **RED** phase.

## test\_update\_a\_counter(self)

```
def test_update_a_counter(self):
    # Here we create a counter
    c_result = self.client.post('/counters/update_counter')
    self.assertEqual(c_result.status_code, status.HTTP_201_CREATED)

    # Here we get the counter
    b_result = self.client.get('/counters/update_counter')
    b_value = b_result.json['update_counter']

    u_result = self.client.put('/counters/update_counter')
    self.assertEqual(u_result.status_code, status.HTTP_200_OK)

    u_result = self.client.get('/counters/update_counter')
    u_value = u_result.json['update_counter']
    self.assertEqual(u_value, b_value + 1)

    # Check status code 404 with uncreated counter
    result = self.client.put('/counters/apple')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

We next had to create the method `update_counter(name)` in the file `counter.py`. The implementation of this method included:

- Create a new route implementing “put”
- Create the function to implement the route
- Increment the counter by `1`
- Return the counter along with the return code `200`

Once this is run in nosetests, we will enter a **GREEN** phase.

## `update_counter(name)`

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS

    if name not in COUNTERS:
        return {"Message": f"Counter {name} not found"}, status.HTTP_404_NOT_FOUND

    # Increment the counter by 1
    COUNTERS[name] += 1

    return {name: COUNTERS[name]}, status.HTTP_200_OK
```



Next, the method `test_read_a_counter(self)` is created. The implementation of this method includes:

- Creating a counter
- Asserting that it was created properly using the return code **201**
- Checking the counter value baseline (which should be 0)
- Asserting that the counter value is correct using the return code **200**
- And finally ensuring that the method properly recognizes when a counter is attempting to be read when it does not exist using the return code **404**

Once this is run in nosetests, we will enter a **RED** phase. Also, this method is extremely similar to `test_update_a_counter(self)`, making this a **REFACTOR** phase as well.

## `test_read_a_counter(self)`

```
def test_read_a_counter(self):
    # Here we create a counter
    c_result = self.client.post('/counters/read_counter')
    self.assertEqual(c_result.status_code, status.HTTP_201_CREATED)

    # Here we read the counter
    result = self.client.get('/counters/read_counter')

    # Check status code 200
    self.assertEqual(result.status_code, status.HTTP_200_OK)

    # Assert that counter value is correct
    expected_value = 0
    actual_value = result.json['read_counter']
    self.assertEqual(actual_value, expected_value)

    # Check status code 404 with uncreated counter
    result = self.client.get('/counters/apple')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

Lastly, we had to make the method *get\_counter(name)*. The implementation of this method includes:

- Create a new route implementing “get”
- Create the function to implement the route
- Return the counter along with the return code **200**

Once this is run in nosetests, we will enter a **GREEN** phase. Also, this method is extremely similar to *update\_counter(name)*, making this a **REFACTOR** phase as well.

## get\_counter(name)

```
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS

    if name not in COUNTERS:
        return {"Message": f"Counter {name} not found"}, status.HTTP_404_NOT_FOUND

    return {name: COUNTERS[name]}, status.HTTP_200_OK
```