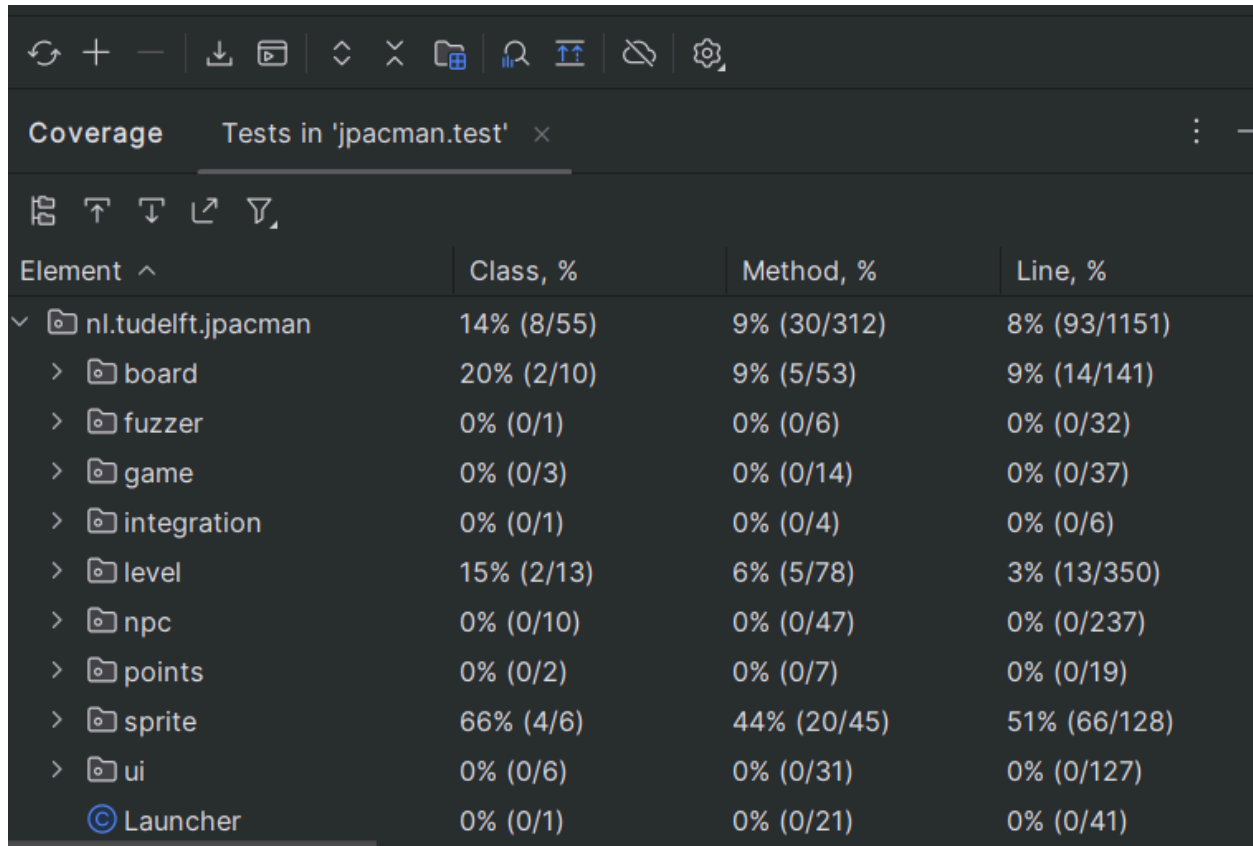


When first taking a look at the Coverage provided by the base code with the `IsAlive()` Test provided, I saw that for the whole repository, it had a Class % of 14%, a Method % of 9%, and a Line % of 8% (Percentages of total coverage).



The screenshot shows a coverage tool interface with a toolbar at the top and a table of results below. The table has four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The 'Element' column lists various packages and classes, including 'nl.tudelft.jpacman', 'board', 'fuzzer', 'game', 'integration', 'level', 'npc', 'points', 'sprite', 'ui', and 'Launcher'. The 'Class, %' column shows coverage percentages and counts (e.g., '14% (8/55)'). The 'Method, %' column shows coverage percentages and counts (e.g., '9% (30/312)'). The 'Line, %' column shows coverage percentages and counts (e.g., '8% (93/1151)').

Element ^	Class, %	Method, %	Line, %
✓  nl.tudelft.jpacman	14% (8/55)	9% (30/312)	8% (93/1151)
>  board	20% (2/10)	9% (5/53)	9% (14/141)
>  fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
>  game	0% (0/3)	0% (0/14)	0% (0/37)
>  integration	0% (0/1)	0% (0/4)	0% (0/6)
>  level	15% (2/13)	6% (5/78)	3% (13/350)
>  npc	0% (0/10)	0% (0/47)	0% (0/237)
>  points	0% (0/2)	0% (0/7)	0% (0/19)
>  sprite	66% (4/6)	44% (20/45)	51% (66/128)
>  ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)

I first implemented a test that Checked the use of the method `getKiller()`, however it was a bit more loaded than I thought, due to the fact that I had to create a player and a ghost, for this example I chose Pinky, and I had to have them collide which required me to get the `PlayerCollision` class imported. This significantly raised some of the percentages which can be seen here. Included is a code snippet as well.

```
public class PlayerTestGetKiller{
    /**
     * I prefer to save the instances for this test in particular
     * because it is really a pain to instantiate Player, and I
     * i will want to test other methods of Player here.
     */
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    private PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
    private Player ThePlayer = Factory.createPacMan();

    GhostFactory Ghostfactory = new GhostFactory(SPRITE_STORE);

    Ghost Pinky = Ghostfactory.createPinky();
    private DefaultPointCalculator point = new DefaultPointCalculator();
    PlayerCollisions playerCollisions = new PlayerCollisions(point);

    @Test
```

```

void testGetKiller(){
    // Lets double check to ensure the Player is alive
    assertThat(ThePlayer.isAlive()).isEqualTo(true);
    ThePlayer.addPoints(1);
    playerCollisions.collide(ThePlayer, Pinky);
    // Now lets Check they have died
    assertThat(ThePlayer.isAlive()).isNotEqualTo(true);
    // Now lets ensure that their killer was pinky

    assertThat(ThePlayer.getKiller().getSprite()).isEqualTo(Pinky.getSprite());

}
}

```

Coverage Tests in 'jpacman.test' ×			
Element ^	Class, %	Method, %	Line, %
✓  nl.tudelft.jpacman	25% (14/55)	15% (49/312)	12% (144/1166)
>  board	20% (2/10)	11% (6/53)	10% (15/141)
>  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	23% (3/13)	16% (13/78)	10% (37/358)
>  npc	40% (4/10)	14% (7/47)	7% (18/243)
>  points	50% (1/2)	14% (1/7)	5% (1/20)
>  sprite	66% (4/6)	48% (22/45)	57% (73/128)
>  ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)

Next was a test Called SetPlayerAlive, which was to test the SetAlive method. This one had a minimal impact and only increased for the line percentage.

```

public class PlayerTestSetAlive{
    /*
     * I prefer to save the instances for this test in particular
     * because it is really a pain to instantiate Player, and I
     * will want to test other methods of Player in here.
     */
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
}

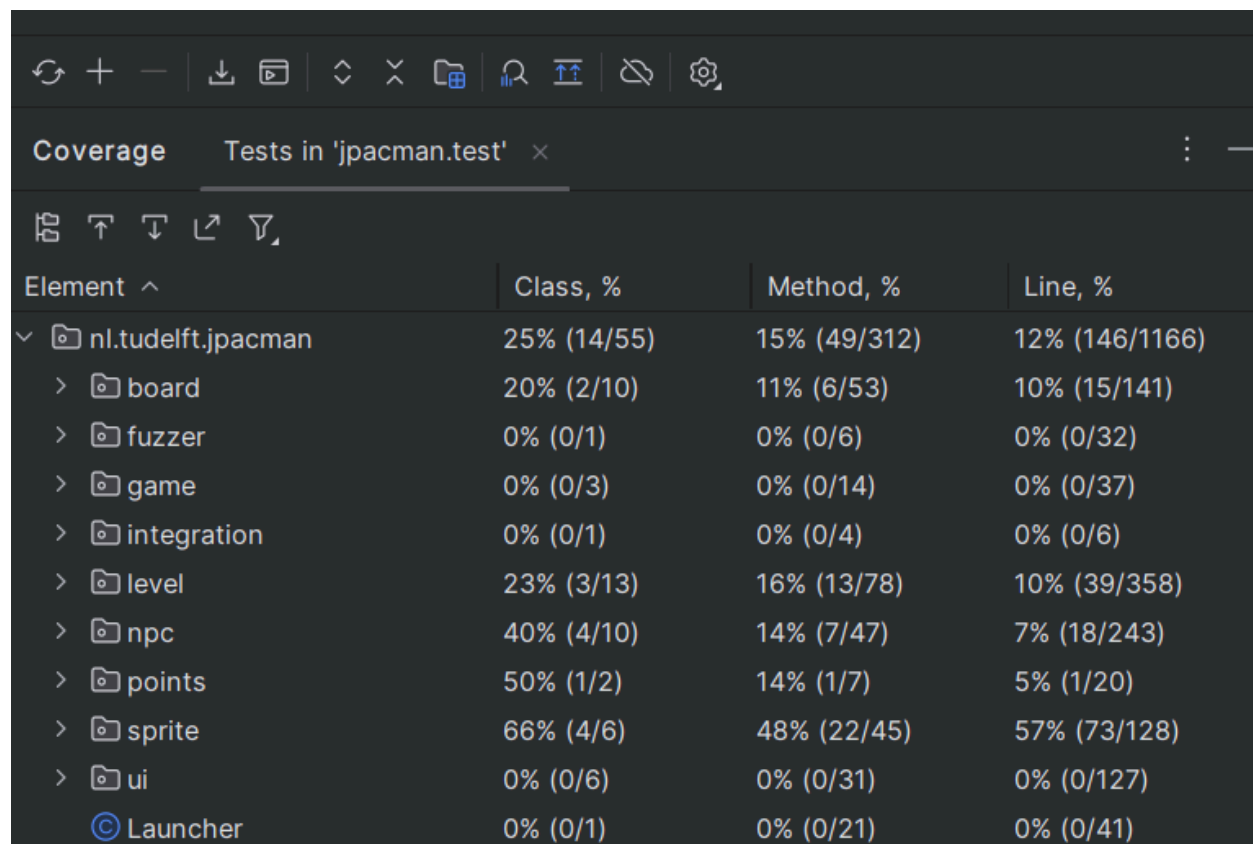
```

```

private PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
private Player ThePlayer = Factory.createPacMan();

@Test
void testDeath() {
    ThePlayer.setAlive(false);
    assertThat(ThePlayer.isAlive()).isEqualTo(false);
    assertThat(ThePlayer.isAlive()).isNotEqualTo(true);
    ThePlayer.setAlive(true);
    assertThat(ThePlayer.isAlive()).isEqualTo(true);
}
}

```



The screenshot shows the Coverage tool in IntelliJ IDEA, displaying test results for 'jpacman.test'. The table below summarizes the coverage data for various elements.

Element ^	Class, %	Method, %	Line, %
✓ nl.tudelft.jpacman	25% (14/55)	15% (49/312)	12% (146/1166)
> board	20% (2/10)	11% (6/53)	10% (15/141)
> 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	23% (3/13)	16% (13/78)	10% (39/358)
> npc	40% (4/10)	14% (7/47)	7% (18/243)
> points	50% (1/2)	14% (1/7)	5% (1/20)
> sprite	66% (4/6)	48% (22/45)	57% (73/128)
> ui	0% (0/6)	0% (0/31)	0% (0/127)
© Launcher	0% (0/1)	0% (0/21)	0% (0/41)

Lastly included is the AddPointTest where I added points to the player's score using the addPoints method then I used the getScore method to ensure the player only had 1 point. This allowed us to check off another method.

```

public class AddPointsTest {
    @Test
    void checkPoints() {
        PacManSprites SPRITE_STORE = new PacManSprites();
        PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
        Player ThePlayer = Factory.createPacMan();

        ThePlayer.addPoints(1);
    }
}

```

```
assertThat(ThePlayer.getScore()).isEqualTo(1);
}
```

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	25% (14/55)	16% (50/312)	12% (147/1166)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
ui	0% (0/6)	0% (0/31)	0% (0/127)
sprite	66% (4/6)	48% (22/45)	57% (73/128)
points	50% (1/2)	14% (1/7)	5% (1/20)
npc	40% (4/10)	14% (7/47)	7% (18/243)
level	23% (3/13)	17% (14/78)	11% (40/358)
integration	0% (0/1)	0% (0/4)	0% (0/6)
game	0% (0/3)	0% (0/14)	0% (0/37)

The coverage results from JaCoCo are sorta similar to the ones in IntelliJ, however, they give much more positive results than IntelliJ. However, it helps a lot more when attempting to get information about branch coverage, this is because when I went to find out why `getSprite()` in `level/player` had a 50% branch coverage, I found out there was a test that covered what `sprite` to get if the player had died, and because we never tested that, we never got to test the death `sprite`. It was incredibly helpful to be able to see uncovered branches with JaCoCo so that in a later test, I can make sure those branches are covered. I prefer JaCoCo Visually because it allows me to check off many more options and for me personally, it's easier to follow and understand in comparison to IntelliJ.

Moving on to the `Test_Coverage` portion of the lab, in the `account.py` script, I had to create tests for the methods, `find`, `delete`, `update`, `from_dict`, and then I had to add a test to check if an account cannot be found when doing an `update`. These tests were very small and allowed coverage for the full program with 100% coverage. Provided below are the test code snippets below. The only issue that I ran into was when statements are involved because both branches must be tested in order to reach 100% coverage, so I had to pass an account without an account ID so that branch could be tested.

```

def test_from_dict(self):
    """ Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account()

    # Call the from_dict method to set attributes from data
    account.from_dict(data)

    # Check if attributes are set correctly
    self.assertEqual(account.name, data["name"])
    self.assertEqual(account.email, data["email"])
    self.assertEqual(account.phone_number, data["phone_number"])
    self.assertEqual(account.disabled, data["disabled"])

def test_update_find_account(self):
    """Test Update/Find account"""
    # Lets create an account
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()

    # now lets go ahead and change name to something that wouldnt be in the data base
    newName = "TestingName"
    account.name = newName

    # lets update the account now
    account.update()

    # now lets ensure that the name is updated using the find
    newaccount = Account.find(account.id)

    self.assertEqual(newaccount.name, newName)

def test_id_not_found(self):
    account = Account(name="Jordan", email="scherf@unlv.nevada.edu")
    with self.assertRaises(DataValidationError) as text:
        account.update()

    self.assertEqual(str(text.exception), "Update called with empty ID field")

def test_delete_account(self):
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    # Accounts been created, lets delete it
    account.delete()
    self.assertEqual(str(Account.find(account.id)), "None")

```

Following Task 5, with the introduction to Task Driven Development, each phase had a cycle of Red, Green, Refactor, in which we start at the red phase creating a test case for something that we have not implemented yet, Green in which we create the function/method in the code for the test case to work properly, then refactor the code as needed, which we will loop back into the red phase.

After completing the step by step tutorial, I entered the red phase when creating the test "test\_update\_a\_counter". From here I had to first create a counter, then send it with a put method to a function that would increase its number by 1 then I would check to ensure that's what it did. This is the red phase (Keep in mind, the function to actually update the counter was not created yet.)

```
def test_update_a_counter(self):
    "It Should return an updated value"
    result = self.client.post('/counters/update')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    updatedResult = self.client.put('/counters/update')
    # I have no idea how to get the value out except with json
    self.assertEqual(updatedResult.json, {'update': 1})
```

Then after, I created the function in counter.py which used the PUT method to update it and first it checked if the name was in COUNTERS, if it was it would update. This is the green phase.

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    if name in COUNTERS:
        COUNTERS[name] = COUNTERS[name] + 1
        return {name: COUNTERS[name]}, status.HTTP_200_OK
```

But what if it doesn't exist? if not, it would return the appropriate status code. This is the refactor stage (blue) because even if it passed the minimum test that we gave it, we should attempt to improve it.

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    if name in COUNTERS:
        COUNTERS[name] = COUNTERS[name] + 1
        return {name: COUNTERS[name]}, status.HTTP_200_OK
    return {"Message":f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
```

This process remained the same for the last task, read\_counter.

First I created a test case for the function, test\_update\_a\_counter (Red Phase)

```
def test_read_counter(self):
    "it should return the actual value in the counter"
    result = self.client.post('/counters/test')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    # Lets increment the counter Twice
    self.client.put('/counters/test')
    self.client.put('/counters/test')
    # now lets check that the value of the counter should be 2
    testNumber = self.client.get('/counters/test').text
    self.assertEqual(int(testNumber), 2)
```

Then created the function that would allow it to pass at the minimum level (Green Phase)

```
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    if name in COUNTERS:
        return str(COUNTERS[name]), status.HTTP_200_OK
```

Then refactored both the test and the function to have more coverage on all possible branches (Blue/Refactoring Phase)

```
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    if name in COUNTERS:
        return str(COUNTERS[name]), status.HTTP_200_OK
    return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
```

```
def test_read_counter(self):
    "it should return the actual value in the counter"
    result = self.client.post('/counters/test')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    # Lets increment the counter Twice
    self.client.put('/counters/test')
    self.client.put('/counters/test')
    # now lets check that the value of the counter should be 2
    testNumber = self.client.get('/counters/test').text
    self.assertEqual(int(testNumber), 2)
    # now lets try to get a counter that doesnt exist
    doesntExist = self.client.get('/counter/Null')
    self.assertEqual(doesntExist.status_code, status.HTTP_404_NOT_FOUND)
```

This allowed 100% coverage.

Counter tests

- It should create a counter
- It should return an error for duplicates
- it should return the actual value in the counter
- It Should return an updated value

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 0.294s

OK

A link to my forked repository is below

<https://github.com/Jordan-Scherf/SeniorDesign-Jordan>