

Task 1 – JPacman Test Coverage

Is the coverage good enough?

No, the coverage is very low out of all the classes there is 3.6% coverage and there is even less coverage when it comes to line coverage with an 1.2%.

Task 2 – Increasing Coverage on JPacman

Figure 1: Unit Test for `src/main/java/nl/tudelft/jpacman/level/Pellet.getValue` and `Pellet.getSprite`

```
1 package nl.tudelft.jpacman.level;
2
3 import nl.tudelft.jpacman.sprite.PacManSprites;
4 import org.junit.jupiter.api.Test;
5
6 import static org.assertj.core.api.Assertions.assertThat;
7
8
9 new *
10 public class PelletTest {
11     2 usages
12     private final static PacManSprites testPacManSprite = new PacManSprites();
13     2 usages
14     private final Pellet testPellet = new Pellet( points: 12, testPacManSprite.getPelletSprite());
15
16     new *
17     @Test
18     void testGetValue() { assertThat(testPellet.getValue()).isEqualTo( expected: 12); }
19
20     new *
21     @Test
22     void testGetSprite() { assertThat(testPellet.getSprite()).isEqualTo(testPacManSprite.getPelletSprite()); }
23 }
```

**Figure 2: Unit Test for
src/main/java/nl/tudelft/jpacman/npc/ghost/GhostFactory.createBlinky**

```

1  package nl.tudelft.jpacman.npc.ghost;
2
3  import nl.tudelft.jpacman.sprite.PacManSprites;
4  import org.junit.jupiter.api.Test;
5
6  import static org.assertj.core.api.Assertions.assertThat;
7
8
9  new *
10 public class GhostFactoryTest {
11
12     new *
13     @Test
14     void createBlinky() {
15         PacManSprites testSpriteStore = new PacManSprites();
16         GhostFactory testGhostFactory = new GhostFactory(testSpriteStore);
17         Blinky testBlinky = new Blinky(testSpriteStore.getGhostSprite(GhostColor.RED));
18         assertThat(testGhostFactory.createBlinky()).assertInstanceOf(testBlinky.getClass());
19     }
20 }

```

Figure 3: Before Adding Unit Tests

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	3.6% (2/55)	1.5% (5/326)	1.2% (14/1161)

Coverage Breakdown


Package 	Class, %	Method, %	Line, %
nl.tudelft.jpacman	0% (0/3)	0% (0/29)	0% (0/77)
nl.tudelft.jpacman.board	20% (2/10)	8.9% (5/56)	9.5% (14/148)
nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/7)	0% (0/33)
nl.tudelft.jpacman.game	0% (0/3)	0% (0/14)	0% (0/37)
nl.tudelft.jpacman.integration	0% (0/1)	0% (0/5)	0% (0/7)
nl.tudelft.jpacman.level	0% (0/13)	0% (0/79)	0% (0/351)
nl.tudelft.jpacman.npc	0% (0/1)	0% (0/4)	0% (0/8)
nl.tudelft.jpacman.npc.ghost	0% (0/9)	0% (0/44)	0% (0/230)
nl.tudelft.jpacman.points	0% (0/2)	0% (0/9)	0% (0/21)
nl.tudelft.jpacman.sprite	0% (0/6)	0% (0/48)	0% (0/122)
nl.tudelft.jpacman.ui	0% (0/6)	0% (0/31)	0% (0/127)

Figure 4: After Adding Unit Tests

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	25.5% (14/55)	13.2% (43/326)	10.3% (122/1182)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
nl.tudelft.jpacman	0% (0/3)	0% (0/29)	0% (0/77)
nl.tudelft.jpacman.board	20% (2/10)	8.9% (5/56)	9.5% (14/148)
nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/7)	0% (0/33)
nl.tudelft.jpacman.game	0% (0/3)	0% (0/14)	0% (0/37)
nl.tudelft.jpacman.integration	0% (0/1)	0% (0/5)	0% (0/7)
nl.tudelft.jpacman.level	23.1% (3/13)	10.1% (8/79)	5.3% (19/357)
nl.tudelft.jpacman.npc	100% (1/1)	25% (1/4)	62.5% (5/8)
nl.tudelft.jpacman.npc.ghost	33.3% (3/9)	11.4% (5/44)	5.1% (12/236)
nl.tudelft.jpacman.points	0% (0/2)	0% (0/9)	0% (0/21)
nl.tudelft.jpacman.sprite	83.3% (5/6)	50% (24/48)	55% (72/131)
nl.tudelft.jpacman.ui	0% (0/6)	0% (0/31)	0% (0/127)

Task 3 – JaCoCo Report on JPacman

Figure 5: JaCoCo Report

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 from JaCoCo are different from the ones retrieved from IntelliJ. This difference is mainly due to what each tool focuses on in their report. JaCoCo focuses on missed instructions and branches while IntelliJ focuses on the percentages covered for the classes, methods, and lines.

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

JaCoCo's uncovered branches visualization is extremely helpful as it gives you a very detailed view of what branches you missed. It even colored the lines where the branches missed coverage.

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

I prefer JaCoCo's report for big projects that need a lot of testing as it has all the tools to aid developers in detecting missed coverage in instruction and branches. I would use IntelliJ's coverage window for small hobby projects that don't really need much testing. When it comes to JPacman I prefer JaCoCo's as it gives a more in depth report of what coverage is missed by the unit tests.

Task 4 – Working with Python Test Coverage

Figure 6: Python Nosetests Coverage

```
→ test_coverage git:(main) ✕ nosetests

Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data
- Test account deletion
- Test account from dict
- repr
- Test account to dict
- Test updates an Account in the database
- Test updates an Account without an id

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 0.234s

OK
```

Figure 7: Python Test Code Snippets

```
def test_from_dict(self):
    """ Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account()
    account.from_dict(data)
    for key in data:
        self.assertEqual(getattr(account, key), data[key])

def test_update(self):
    """ Test updates an Account in the database """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    account.name = "Test update"
    account.update()
    updatedAccount = Account.find(account.id)
    self.assertEqual(updatedAccount.name, "Test update")

def test_update_without_id(self):
    """ Test updates an Account without an id """
    account = Account(name="name", email="email")
    with self.assertRaises(DataValidationError):
        account.update()

def test_delete(self):
    """ Test account deletion """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    account.delete()
    self.assertEqual(len(Account.all()), 0)
```

Task 5 - TDD

Red Phase

Update A Counter:

To test if a counter has been properly updated, we need to first create that counter then update it. In my test, I want to make sure the status code reflects the item has been created so I am able to update it using a PUT method. After using the PUT method to update the counter, I want to make sure the update was successful by checking the status code. Lastly, I want to make sure that the counter has been updated and thus the baseline value should be less than the new updated value.

Figure 8: Code Snippet for test_update_a_counter

```
def test_update_a_counter(self):
    """It should update a counter"""
    base = self.client.post('/counters/baz')
    self.assertEqual(base.status_code, status.HTTP_201_CREATED)
    result = self.client.put('/counters/baz')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertGreater(result.json['baz'], base.json['baz'])
```

Get A Counter:

To test if a counter can be retrieved, we first need to create that counter, and validate that the status code shows the counter was created. After we want to use the GET method to retrieve that counter using the proper counter id. We want to check that the counter was retrieved successfully by asserting the status code. Finally, we want to check if the value retrieved by the GET method is valid.

Figure 9: Code Snippet for test_get_a_counter

```
def test_get_a_counter(self):
    """It should get a counter"""
    result = self.client.post('/counters/bat')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = self.client.get('/counters/bat')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(result.json['bat'], 0)
```

The nosetests results for both of these red phases give the same 405 status code since this method has not yet been implemented, which is to be expected with TDD.

Figure 10: Nostests Result for Update Counter Red Phase

```
- It should update a counter (FAILED)

=====
FAIL: It should update a counter
-----
Traceback (most recent call last):
  File "/Users/marcos/Desktop/tdd/tests/test_counter.py", line 44, 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 0.069s

FAILED (failures=1)
```

Figure 11: Nostests Result for Get Counter Red Phase

```
- It should get a counter (FAILED)
- It should update a counter

=====
FAIL: It should get a counter
-----
Traceback (most recent call last):
  File "/Users/marcos/Desktop/tdd/tests/test_counter.py", line 52, in test_get_a_counter
    self.assertEqual(result.status_code, status.HTTP_200_OK)
AssertionError: 405 != 200
----- >> begin captured logging << -----
src.counter: INFO: Request to create counter: bat
----- >> end captured logging << -----

Name           Stmts   Miss  Cover   Missing
-----
src/counter.py    18      1    94%    29
src/status.py      6      0   100%
-----
TOTAL              24      1    96%
-----
Ran 4 tests in 0.069s

FAILED (failures=1)
```

Green Phase

Update A Counter:

To implement the update counter method I first declared the route above the function name with the PUT method. In order to update a counter it needs to exist in COUNTERS, I check if the name of the counter given exists in the counters dictionary. If it doesn't exist I throw a 404 status code error with a message saying the counter does not exist. If it exists I update the counter by incrementing it by one and return the new value of the counter with a successful status code.

Figure 12: Code Snippet for update_counter

```
@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} does not exist"}, status.HTTP_404_NOT_FOUND
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

Get A Counter:

The implementation of getting a counter is almost the same as updating a counter. I declared the route above the function with the GET method. I check if the counter is in the COUNTERS dictionary, if it isn't I return an appropriate error message with a 404 code. Else I return the retrieved counter from the dictionary with a 200 code.

Figure 13: Code Snippet for get_counter

```
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Get a counter"""
    app.logger.info(f"Request to get counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

After implementing the two functions the nosetests pass without issue. However, there are two lines that don't get covered which are the error handling lines.

Figure 14: Nosetests Results for Green Phase

```
→ tdd git:(main) ✖ nosetests

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

Name                Stmts  Miss  Cover   Missing
-----
src/counter.py       24      2    92%    29, 39
src/status.py         6      0   100%
-----
TOTAL                 30      2    93%
-----

Ran 4 tests in 0.085s

OK
```

Refactor Phase

In order to fix the missing line coverage I added two new tests that made sure the 404 status codes were returned when trying to access items that were not in the COUNTERS dictionary.

Figure 15: Code Snippets of Code Refactor

```
def test_get_a_counter_that_does_not_exist(self):
    """It should return an error for a counter that does not exist"""
    result = self.client.get('/counters/does_not_exist')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

def test_update_a_counter_that_does_not_exist(self):
    """It should return an error for a counter that does not exist"""
    result = self.client.put('/counters/does_not_exist')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

Figure 16: Nosetests Results of Refactor

```
→ tdd git:(main) ✕ nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- It should get a counter
- It should return an error for a counter that does not exist
- It should update a counter
- It should return an error for a counter that does not exist

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 0.086s

OK
```