














Fork Repository Link: <https://github.com/Snapshot0010/Group8/tree/master>

## Task 2.1

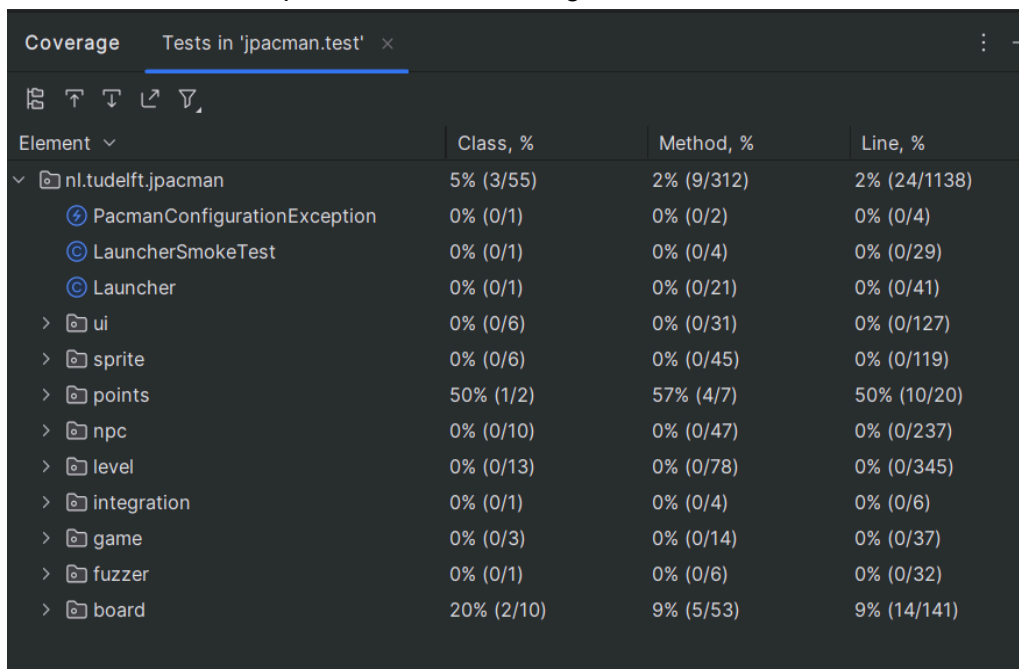
Element ▾	Class, %	Method, %	Line, %
▼  nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
 PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)
 LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
 Launcher	0% (0/1)	0% (0/21)	0% (0/41)
>  ui	0% (0/6)	0% (0/31)	0% (0/127)
>  sprite	0% (0/6)	0% (0/45)	0% (0/119)
>  points	0% (0/2)	0% (0/7)	0% (0/19)
>  npc	0% (0/10)	0% (0/47)	0% (0/237)
>  level	0% (0/13)	0% (0/78)	0% (0/345)
>  integration	0% (0/1)	0% (0/4)	0% (0/6)
>  game	0% (0/3)	0% (0/14)	0% (0/37)
>  fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
>  board	20% (2/10)	9% (5/53)	9% (14/141)

- Above is the test coverage before completing any unit tests. As can be seen, there is only 3% coverage for the classes and 1% coverage for the methods (and lines as well but those matter a bit less than the other 2). As such, many new unit tests need to be added.

## Method 1:

```
1 package nl.tudelft.jpacman.points;
2 import static org.assertj.core.api.Assertions.assertThat;
3 import org.junit.jupiter.api.Test;
4 public class PointCalculatorLoaderTester
5 {
6     PointCalculatorLoader loadTester = new PointCalculatorLoader();
7
8     @Test
9     void testLoad()
10    {
11        assertThat(loadTester.load()).isNotNull();
12    }
13
14
15 }
```

- The first method that I tested was the load method for the PointCalculatorLoader class. The only thing that really need to be checked was that after the load method was called, a null would not be returned as is shown above (image taken from Github repo)
- After this test case implemented the coverage looked like as follows:



The screenshot shows the 'Coverage' window in IntelliJ IDEA, titled 'Tests in 'jpacman.test''. It displays a table of coverage data for various classes and methods. The 'points' package shows 50% class coverage and 57% method coverage, with 10 out of 20 lines covered. The 'board' package shows 20% class coverage and 9% method coverage, with 14 out of 141 lines covered.

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	5% (3/55)	2% (9/312)	2% (24/1138)
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	0% (0/6)	0% (0/45)	0% (0/119)
points	50% (1/2)	57% (4/7)	50% (10/20)
npc	0% (0/10)	0% (0/47)	0% (0/237)
level	0% (0/13)	0% (0/78)	0% (0/345)
integration	0% (0/1)	0% (0/4)	0% (0/6)
game	0% (0/3)	0% (0/14)	0% (0/37)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
board	20% (2/10)	9% (5/53)	9% (14/141)

- It gave a decent boost to the coverage of the points section of the code.

## Method 2:

```
1 package nl.tudelft.jpacman.npc.ghost;
2
3 import static org.assertj.core.api.Assertions.assertThat;
4
5
6 import nl.tudelft.jpacman.sprite.PacManSprites;
7 import org.junit.jupiter.api.Test;
8 public class GhostFactoryTest
9 {
10     private static final PacManSprites testSprite = new PacManSprites();
11     private GhostFactory testFactor = new GhostFactory(testSprite);
12
13     @Test
14     void testCreateInky()
15     {
16         assertThat(testFactor.createClyde()).isNotNull();
17     }
18 }
```

- The next method I decided to test was the createClyde method. Similarly to the load method, I simply needed to make sure that a Clyde object was created and thus the statement could not be null. However, I did need to do an extra set of steps as I first had to make a GhostFactory object, which required the creation of a PacManSprites object.
- After the implementation of this method, the test coverage looked like this:

Element ▾	Class, %	Method, %	Line, %
▾ nl.tudelft.jpacman	20% (11/55)	10% (34/312)	9% (110/1153)
⚡ 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)	42% (19/45)	49% (63/128)
> points	50% (1/2)	57% (4/7)	50% (10/20)
> npc	40% (4/10)	12% (6/47)	9% (23/243)
> level	0% (0/13)	0% (0/78)	0% (0/345)
> integration	0% (0/1)	0% (0/4)	0% (0/6)
> game	0% (0/3)	0% (0/14)	0% (0/37)
> fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
> board	20% (2/10)	9% (5/53)	9% (14/141)

- This test case added a lot of coverage overall (mostly in the sprites and npc sections) of the code

## Method 3:

```
1 package nl.tudelft.jpacman.level;
2 import static org.assertj.core.api.Assertions.assertThat;
3
4 import nl.tudelft.jpacman.npc_ghost.GhostFactory;
5 import nl.tudelft.jpacman.points.PointCalculator;
6 import nl.tudelft.jpacman.points.PointCalculatorLoader;
7 import nl.tudelft.jpacman.sprite.PacManSprites;
8 import org.junit.jupiter.api.Test;
9
10 public class LevelFactoryTester
11 {
12     private PointCalculator testCalc;
13     private static final PacManSprites testSprite = new PacManSprites();
14     private GhostFactory testFactor = new GhostFactory(testSprite);
15     private LevelFactory testLevelFactory = new LevelFactory(testSprite, testFactor, testCalc);
16
17     @Test
18     void testCreateGhost()
19     {
20         assertThat(testLevelFactory.createGhost()).isNotNull();
21     }
22 }
```

- The final method that I decided to test was the createGhost method. This was the most complicated method to test as I needed to create a LevelFactory object to test it, meaning I needed to create a PointCalculator, PacManSprites, and GhostFactory object before initializing the LevelFactory object and making sure that the createGhost method returned something that was not null.
- After implementing this test the test coverage looked like this:

Element ▾	Class, %	Method, %	Line, %
▾ nl.tudelft.jpacman	23% (13/55)	12% (39/312)	10% (124/1155)
⚡ 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)	42% (19/45)	49% (63/128)
> points	50% (1/2)	57% (4/7)	50% (10/20)
> npc	50% (5/10)	19% (9/47)	11% (27/243)
> level	7% (1/13)	2% (2/78)	2% (10/347)
> integration	0% (0/1)	0% (0/4)	0% (0/6)
> game	0% (0/3)	0% (0/14)	0% (0/37)
> fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
> board	20% (2/10)	9% (5/53)	9% (14/141)

- This test case added the least amount of coverage, only adding a little bit of coverage to the level and npc sections of the code. However, there is likely some overlap with the createClyde method so that could be making it look like this test covers less than it actually does.

## Task 3

Questions:

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

### 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

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	23% (13/55)	12% (39/312)	10% (124/1155)
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)	42% (19/45)	49% (63/128)
points	50% (1/2)	57% (4/7)	50% (10/20)
npc	50% (5/10)	19% (9/47)	11% (27/243)
level	7% (1/13)	2% (2/78)	2% (10/347)
integration	0% (0/1)	0% (0/4)	0% (0/6)
game	0% (0/3)	0% (0/14)	0% (0/37)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
board	20% (2/10)	9% (5/53)	9% (14/141)

No, the results are quite different. For one, there are even different amounts of classes, methods, and lines in both. At a first glance, this appears to be because things like the launcher are missing from JaCoCo. However, the percentage covered is much more in JaCoCo, and it also has more parameters it is tracking (such as branches). This is likely because it has different standards from the IntelliJ version and does not use the test cases that we created.

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

Yes I did, it helped show what parts of each source code file had already been covered by test cases, making it clear which specific lines still needed coverage.

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

Personally, I prefer IntelliJ. While the extra parameters tracked are really nice to have, I do personally think that only adds a little bit towards JaCoCo's favor. Otherwise, I like that IntelliJ is all done within the IDE itself so I don't need to keep track of an extra window. In my opinion, that makes the process faster. Additionally, IntelliJ has a similar feature to the source code visualization, however, clicking on the files leads you directly to files themselves so you could make changes to the source code right there if necessary, also making the testing process faster.

## Task 4

```
Name                StmtS  Miss  Cover  Missing
-----
models\__init__.py    7     0  100%
models\account.py    40     4   90%  45-48
-----
TOTAL                 47     4   91%
-----

Ran 7 tests in 4.261s

OK
```

- By the end of task 4, I managed to get the test coverage overall to 91% (I'm only missing the update method)

## Testing from\_dict

Code Snippet for from\_dict:

```
new *
def test_from_dict(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account()
    account.from_dict(data)
    self.assertEqual(account.name, data["name"])
```

- To test from\_dict I first put data from a random account from the file into the data variable then made an empty account. I then called from\_dict with the data as the parameter. Then, I simply needed to double check that the name linked to the account was the same name held in the "name" index of the data dictionary.

## Testing delete

Code Snippet for delete:

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

- To test delete, I first made an account and filled it with randomly generated data from the list of possible accounts. I then immediately called delete to remove it and just had to assert that the length of the list of all accounts was 0 (as no other accounts should have been made in this test case)

## Testing find

Code Snippet for test\_find

```
new +
def test_find(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    self.assertIsNotNone(account.find(account.id))
```

- To test find, I first made an account and filled it with randomly generated data from the list of possible accounts. Then I used create to ensure that the account had been made. Then, I simply need to ensure that I can find the account by its id. As such, I simply needed to assert that the return from find was not null



## Task 5

### My Task

As a small preface, my phases were a little weird as I could not for the longest time figure out how to retrieve data from the counters which interfered with my ability to effectively write the test cases.

### Update Counter

For update, I started off in the red phase with this as my test\_update code:

```
new *
def test_update_a_counter(self):
    result = self.client.post('counters/update')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    baseline = result.data
    result = self.client.put('counters/update')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(result.data, baseline+1)
```

From there, I wrote the update method as such:

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

However, that gave me an error (not a test failure, an error) so I had gone wrong somewhere while writing my update counter. The error was as follows:

```
Traceback (most recent call last):
  File "D:\Lab_Repos\tdd\tests\test_counter.py", line 43, in test_update_a_counter
    self.assertEqual(result.data, baseline+1)
TypeError: can't concat int to bytes
```

I figured the issue was with my getting of the actual data from the counter so I ended up trying many different setups. I can not include them all here in the picture as that would cause the report to become too long, but they all included changing what into the baseline in some way. One attempt of note, however, was when I changed the return statement of update itself to:

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

However, that failed with:

```
File "D:\Lab_Repos\tdd\tests\test_counter.py", line 42, in test_update_a_coun
    self.assertEqual(result.status_code, status.HTTP_200_OK)
AssertionError: 500 != 200
```

Clearly something was not working but I kept trying different combinations for baseline until I landed on

```
def test_update_a_counter(self):
    result = self.client.post('counters/update')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    baseline = result.json['update']
    result = self.client.put('counters/update')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(result.json['update'], baseline+1)
```

Which got me into the green phase and thus ended the development for the update method

## Get

For update I started in the red phase with the following code:

```
def test_read_a_counter(self):
    result = self.client.post('counters/read')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = self.client.get('counters/read')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(result.data, {'second': 0})
```

The get method then started out as follows:

```
@app.route(rule: '/counters/<name>', methods=['GET'])
def get_counter(name):
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

I had moved on from update to try get after failing to do update and thought I might be able to figure it out. As such, the refactor and continuous red phases were somewhat shared between the two methods. However, one additional aspect about get testing came about while trying to refactor. I realized it should return an error code if the id does not exist, so I ended up adding another test case below this one coded as follows;

```
def test_read_empty_counter(self):  
    result = self.client.get('counters/emptyRead')  
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

I then made the following changes to the get method:

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

The test\_read\_empty\_counter worked immediately, which showed I was somewhat on the right track and something was simply going wrong in my retrieval of the data. After figuring out how to retrieve the data, my test\_read\_a\_counter method read as follows:

```
def test_read_a_counter(self):  
    result = self.client.post('counters/read')  
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)  
    result = self.client.get('counters/read')  
    self.assertEqual(result.status_code, status.HTTP_200_OK)  
    self.assertEqual(result.json['read'], second: 0)
```

And got me to the green phase, thus ending the testing process.