Dynamic Analysis - Unit Testing

Executive Report

09 SEPTEMBER 2024 / CS472 - 1001 / Group 2 / Justin Delos Reyes

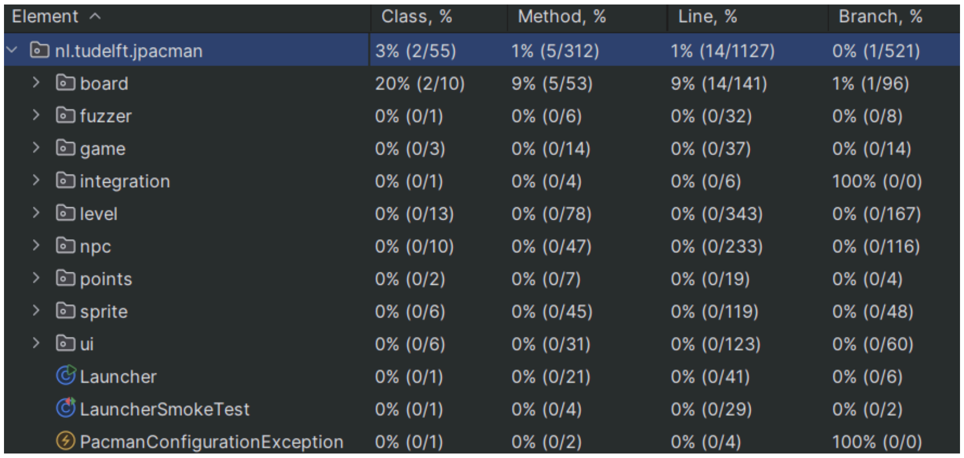
AGENDA

In this lab we aimed to increase the unit test coverage of the JPacman repository through implementing unit tests of our own.

Personal Repository : https://github.com/justinrreyes/CS472_LABS

Group Repository : https://github.com/uid106/CS472_LABS

Initial Coverage

1. Below is a screenshot of the coverage analysis of the Jpacman repository as given by the IntelliJ IDE Coverage tool.
    a. 

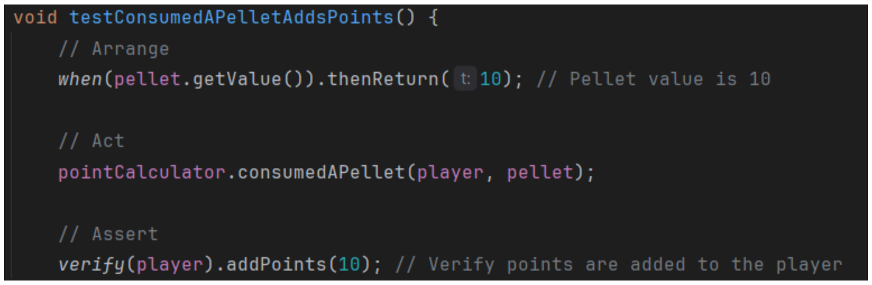
Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1127)	0% (1/521)
board	20% (2/10)	9% (5/53)	9% (14/141)	1% (1/96)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
level	0% (0/13)	0% (0/78)	0% (0/343)	0% (0/167)
npc	0% (0/10)	0% (0/47)	0% (0/233)	0% (0/116)
points	0% (0/2)	0% (0/7)	0% (0/19)	0% (0/4)
sprite	0% (0/6)	0% (0/45)	0% (0/119)	0% (0/48)
ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)	0% (0/6)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)	0% (0/2)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)

2. As you can see the initial starting point consists of nearly all main packages having no coverage at all with 0%.

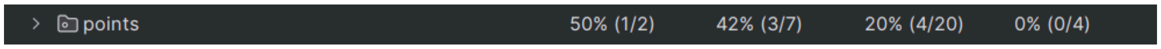
Increasing Coverage

- In order to increase coverage I developed unit tests for three different classes. These classes are:
    - DefaultPointCalculator
    - GhostFactory
    - Game

DefaultPointCalculator Test

- The DefaultPointCalculator tests consisted of checking that points were only added when a pellet was consumed and NOT when moving or colliding with a ghost.
    - 

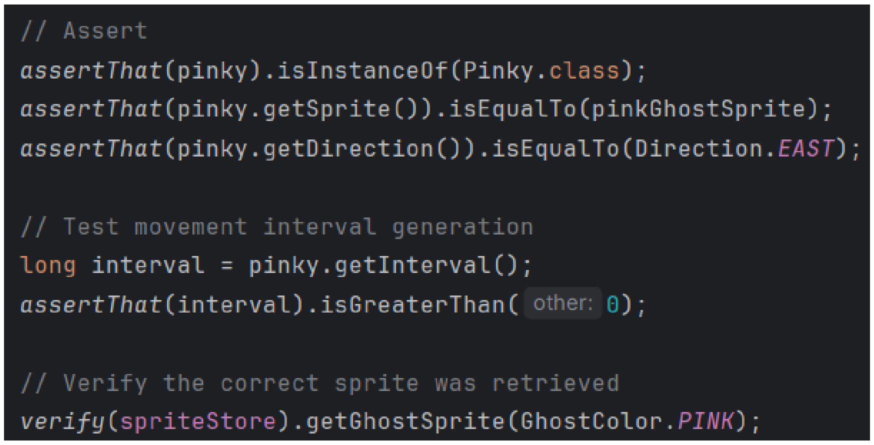
```java
void testConsumedAPelletAddsPoints() {
    // Arrange
    when(pellet.getValue()).thenReturn( t: 10); // Pellet value is 10

    // Act
    pointCalculator.consumedAPellet(player, pellet);

    // Assert
    verify(player).addPoints(10); // Verify points are added to the player
```

- The coverage analysis after implementation of this test is shown below.
    - 

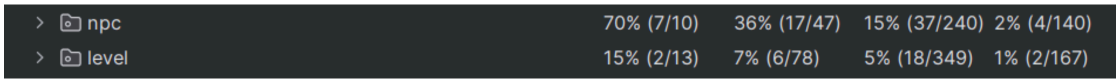
```
>  points                                50% (1/2)    42% (3/7)    20% (4/20)    0% (0/4)
```

    - We see an increase of 50%, 42%, and 20% coverage for the categories.

GhostFactory Test

- The GhostFactory tests involved ensuring that the proper ghost was instantiated and checking its initial move direction and interval were set properly. This was done for all four ghosts.
    - 

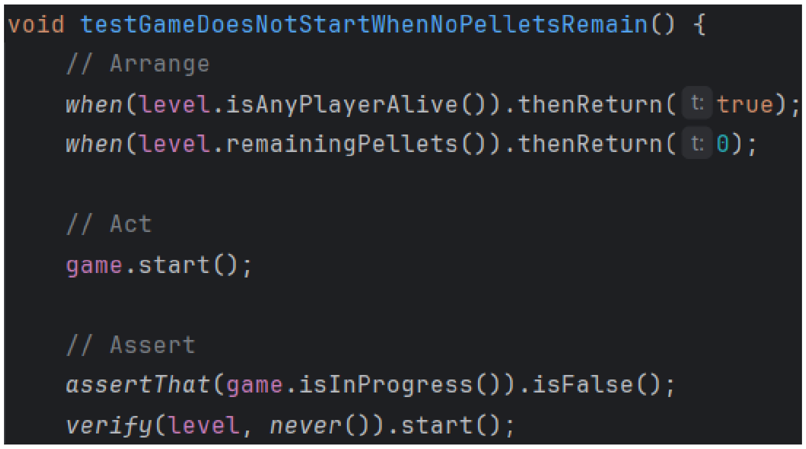
```java
// Assert
assertThat(pinky).isInstanceOf(Pinky.class);
assertThat(pinky.getSprite()).isEqualTo(pinkGhostSprite);
assertThat(pinky.getDirection()).isEqualTo(Direction.EAST);

// Test movement interval generation
long interval = pinky.getInterval();
assertThat(interval).isGreaterThan( other: 0);

// Verify the correct sprite was retrieved
verify(spriteStore).getGhostSprite(GhostColor.PINK);
```

- The coverage analysis after implementation of this test is shown below.
    - 

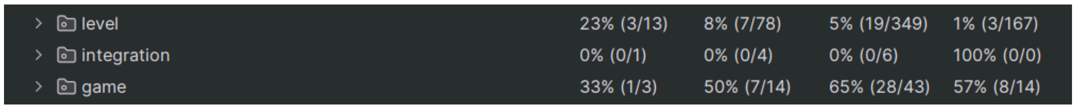
```
>  npc                                    70% (7/10)    36% (17/47)    15% (37/240)    2% (4/140)
>  level                                  15% (2/13)    7% (6/78)      5% (18/349)     1% (2/167)
```

    - We see a massive increase in npc coverage and slight increase in level coverage as well. These categories were all at 0% initially.

Game Test

- The Game test ensured that the game had been created properly and was progressing as intended. This includes checking if the player is alive, checking if there are pellets left to be collected, and that the player is able to move.
    - 

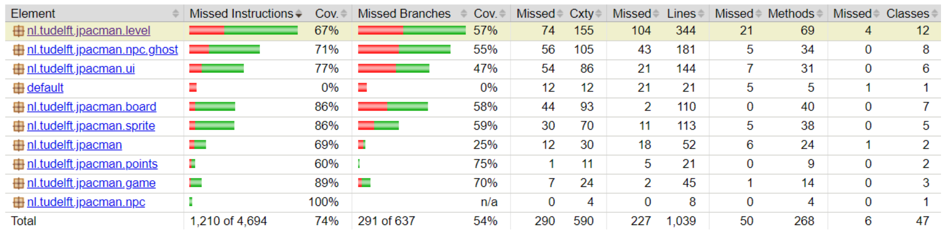
```
void testGameDoesNotStartWhenNoPelletsRemain() {
    // Arrange
    when(level.isAnyPlayerAlive()).thenReturn( t: true);
    when(level.remainingPellets()).thenReturn( t: 0);

    // Act
    game.start();

    // Assert
    assertThat(game.isInProgress()).isFalse();
    verify(level, never()).start();
```

- The coverage analysis after implementation of this test is shown below.
    - 

level	23% (3/13)	8% (7/78)	5% (19/349)	1% (3/167)
integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
game	33% (1/3)	50% (7/14)	65% (28/43)	57% (8/14)

    - We see a slight increase to level coverage and big increase to the game coverage as a result of the tests.

Jacoco Report VS IntelliJ

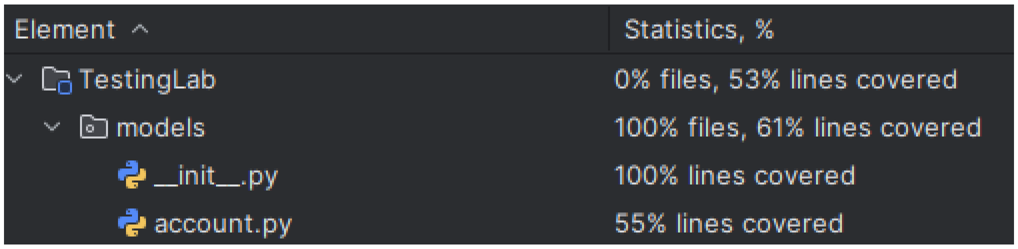
- The included Jacoco coverage analysis is shown below.
    - 

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	🟥🟩	89%	🟥🟩	70%	7	24	2	45	1	14	0	3
nl.tudelft.jpacman.npc	🟩	100%		n/a	0	4	0	8	0	4	0	1
Total	1,210 of 4,694	74%	291 of 637	54%	290	590	227	1,039	50	268	6	47

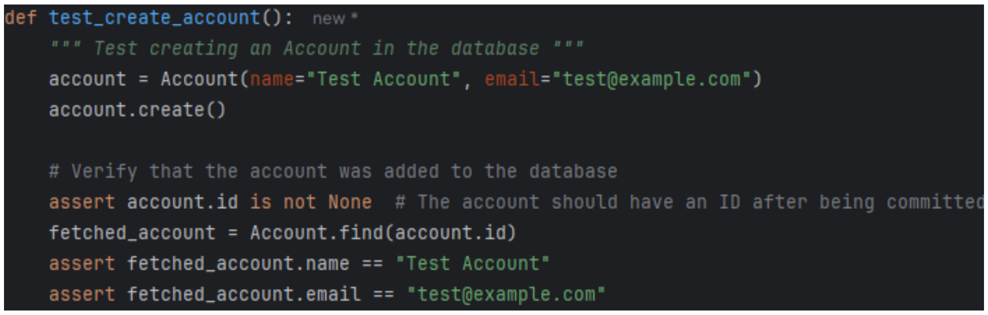
- We can see that the Jacoco report shows more in depth analysis of what lines were covered and what was missed.
- I find that the Jacoco is easier to understand and navigate when it comes to coverage analysis, although a slight drawback is that it is not

directly implemented into the IDE, however I believe the better visibility and navigation to be of more value.

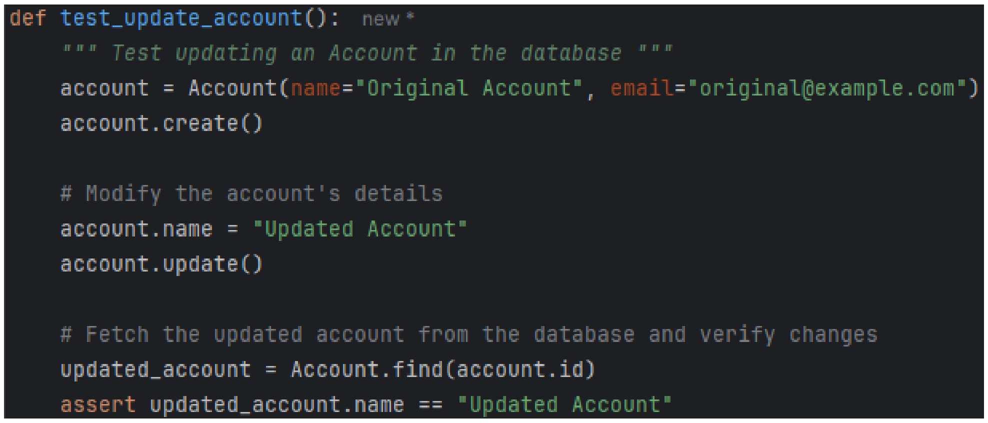
Python Test Coverage

- For this task we are given an account.py file and a test_account.py file. Our objective is to provide full coverage of the file and all of its definitions. Below is the initial coverage testing
    - 

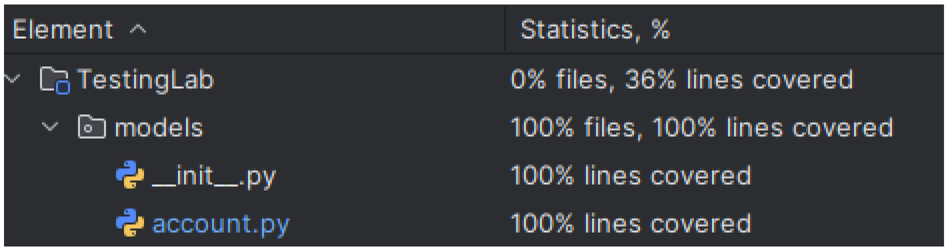
Element	Statistics, %
TestingLab	0% files, 53% lines covered
models	100% files, 61% lines covered
__init__.py	100% lines covered
account.py	55% lines covered

- There are a few definitions that required tests to be made, they are as follows:
    - From_dict
    - Create
    - Update
    - Delete
    - All
    - Find
- Provided below are some of the implementations of test cases for these definitions
    - 

```
def test_create_account():  new *
    """ Test creating an Account in the database """
    account = Account(name="Test Account", email="test@example.com")
    account.create()

    # Verify that the account was added to the database
    assert account.id is not None  # The account should have an ID after being committed
    fetched_account = Account.find(account.id)
    assert fetched_account.name == "Test Account"
    assert fetched_account.email == "test@example.com"
```

- 

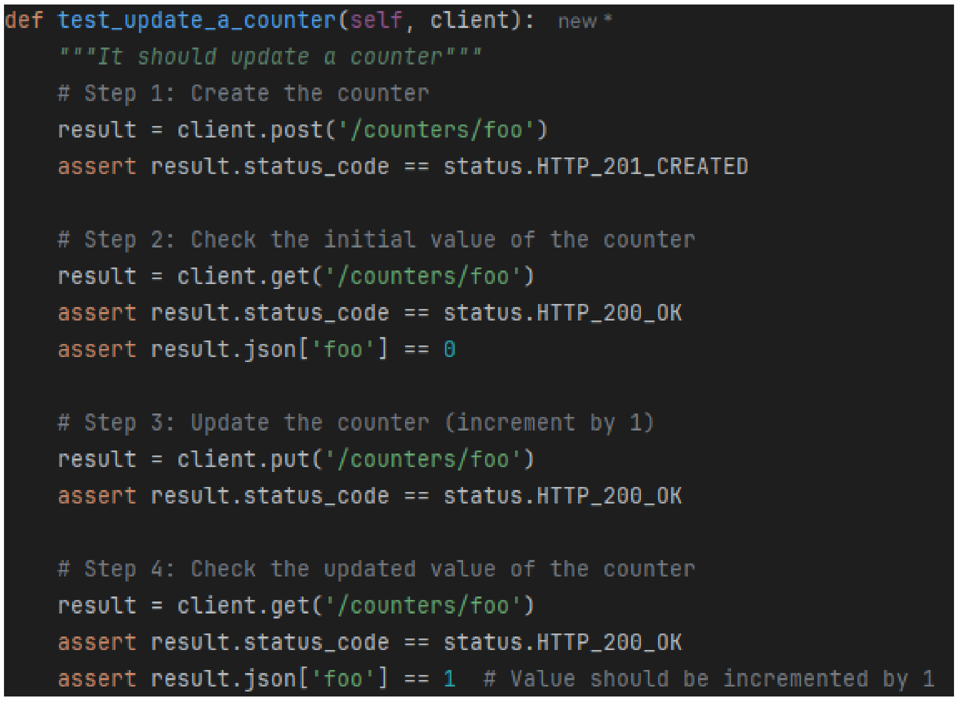
```python
def test_update_account():  new *
    """ Test updating an Account in the database """
    account = Account(name="Original Account", email="original@example.com")
    account.create()

    # Modify the account's details
    account.name = "Updated Account"
    account.update()

    # Fetch the updated account from the database and verify changes
    updated_account = Account.find(account.id)
    assert updated_account.name == "Updated Account"
```

- After implementing these tests for all definitions the new coverage report is as follows:
  - 

Element	Statistics, %
TestingLab	0% files, 36% lines covered
models	100% files, 100% lines covered
__init__.py	100% lines covered
account.py	100% lines covered

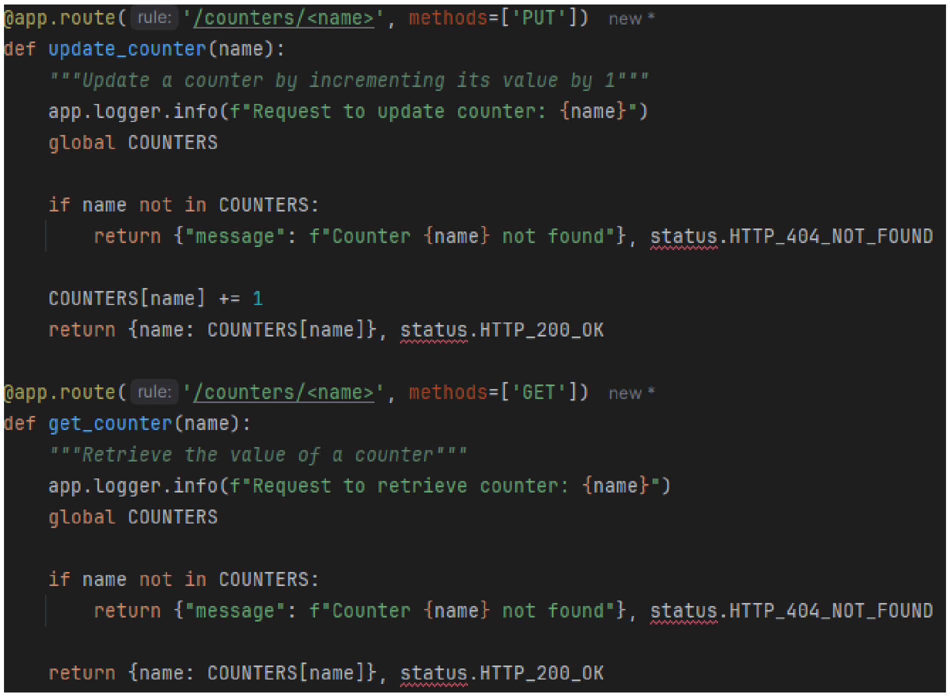
TDD

- For this task we are to implement test cases first that we will then use to develop the proper definitions in order to pass said test cases. This is the Test Driven Development approach otherwise known as TDD.
- First part of the task was to create the "test_update_a_counter". The code for it is listed below.

- 

```
def test_update_a_counter(self, client):  new *
    """It should update a counter"""
    # Step 1: Create the counter
    result = client.post('/counters/foo')
    assert result.status_code == status.HTTP_201_CREATED

    # Step 2: Check the initial value of the counter
    result = client.get('/counters/foo')
    assert result.status_code == status.HTTP_200_OK
    assert result.json['foo'] == 0

    # Step 3: Update the counter (increment by 1)
    result = client.put('/counters/foo')
    assert result.status_code == status.HTTP_200_OK

    # Step 4: Check the updated value of the counter
    result = client.get('/counters/foo')
    assert result.status_code == status.HTTP_200_OK
    assert result.json['foo'] == 1  # Value should be incremented by 1
```

- When running this code we see an error message due to the proper function not being implemented yet.
- The next step was to implement the proper function to pass this test. The code is as follows.



```
@app.route( rule: '/counters/<name>', methods=['PUT'])  new *
def update_counter(name):
    """Update a counter by incrementing its value by 1"""
    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

    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK

@app.route( rule: '/counters/<name>', methods=['GET'])  new *
def get_counter(name):
    """Retrieve the value of a counter"""
    app.logger.info(f"Request to retrieve 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
```

- With this code implemented the test cases pass and are fully covered.