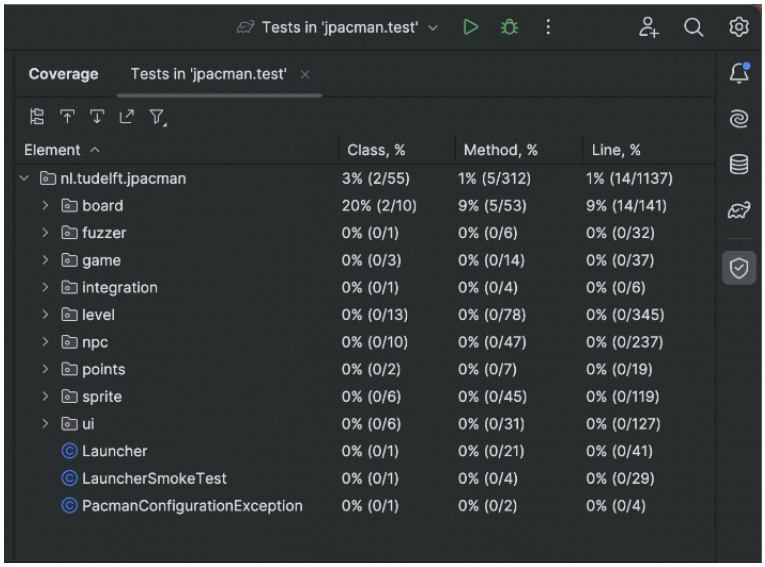
Xiaojun Ruan
Feb 5, 2024
https://github.com/junxjr/cs472-group4

Unit Test Report

Task 1:

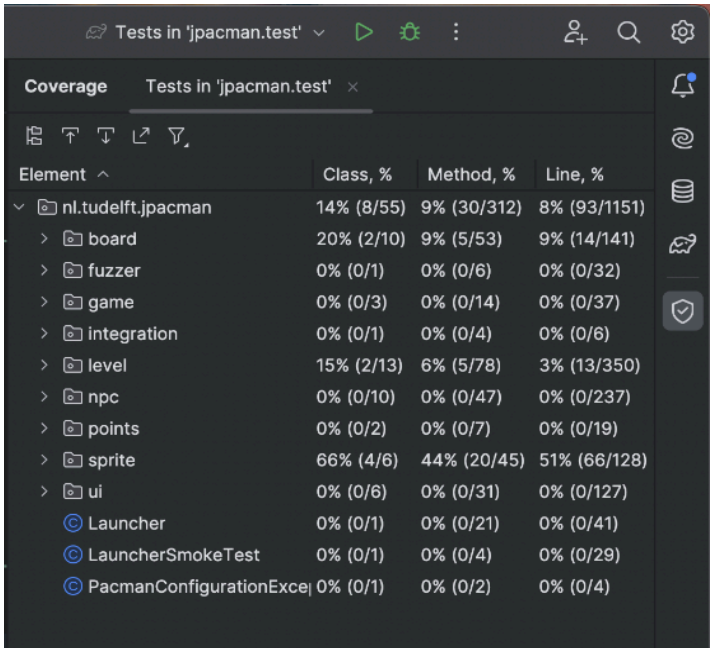
This is a screenshot of the coverage when I first ran the jpacman. The coverage is not good enough.



Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
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	0% (0/13)	0% (0/78)	0% (0/345)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	0% (0/6)	0% (0/45)	0% (0/119)
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)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

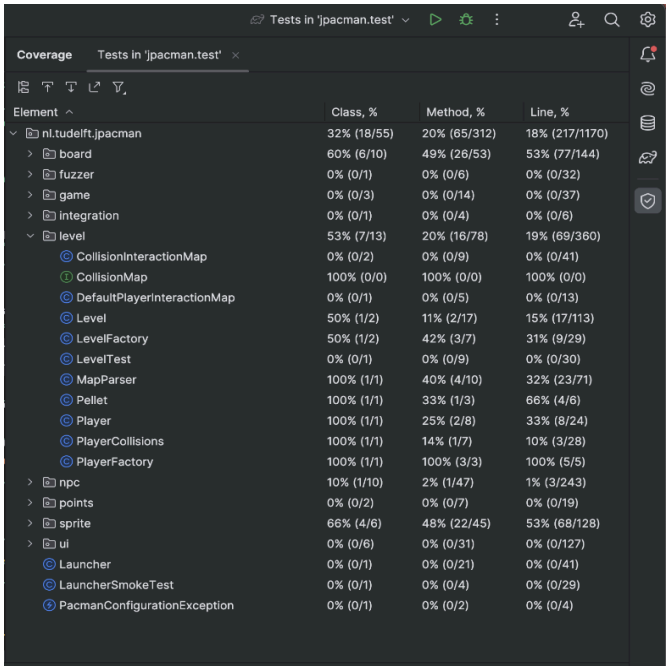
Task 2.1:

This is a screenshot of the coverage before the tests were implemented, it has the PlayerTest.



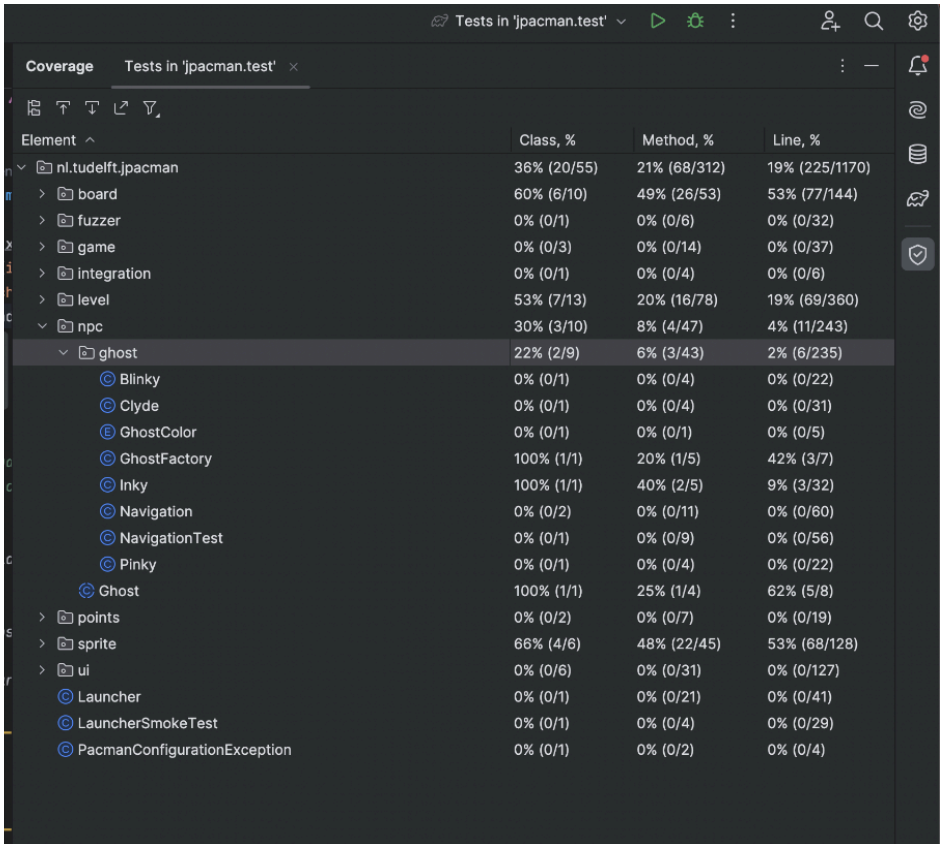
After implementing the unit test for MapParser.parseMap, the coverage went up to 20% for the methods. I created objects for all the arguments I needed such as new BoardFatory, new GhostFactory, PointCalculator, and LevelFactory. In order to keep it simpler, I imported the PlayerTest to use the objects I created for Store_Sprite and new Player. Into the testParseMap, 5 arguments were created, then I called the MapParser.parseMap with the map argument that was created. In the end, I used assertThat.isNotNull to ensure that the map is not null. Here are the screenshots of both the unit test and coverage.

```
new *
public class MapParserTest {
    1 usage
    private final BoardFactory The_Board_Factory = new BoardFactory(PlayerTest.get_Store_Sprite());
    1 usage
    private static final GhostFactory Ghost_Factory = new GhostFactory(PlayerTest.get_Store_Sprite());
    1 usage
    @Mock
    private PointCalculator Point_Calculator;
    1 usage
    private final LevelFactory Level_Factory = new LevelFactory(PlayerTest.get_Store_Sprite(), Ghost_Factory, Point_Calculator);
    1 usage
    private final MapParser Map_Parser = new MapParser(Level_Factory, The_Board_Factory);

    new *
    @Test
    void testParseMap(){
        char[][] map = {{'.', '.', '.', '.'}, {'.', '.', '.', '.'}, {'.', '.', '.', '.'}, {'.', '.', '.', '.'}};
        int width = 4, height = 4;
        Square[][] grid = new Square[width][height];

        List<Ghost> ghost;
        ghost = new ArrayList<>();

        List<Square> Start_Positions;
        Start_Positions = new ArrayList<>();

        Map_Parser.parseMap(map);
        assertThat(grid).isNotNull();

    }
}
```



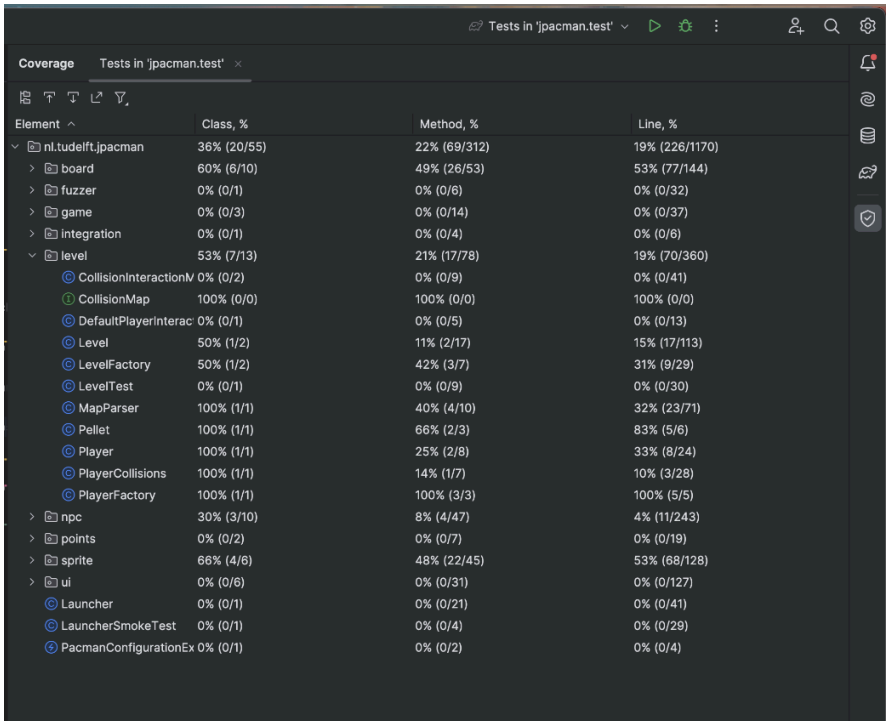
The second unit test was for Inky.Inky, and the coverage went up to 21% for the methods. With the testInky(), a hashMap argument was needed, then I passed in the hashmap to create a new Inky object. Finally, I used assertThat.isNotNull to ensure that the New_Inky is not null. Here are the screenshots of both the unit test and coverage.

```
new *
public class InkyTest {
    new *
    @Test
    void testInky() {
        Map<Direction, Sprite> Sprite_Map = new HashMap<>();
        Inky New_Inky = new Inky(Sprite_Map);
        assertThat(New_Inky).isNotNull();
    }

}
```



The last unit test was for the LevelFactory.createPellet and the coverage went up to 22% for the methods. Again, starting off with creating all the arguments I need to pass in to create a new LevelFactory for me to call the function createPellet. Then with testCreatePallet(), I called assertThat.isNotNull with the new_pallet to make sure it is not null, then I call assertEquals with the pelletValue that was initially set to 10, and the value of the new_Pellet.getValue(). Here are the screenshots of both the unit test and coverage.

```java
new *
public class LevelFactoryTest {
    1 usage
    int Pellet_value = 10;
    no usages
    PacManSprites sprite = mock(PacManSprites.class);
    1 usage
    @Mock
    private PointCalculator Point_Calculator;
    3 usages
    private static final PacManSprites Store_Sprite = new PacManSprites();
    1 usage
    private final PlayerFactory The_Factory = new PlayerFactory(Store_Sprite);
    no usages
    private final Player New_Player = The_Factory.createPacMan();
    1 usage
    private static final GhostFactory Ghost_Factory = new GhostFactory(Store_Sprite);

    1 usage
    private final LevelFactory Level_Factory = new LevelFactory(Store_Sprite, Ghost_Factory, Point_Calculator);
    2 usages
    Pellet new_pellet = Level_Factory.createPellet();

    new *
    @Test
    void testCreatePallet(){
        assertThat(new_pellet).isNotNull();
        assertEquals(Pellet_value, new_pellet.getValue());
    }
}
```



Task 3:

1. Yes, it is similar. It is because they both showed the parts that the unit tests have covered. For example, the testAlive() was covered after the implementation of the PlayerTest, and reported on both JaCoCo and intelliJ.

2. Yes, it was clearly highlighted with different colors, and I can indicate which branch is not being exercised by looking at the red highlighted lines. It also showed partly covered which is very helpful.
3. Personally, I would prefer JaCoCo. It is better to visualize by looking at the actual line of code instead of percentages. It is harder to tell the specific parts that are not covered with IntelliJ.

Task 4:

Working with the parts that were not covered, I started off with lines 34-35 for test_from_dict. Just like the test_to_dict, the from_dict test will be setting the attributes from a dictionary instead. First, I generated a random account, and then using account.from_dict(data) to populate from the account in the dictionary, then I used assertEqual to make sure all the information matches.

```
def test_from_dict(self):
    """ Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.from_dict(data)
    self.assertEqual(account.name, data.get("name"))
    self.assertEqual(account.email, data.get("email"))
    self.assertEqual(account.phone_number, data.get("phone_number"))
    self.assertEqual(account.disabled, data.get("disabled"))
    self.assertEqual(account.date_joined, data.get("date_joined"))
```

Going into lines 45-48, a test for update was needed. An account was created first with generating a random account. I used a try catch block to ensure that the account is not empty. The next step is to get the account id and set it as the updated account object, then use assertEqual with the account and the updated account to verify if it matches. In order to cover line 47, another test was needed for test_update_emptyIdAccount. I created an account first then try to raise the DataValidationError for any accounts that are not in the database.

```
def test_update(self):
    """ Test account update """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()

    try:
        account.update()    # try to update the account
    except DataValidationError as empty:
        with self.assertRaises(DataValidationError) as e:
            account.update()    # if the account is empty, then throw

    updated = Account.query.get(account.id) # get the information

    self.assertEqual(account.name, updated.name)    # check if matches
    self.assertEqual(account.email, updated.email)
    self.assertEqual(account.phone_number, updated.phone_number)
    self.assertEqual(account.disabled, updated.disabled)
    self.assertEqual(account.date_joined, updated.date_joined)

def test_update_emptyIdAccount(self):
    """ Test account update with empty id"""
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()

    emptyIdAccount = Account()  # for accounts that is not in the database
    with self.assertRaises(DataValidationError) as empty:
        emptyIdAccount.update()
```

The next is to cover the delete test part for lines 52-54. For this part, all I needed to do is to create an account and then verify the account exists first, then I implement the delete().

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

    account_to_delete = Account.query.get(account.id)
    self.assertTrue(account_to_delete) # check if the account is in the database

    account_to_delete.delete()  # delete the account
    after_deleted = Account.query.get(account.id)
```

Finally, the last section is testing the test_find() for lines 74-75. This part is very similar to the delete testing. After I created an account, I just needed to find the account with Account.find and then use assertTrue to ensure that the account is found in the database.

```
def test_find(self):
    """ Test account find """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()

    found = Account.find(account.id)    # find the account
    self.assertTrue(found)
```

After running the nosetests, the coverage turned out to be 100% with no errors.

```
Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data
- Test account delete
- Test account find
- Test account from dict
- Test the representation of an account
- Test account to dict
- Test account update
- Test account update with empty id

Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
models/__init__.py      7      0   100%
models/account.py      40      0   100%
-------------------------------------------------
TOTAL                  47      0   100%
-------------------------------------------------
Ran 9 tests in 0.474s

OK
```

Task 5:

The RED Phase. Since we only implement the test_counter.py in this step, it will be in the RED Phase because the endpoint has not been created yet in counter.py. By following the instructions given, I first created a counter and ensured that the return code went successfully with assertEquals. Then I get the baseline value, and use self.client.put to update the counter, then ensure the return code went successfully. Finally, I stored the value of baseline+1 and then made sure that the return code is baseline+1.

```
def test_update_a_counter(self):
    """It should update a counter"""
    create = self.client.post('/counters/update')
    self.assertEqual(create.status_code, status.HTTP_201_CREATED)

    baseline = create.get_json()['update']
    updated = self.client.put('/counters/update')
    self.assertEqual(updated.status_code, status.HTTP_200_OK)

    inc_baseline = baseline + 1
    self.assertEqual(inc_baseline, updated.get_json()['update'])
```

Here is the RED Phase for reading a counter. Just like updating, the endpoint has not been created. I used post to create a counter first, and then read the counter with get. For both create and read, I used assertEqual to ensure that the return code went successfully.

```
def test_read_a_counter(self):
    """It should read a counter"""
    create = self.client.post('/counters/read')
    self.assertEqual(create.status_code, status.HTTP_201_CREATED)

    read = self.client.get('/counters/read')
    self.assertEqual(read.status_code, status.HTTP_200_OK)
```

The GREEN Phase. This is the part after I implemented the counter.py. Starting with updating a counter. The first thing I needed to do was create a route on the endpoint for method PUT to update the counter. With my global counter, I incremented it by 1, then returned it with the 200_OK return code.

```python
# Create a route for method PUT on endpoint /counters/<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

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

Here is the GREEN Phase for reading a counter. I created a route on the endpoint for method GET to read a counter. With the global counters, I returned it with the 200_OK return code.

```python
# Create a route for method PUT on endpoint /counters/<name>.
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    """read a counter"""
    app.logger.info(f"Request to read counter: {name}")
    global COUNTERS

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

Finally, the Refactor Phase. A setUp() was created because the code self.client = app.test_client() was being used many times. The refactor will help for code that will be used many times.

```python
def setUp(self):
    self.client = app.test_client()
```

After everything was implemented, my nosetests passed with 100% coverage.

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

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 0.274s

OK
```