

Ryan Parker  
Professor Businge  
CS 472  
26 January 2024

## CS472 - Dynamic Analysis Report

Repository: <https://github.com/munch2024/munch>

Forked Repository: <https://github.com/rparker2003/cs472project-fork>

### Task 2.1











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)
Ⓢ LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
⚡ PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

Figure 1. JPacman Test Coverage After Writing level/PlayerTest

For this task, I wrote four tests for the following methods:

- src/main/java/nl/tudelft/jpacman/board/Board.getWidth() / getHeight()

```
public class BoardSizeTest {
    private final Square[][] grid = createFilledGrid(2, 4);
    private final Board board = new Board(grid);
    @Test
    void getWidth() {
        assertThat(board.getWidth()).isGreaterThan(0);
    }
    @Test
    void getHeight() {
        assertThat(board.getHeight()).isGreaterThan(0);
    }
    // function to create and fill the grid
    private Square[][] createFilledGrid(int width, int height) {
        Square[][] filledGrid = new Square[width][height];
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                filledGrid[x][y] = new BasicSquare();
            }
        }
        return filledGrid;
    }
}
```

- src/main/java/nl/tudelft/jpacman/board/BoardFactory.createWall()

```
public class CreateWallTest {
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    private static final BoardFactory temp = new BoardFactory(SPRITE_STORE);

    @Test
    void createWall(){ assertEquals(temp.createWall()).isNotNull(); }
}
```

- src/main/java/nl/tudelft/jpacman/level/Pellet.getValue()

```
public class PelletValueTest {
    private static final EmptySprite EmptyAccessorSprite = new EmptySprite();
    private static final Sprite temp = EmptyAccessorSprite.split(0, 0, 0, 0);
    private static final Pellet PelletVariable = new Pellet(3, temp);

    @Test
    void getValue() { assertEquals(PelletVariable.getValue()).isGreaterThan(0); }
}
```

Element ^	Class, %	Method, %	Line, %
✓  nl.tudelft.jpacman	25% (14/55)	14% (46/312)	11% (128/1155)
>  board	60% (6/10)	32% (17/53)	29% (42/144)
>  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)	8% (7/78)	5% (18/351)
>  npc	0% (0/10)	0% (0/47)	0% (0/237)
>  points	0% (0/2)	0% (0/7)	0% (0/19)
>  sprite	83% (5/6)	48% (22/45)	53% (68/128)
>  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)

Figure 2. JPacman Test Coverage After Writing the Above Methods

As you can see in Figure 1, the coverage for the board was 20%, and the level was 15% due to the PlayerTest written in task two. The total source code coverage was 14% class, 9% method, and 8% lines. There is little coverage; therefore, we must write more tests. I wrote four new tests in total throughout the board and level packages, and as you can see in Figure 2, the coverages for the jpacman package after writing the four tests are 25% class, 14% method, and 11% lines. The new coverage for the board was 60%, and the level was 23%. None of the other categories went up, as there are no tests for them, except for the sprite package because I used it when creating some of my new tests.

### Task 3




















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
default		0%		0%	12 12	21 21	5 5	1 1
nl.tudelft.jpacman		69%		25%	12 30	18 52	6 24	1 2
nl.tudelft.jpacman.board		86%		58%	44 93	2 110	0 40	0 7
nl.tudelft.jpacman.game		87%		60%	10 24	4 45	2 14	0 3
nl.tudelft.jpacman.level		67%		57%	74 155	104 344	21 69	4 12
nl.tudelft.jpacman.npc		100%	n/a	n/a	0 4	0 8	0 4	0 1
nl.tudelft.jpacman.npc.ghost		71%		55%	56 105	43 181	5 34	0 8
nl.tudelft.jpacman.points		60%		75%	1 11	5 21	0 9	0 2
nl.tudelft.jpacman.sprite		87%		59%	29 70	10 113	4 38	0 5
nl.tudelft.jpacman.ui		77%		47%	54 86	21 144	7 31	0 6
Total	1,209 of 4,694	74%	293 of 637	54%	292 590	228 1,039	50 268	6 47

Figure 3. JaCoCo Report on JPacman

The coverage results for the JaCoCo report are similar to the ones that I got from IntelliJ in the previous task, however, JaCoCo gives much more information that is very useful to determine the next step in development. JaCoCo also has colored depictions of the coverage for the instructions and branches and I believe this is very helpful as it gives a visual representation of the coverage instead of just relying on a percentage. I prefer JaCoCo's report over the IntelliJ coverage window because it provides a lot more information about the coverage and makes that information easier to understand.

### Task 4

Name	Stmts	Miss	Cover	Missing
models\__init__.py	7	0	100%	
models\account.py	40	13	68%	26, 30, 34-35, 45-48, 52-54, 74-75
TOTAL	47	13	72%	

Ran 2 tests in 0.460s

Figure 4. Original Python Test Coverage

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 0.483s

Figure 5. Python Test Coverage After Writing New Tests

In total, I wrote four new tests to achieve 100% coverage for this Python repository.

- test\_from\_dict

```
def test_from_dict(self):
    """ Test creating an Account from a dictionary """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)

    new_account = Account()
    new_account.from_dict(account.to_dict())

    self.assertEqual(account.name, new_account.name)
    self.assertEqual(account.email, new_account.email)
```

- test\_update

```
def test_update(self):
    """ Test account update """
    for data in ACCOUNT_DATA:
        account = Account(**data)
        account.create()

    new_name, new_email = "Ryan", "Ryan12345@gmail.com"
    account.name = new_name
    account.email = new_email

    account.update()
    updated_account = Account.find(account.id)

    self.assertEqual(updated_account.name, new_name)
    self.assertEqual(updated_account.email, new_email)

    account_no_id = Account(**data)
    with self.assertRaises(DataValidationError) as context:
        account_no_id.update()
    self.assertEqual(str(context.exception), "Update called with empty ID field")
```

- test\_delete

```
def test_delete(self):
    """Test account delete"""
    for data in ACCOUNT_DATA:
        account = Account(**data)
        account.create()
    random = self.rand

    account = Account.query.get(random)
    account.delete()

    self.assertIsNone(Account.query.get(random))
```

- test\_find

```
def test_find(self):
    """Test find account"""
    for data in ACCOUNT_DATA:
        account = Account(**data)
        account.create()
    random = self.rand

    account = Account.find(random)
    result = ACCOUNT_DATA[(random-1)]

    self.assertEqual(account.name, result["name"])
    self.assertEqual(account.email, result["email"])
```

## Task 5

- Code for creating the counter for both red and green phases

```
@app.route('/counters/<name>', methods=['POST'])
def create_counter(name):
    """Create a counter"""
    app.logger.info(f"Request to create counter: {name}")
    global COUNTERS
    if name in COUNTERS:
        return {"Message": f"Counter {name} already exists"}, status.HTTP_409_CONFLICT
    COUNTERS[name] = 0
    return {name: COUNTERS[name]}, status.HTTP_201_CREATED
```

```
def test_create_a_counter(self):
    """It should create a counter"""
    client = app.test_client()
    result = client.post('/counters/foo')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
```

- Code for updating the counter for both red and green phases

```
@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 in COUNTERS:
        COUNTERS[name] += 1
    else:
        COUNTERS[name] = 1
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_204_NO_CONTENT

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

```
def test_update_a_counter(self):
    """It should update a counter"""
    # 1. Make a call to Create a counter.
    result = self.client.post('/counters/update')

    # 2. Ensure that it returned a successful return code.
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    # 3. Check the counter value as a baseline.
    baseline_value = result.get_json()['update']
    self.assertEqual(baseline_value, 0)

    # 4. Make a call to Update the counter that you just created.
    updated_result = self.client.put('/counters/update')

    # 5. Ensure that it returned a successful return code.
    self.assertEqual(updated_result.status_code, status.HTTP_200_OK)

    # 6. Check that the counter value is one more than the baseline you measured in step 3.
    updated_value = updated_result.get_json()['update']
    self.assertEqual(updated_value, baseline_value+1)

    # Check that updating a new client returns a proper return code.
    result2 = self.client.put('/counters/update_fail')
    self.assertEqual(result2.status_code, status.HTTP_204_NO_CONTENT)
```

- Code for reading the counter for both red and green phases

```
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    """Read a counter"""
    app.logger.info(f"Request to read 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
```

```
def test_read_a_counter(self):
    """It should read a counter"""
    # Make a call to create a counter, and put the counter
    self.client.post('/counters/read')
    self.client.put('/counters/read')

    # Make a call to get a counter, and ensure it returned a good code
    result = self.client.get('/counters/read')
    self.assertEqual(result.status_code, status.HTTP_200_OK)

    # Check that reading a fake client returns a proper return code.
    result2 = self.client.get('/counters/read_fail')
    self.assertEqual(result2.status_code, status.HTTP_404_NOT_FOUND)
```

This task was the hardest for this assignment. Understanding how the post, put, and get operations work was difficult, but determining how to get the values of the counters without

accessing the COUNTERS array in counter.py was the most challenging for me. I did not have to do much refactoring as most of the code did not repeat.

```
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	25	0	100%	
src\status.py	6	0	100%	
TOTAL	31	0	100%	

```
Ran 4 tests in 0.136s
```

Figure 6. Coverage for Python Testing Lab After Writing Above Tests

After writing the required code and tests for creating, updating, and reading counters and testing for duplicate counters for the Python Testing Lab, I got 100% coverage with 25 statements in src/counter.py.