Taylor Finelli
CS472 - Software Development I
5 February, 2024

<div align="center">

JaCoCo Report on JPacman
Fork repo link: https://github.com/taylorfinelli/Barbell.git

</div>

Task 1

Is the coverage good enough?

No, the coverage is not enough. Coverage should be around 90% at the minimum, and this coverage is only around 3%.

Task 2

Code coverage before increasing coverage

Element ▲	Class, %	Method, %	Line, %
⌄ 📁 nl.tudelft.jpacman	14% (8/...	9% (29/312)	8% (92/11...
› 📁 board	20% (2/...	7% (4/53)	9% (13/136)
› 📁 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/...	6% (5/78)	3% (13/334)
› 📁 npc	0% (0/10)	0% (0/47)	0% (0/236)
› 📁 points	0% (0/2)	0% (0/7)	0% (0/19)
› 📁 sprite	66% (4/6)	44% (20/...	51% (66/1...
› 📁 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)
⊘ PacmanConfigurationEx	0% (0/1)	0% (0/2)	0% (0/4)

Code snippets for unit testing:

Test 1: **src/main/java/nl/tudelft/jpacman/level/Player.addPoints**

```java
@Test
void testAddPoints(){

    ThePlayer.addPoints(5);

    assertThat(ThePlayer.getScore()).isEqualTo(5);

}
```

Test 2: **src/main/java/nl/tudelft/jpacman/level/Player.getScore**

```java
@Test
void testGetScore(){

    assertThat(ThePlayer.getScore()).isEqualTo(0);

}
```

Test 3:
src/main/java/nl/tudelft/jpacman/npc/ghost/GhostFactory.createPinky

```java
@BeforeEach
void setup(){

    PacManSprites pacManSprites = Mockito.mock(PacManSprites.class);

    ghostFactory = new GhostFactory(pacManSprites);

}


@Test
void testCreatePinky(){

    assertTrue(ghostFactory.createPinky() instanceof Pinky);

}
```

Code coverage after increasing coverage

Element ▲	Class, %	Method, %	Line, %
∨ 📁 nl.tudelft.jpacman	21% (12...	12% (38/3...	9% (113/1...
> 📁 board	20% (2/...	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/...	8% (7/78)	4% (16/3...
> 📁 npc	40% (4/...	12% (6/47)	6% (17/24...
> 📁 points	0% (0/2)	0% (0/7)	0% (0/19)
> 📁 sprite	66% (4/6)	44% (20/...	51% (66/1...
> 📁 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)
⚡ PacmanConfigurationExc	0% (0/1)	0% (0/2)	0% (0/4)

Task 3

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

IntelliJ coverage results

Element ▲	Class, %	Method, %	Line, %
∨ 📁 nl.tudelft.jpacman	21% (12/55)	12% (38/312)	9% (113/1157)
> 📁 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)	8% (7/78)	4% (16/350)
> 📁 npc	40% (4/10)	12% (6/47)	6% (17/243)
> 📁 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)

JaCoCo coverage results

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

As you can see, the two coverage reports are not the same. The difference is because each coverage report has a different focus and they calculate it differently. JaCoCo calculates based on *instructions* and *branches* (I assume it examines the Java ByteCode), which is different from what IntelliJ calculates coverage on. IntelliJ shows the coverage based on lines of code that get used.

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

Yes I did find the source code visualization from JaCoCo helpful. I like how it will highlight in red the lines that were missed and will highlight in green the lines that were checked. This is especially useful because instead of it just saying which method was tested, we can see which lines inside of the method were tested. This ensures that we can get complete coverage for our unit testing.

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

I personally liked the report from JaCoCo more than the report from IntelliJ. I think it is more user-friendly and more visually appealing. I'm not really much of a fan of the dropdown-style menu that IntelliJ uses. I like the organization style of clicking through each package and only seeing stuff from that package. Still, my favorite feature is the highlighting of the lines to show what has been covered and what needs to be covered.

Task 4

Unit tests in Python

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

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

def test_update_with_id(self):
    """ Test account update with id """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account.name = "12345"
    account.update()
    updated_account = Account.find(account.id)
    self.assertEqual(updated_account.name, "12345")

def test_update_no_id(self):
    """ Test account update without id """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    with self.assertRaises(DataValidationError):
        account.update()
```

100% coverage report

```
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 0.254s


OK
```

Task 5

Red Phase

We need to test and see if the counter is working correctly. To do this, we will attempt to do the request using a PUT method and verify if it increased by the correct value. To retrieve it, we will implement a GET request.

Version 1 of nosetest:

```python
def test_update_a_counter(self):
    """It should update an existing counter"""
    baseline = self.client.post('/counters/baz')
    self.assertEqual(baseline.status_code, status.HTTP_201_CREATED)
    response = self.client.put('/counters/baz')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.json['baz'], baseline.json['baz'] + 1)


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

Note: These return a 405 error since we have not created the routes for the PUT and GET request methods yet.

Green Phase

To get the REST API calls to work, we need to create a route for them, otherwise they will always just return a 405 error.

```python
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update")
    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


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

Refactor Phase

There are currently two lines that are not being tested:

```
Name                     Stmts   Miss  Cover   Missing
------------------------------------------------------
src/counter.py              24      2    92%   31, 41
src/status.py                6      0   100%
------------------------------------------------------
TOTAL                       30      2    93%
```

To make sure these lines get tested, we will add the following test cases:

```python
def test_update_a_counter_DNE(self):
    """It should return an error for trying to update a non-existing counter"""
    response = self.client.put('/counters/DNE')
    self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)

def test_get_a_counter_DNE(self):
    """It should return an error for trying to get a non-existing counter"""
    response = self.client.get('/counters/DNE')
    self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```

These test cases are sure to check the 404 errors.

We now have 100% test coverage.

```
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 0.092s

OK
```