Blake Gilmore

Unit Test Report

Task 2

The fully qualified method names of the methods I tested for question 2.1 are:

src\main\java\nl\tudelft\jpacman\level\LevelFactory.createGhost
src\main\java\nl\tudelft\jpacman\npc\ghost\MapParser.checkMapFormat
src\main\java\nl\tudelft\jpacman\level\PlayerCollisions.playerVersusGhost

LevelFactory.createGhost

```
public class LevelFactoryTest {

    2 usages
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    1 usage
    private static final GhostFactory testFactory = new GhostFactory(SPRITE_STORE);
    1 usage
    private static final PointCalculator pointCalculator = new PointCalculatorLoader().load();

    5 usages
    private static final LevelFactory LFactory = new LevelFactory(SPRITE_STORE, testFactory, pointCalculator);
    new *
    @Test
    void testCreateGhost(){
        //Test for all different ghosts and make sure it loops back around to Blinky
        assertThat(LFactory.createGhost()).isInstanceOf(Blinky.class);
        assertThat(LFactory.createGhost()).isInstanceOf(Inky.class);
        assertThat(LFactory.createGhost()).isInstanceOf(Pinky.class);
        assertThat(LFactory.createGhost()).isInstanceOf(Clyde.class);
        assertThat(LFactory.createGhost()).isInstanceOf(Blinky.class);
    }
}
```

My test for LevelFactory.createGhost is intended to verify that creating ghosts will properly loop across Blinky, Inky, Pinky, and Clyde. I first set up the level factory to create ghosts, and repeated the task of creating ghosts 5 times. Each time I checked if the ghosts were instances of the classes of ghosts they were supposed to be. By doing this, we know the counter in the LevelFactory which is responsible for making sure the first four ghosts are of different types is working to populate the ghosts properly.

MapParser.checkMapFormat

```java
public class MapParserTest {
    3 usages
    PacManSprites sprites = new PacManSprites();
    1 usage
    LevelFactory levelFactory = new LevelFactory(
        sprites,
        new GhostFactory(sprites),
        mock(PointCalculator.class));
    4 usages
    private final MapParser parser = new MapParser(levelFactory, new BoardFactory(sprites));

    /**
     * Checks that map format will throw errors for incorrect
     * map formats.
     * @throws IOException
     */
```

```java
    @Test
    void checkMapFormatTest() {
        //Null list
        assertThatThrownBy(() -> {
            parser.parseMap((List<String>) null);
        }).isInstanceOf(PacmanConfigurationException.class);

        //Empty list
        assertThatThrownBy(() -> {
            parser.parseMap(Lists.newArrayList(new ArrayList<>()));
        }).isInstanceOf(PacmanConfigurationException.class);

        //Empty string in list
        assertThatThrownBy(() -> {
            parser.parseMap(Lists.newArrayList(Lists.newArrayList( ...elements: "", "")));
        }).isInstanceOf(PacmanConfigurationException.class);

        //Not a square
        assertThatThrownBy(() -> {
            parser.parseMap(Lists.newArrayList(Lists.newArrayList( ...elements: "###", "#")));
        }).isInstanceOf(PacmanConfigurationException.class);
    }
}
```

The MapParser class will take in an array of various strings to build the map that will be played. Typically this map is made up of ‘#’, ‘G’, ‘P’, and ‘ ‘ for walls, ghosts, player, and movement spaces respectively. The MapParser class checks for valid map configurations using the checkMapFormat method, which is private so we need to access it through the parseMap method. I checked the four cases that checkMapForatTeset looks for. A null input, an empty list, empty strings in the list, and unequal map dimensions. I simply sent a null List value, an empty ArrayList value, a list of values “” and “”, and a list of values “###” and “#” to verify that the proper exceptions were thrown within the MapParser class.

PlayerCollisions.playerVersusGhost

```
public class PlayerCollisionsTest {

    2 usages
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    1 usage
    private final PlayerFactory Factory = new PlayerFactory(SPRITE_STORE);
    3 usages
    private final Player ThePlayer = Factory.createPacMan();
    1 usage
    private final GhostFactory gFactory = new GhostFactory(SPRITE_STORE);
    1 usage
    private static final PointCalculator pointCalculator = new PointCalculatorLoader().load();
```

```
    @Test
    void playerVersusGhostTest(){
        //Create ghost
        Ghost killerGhost = gFactory.createBlinky();
        //Make sure player starts alive
        assertThat(ThePlayer.isAlive()).isEqualTo( expected: true);

        PlayerCollisions playerCollisions = new PlayerCollisions(pointCalculator);
        //Kill player
        playerCollisions.playerVersusGhost(ThePlayer, killerGhost);
        assertThat(ThePlayer.isAlive()).isEqualTo( expected: false);
    }
```

The playerVersusGhost method is intended to make sure the player collision between a player and a ghost in the PlayerCollisionsTest class will kill the player as intended. This is simply done by creating the instances of the Ghost and the Player objects, and passing them into the playerVersusGhost method, then verifying that the Player object’s isAlive method is changed to false. This extends the test case in question 2 of the lab because it needs to first check that the player is alive when the test starts.

Test Coverage Improvements

The following Screenshots also show my test coverage progress:

Class, %	Method, %	Line, %
3% (2/55)	1% (5/312)	1% (14/1137)

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)

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacma	30% (17/55)	16% (52/312)	13% (154/1160)
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	23% (3/13)	8% (7/78)	7% (26/352)
npc	70% (7/10)	31% (15/47)	14% (35/243)
points	50% (1/2)	57% (4/7)	50% (10/20)
sprite	66% (4/6)	46% (21/45)	53% (69/128)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSm	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConf	0% (0/1)	0% (0/2)	0% (0/4)

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	36% (20/55)	18% (58/312)	15% (176/1163)
board	30% (3/10)	13% (7/53)	12% (18/143)
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	30% (4/13)	12% (10/78)	11% (42/353)
npc	70% (7/10)	31% (15/47)	14% (35/243)
points	50% (1/2)	57% (4/7)	50% (10/20)
sprite	66% (4/6)	46% (21/45)	53% (69/128)
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)
PacmanConfigurationEx	100% (1/1)	50% (1/2)	50% (2/4)

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	40% (22/55)	20% (64/312)	16% (195/1170)
board	30% (3/10)	13% (7/53)	12% (18/143)
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	38% (5/13)	17% (14/78)	15% (56/360)
npc	70% (7/10)	31% (15/47)	14% (35/243)
points	100% (2/2)	71% (5/7)	55% (11/20)
sprite	66% (4/6)	48% (22/45)	57% (73/128)
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)
PacmanConfiguratic	100% (1/1)	50% (1/2)	50% (2/4)

The test coverage gradually increased from 3%, 1%, 1% to 40%, 20%, 16% after implementing the four test cases from question 2.

Task 3

The coverage from JaCoCo is shown below:

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		69%		60%	70	155	100	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		71%		25%	11	30	16	52	5	24	0	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	1,189 of 4,694	74%	289 of 637	54%	288	590	223	1,039	50	268	5	47

On first glance the results seem fairly different. According to JaCoCo, my testing has much more coverage. It has my coverage of the instructions for the level package at 69%, but my line coverage as shown by IntelliJ is 15%. The totals on instruction counts are very different, and point to both methods considering line and instruction counts very differently. The branch count on JaCoCo however seems very useful. It'd be very helpful to know that classes are having certain cases checked and others not, so we can efficiently adjust the branches in the testing files to account for these faster. I would likely prefer to use IntelliJ's method simply for better useability. Instead of having to navigate to JaCoCo, IntelliJ having the Gradle results show in the tab is easier and faster, and feels like a natural extension of my programming flow rather than an add-on.

Task 4

The methods that were updated during Task 4 to obtain full coverage were test_from_dict, test_update, and test_delete

```python
def test_from_dict(self):
    """Create dictionary with values and test reverse of to_dict functionality"""
    data = ACCOUNT_DATA[self.rand] #Get random account
    account = Account(**data)

    """Create new account dictionary information"""
    new_info = {
        "id": 20,
        "name": "Test Name",
        "email": "test@email.com",
        "phone_number": "555-555-5555",
        "disabled": False,
        "date_joined": datetime(2000, 1, 1),
    }
    account.from_dict(new_info) #Create new account from dictionary

    self.assertEqual(account.id, new_info["id"])
    self.assertEqual(account.name, new_info["name"])
    self.assertEqual(account.email, new_info["email"])
    self.assertEqual(account.phone_number, new_info["phone_number"])
    self.assertEqual(account.disabled, new_info["disabled"])
    self.assertEqual(account.date_joined, new_info["date_joined"])
```

test_from_dict testing method

I used a variation of the test_to_dict testing function written during the lab. I first created an account object in the same way through selecting a random account from the given ACCOUNT_DATA in order to track changes in this account. Next I set up a simple test dictionary with test values for every variable that's tracked in an account object.

Using the account.from_dict function, I passed in the new dictionary, and asserted every account variable to make sure its new information matched the information from the dictionary I passed in. This is essentially a reversed version of test_to_dict by flipping the account variables and the dictionary indexing.

```python
def test_update(self):
    """Update item in database"""
    data = ACCOUNT_DATA[self.rand] #Get random account and add to database
    account = Account(**data)
    account.id = 1
    account.create() #Add account to database

    #Check what the name is in database
    test_account = Account.find(1)
    name_to_check = test_account.name #Get original name

    account.name = "Test Name"
    account.update()

    print(account.all())

    #Make sure that name has changed in the database
    test_account = Account.find(1)
    self.assertEqual(test_account.name, "Test Name")
    self.assertNotEqual(test_account.name, name_to_check)

    #Now make sure that it raises an error with a None id type
    with self.assertRaises(DataValidationError):
        account = Account(**data)
        account.id = None
        account.update()
```

test_update testing function

In order to properly test the test_update function, we first need to create an account from a random ACCOUNT_DATA value. By doing this, we know the sql backend has the account we want to update. I was sure to supply an id value for this account so I could search and retrieve it. Next, I retrieved it and kept the name stored to test that the update changes the name.

I then changed the account.name property to “Test Name”, and ran the account.update function. I once again retrieved the account from the find function with the same id number. I asserted two equalities to check that the updated name was “Test Name”, but also to check that the previous name and the new name were different.

There is also a branch in account.update that checks for a null id that is being searched for. I used an assertRaises call to expect the DataValidationError exception once an account with id value of None was attempted to be updated.

```python
    def testDelete(self):
        """Update item in database"""
        data = ACCOUNT_DATA[self.rand] #Get random account and add to database
        account = Account(**data)
        account.id = 1
        account.create() #Add account to database

        self.assertNotEqual(account.find(1), None) #Verify account exists

        #Delete the account
        account.delete()
        self.assertEqual(account.find(1), None) #Verify the account is gone
```

testDelete test function

The testDelete function is simply done by first creating an account from a random ACCOUNT_DATA account as in previous tests. Next, I verified that the account existed by doing a find for the account id and verifying that it did not return None. I call the account.delete() function, retry the account.find method, and assert that the find returned value should be equal to none. This verifies that the account was created then no longer existing.

Below is the proof of 100% test results

```
blake@LAPTOP-2SL71IQK MINGW64 /d/School/Spring 2024/CS 472/TestingLab/test_coverage (main)
$ nosetests

Test Account Model
- Update item in database
- Test creating multiple Accounts
- Test Account creation using known data
- Create dictionary with values and test reverse of to_dict functionality
- Test the representation of an account
- Test account to dict
- Update item in database

Name                  Stmts   Miss  Cover   Missing
---------------------------------------------------
models\__init__.py        7      0   100%
models\account.py        40      0   100%
---------------------------------------------------
TOTAL                    47      0   100%
----------------------------------------------------------------------
Ran 7 tests in 0.850s

OK
```

Task 5

My final results for Task 5 are below

```
blake@LAPTOP-2SL71IQK MINGW64 /d/School/Spring 2024/CS 472/TestingLab/tdd (main)
$ nosetests

Counter tests
- It should create a counter
- It should return an error for duplicates
- Tests updating counters

Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
src\counter.py       16      0   100%
src\status.py         6      0   100%
-----------------------------------------------
TOTAL                22      0   100%
----------------------------------------------------------------------
Ran 3 tests in 0.434s

OK

(tddVenv)
```

Successfully running the tests in Task 5 required the red/green/refactoring of several components in the testing and code environment.

My code for the counter test is below:

```python
# we need to import the file that contains the status codes
from src import status

class CounterTest(TestCase):
    """Counter tests"""
    def setUp(self):
        self.client = app.test_client()

    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)

    def test_duplicate_a_counter(self):
        """It should return an error for duplicates"""
        result = self.client.post('/counters/bar')
        self.assertEqual(result.status_code, status.HTTP_201_CREATED)
        result = self.client.post('/counters/bar')
        self.assertEqual(result.status_code, status.HTTP_409_CONFLICT)

    def test_update_a_counter(self):
        """Tests updating counters"""
        result = self.client.post('/counters/update')
        self.assertEqual(result.status_code, status.HTTP_201_CREATED)
        test_counter = result.json['update']
        result = self.client.put('/counters/update')
        self.assertEqual(result.status_code, status.HTTP_200_OK)
        self.assertGreater(result.json['update'], test_counter)
```

My code for the counter implementation:

```python
# we need to import the file that contains the status codes
from src import status
from flask import Flask

app = Flask(__name__)

COUNTERS = {}

# We will use the app decorator and create a route called slash counters.
# specify the variable in route <name>
# let Flask know that the only methods that is allowed to called
# on this function is "POST".
@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

@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    COUNTERS[name] = COUNTERS[name]+1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

Key Points

The three tests for this Flask app needed to return the proper HTTP results. Tests create_a_counter, duplicate_a_counter, and update_a_counter must return 200 or 201 results depending on their goals.

Test create_a_counter first experienced a **Red** testing output when Flask was improperly routed. This means the endpoint did not exist that create_a_counter needed. It was trying to send a post request to the /counter endpoint of the app, which had not been configured. By refactoring the Flask app, adding a wrapper to the app and routing the post request to the create_counter method, we could create the counter with the name as defined by the post endpoint /counter/{name}, and return the proper message. After routing to /counter/{name} and returning a 201 code, the testing results were **Green**.

Test duplicate_a_counter caused a **Red** testing case due to the improper handling of duplicate post requests in the Flask app. It's not proper to have multiple counters of the same name, as that causes issues, so the /counter/{name} post request was refactored to add a branch checking for the existence of the counter already. If the counter exists, we return a 409 CONFLICT result instead of the 201 CREATED result we started with. By applying this to the code, we receive a **Green** test result.

Finally, the update_a_counter required the creation of a new Flask endpoint to properly abide by REST API standards. Our testing sends a put request to our Flask app, which causes a **Red** testing case. In order to fix this, we refactor our Flask app to implement a new wrapper for a put method on the same endpoint. The testing code is written to create a new counter and update the counter by one using the new put endpoint. The assertions following the update check that the counter was increased by 1, and give us a **Green** test result.