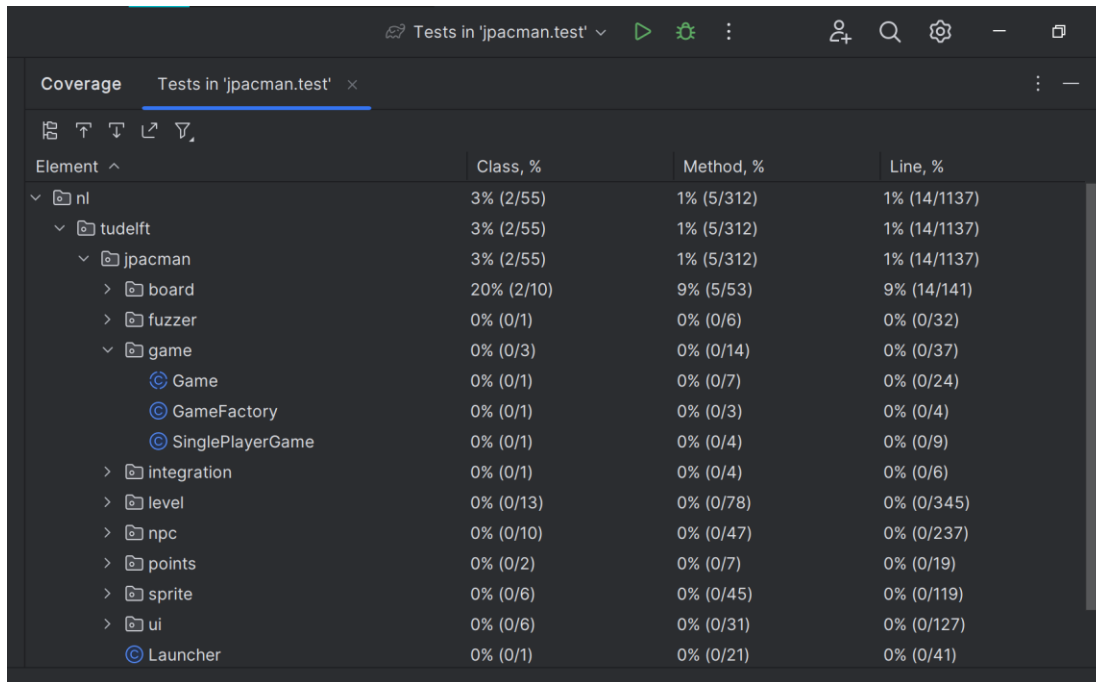


CS 472 Testing Report

Task 2.1: Method Testing

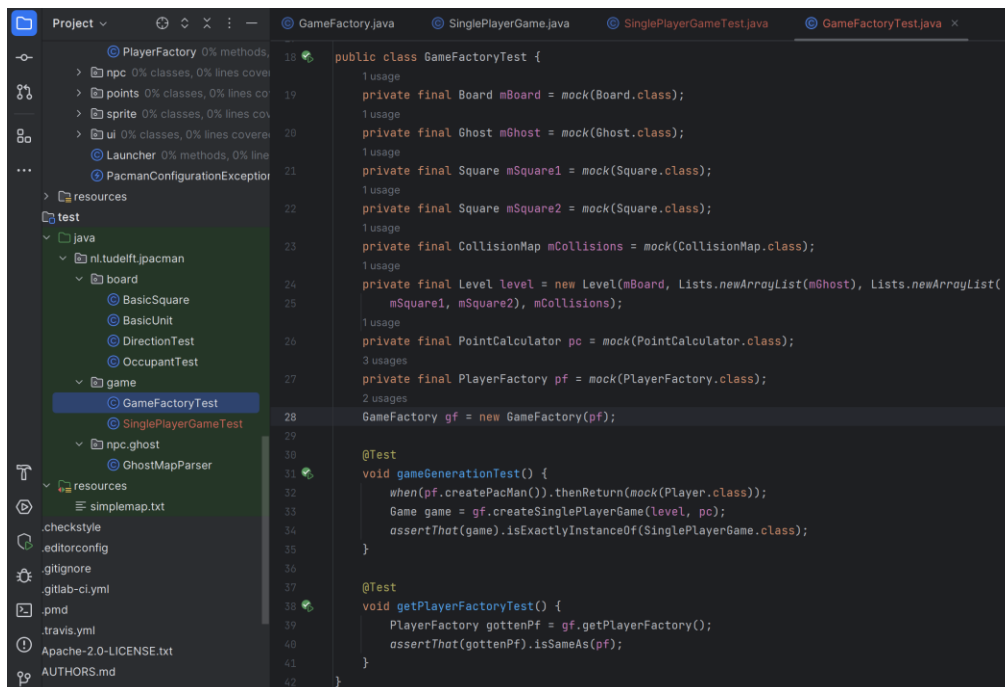
I have decided to do most of the testing for the game/ folder file. These tests will include all the GameFactory, SinglePlayerGame, and Game files. Here is an image of how the initial test coverage looks before testing the files.



The screenshot shows the 'Coverage' window in IntelliJ IDEA for the test suite 'jpacman.test'. The table displays coverage percentages for classes, methods, and lines across various elements in the project.

Element	Class, %	Method, %	Line, %
nl	3% (2/55)	1% (5/312)	1% (14/1137)
tudelft	3% (2/55)	1% (5/312)	1% (14/1137)
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)
Game	0% (0/1)	0% (0/7)	0% (0/24)
GameFactory	0% (0/1)	0% (0/3)	0% (0/4)
SinglePlayerGame	0% (0/1)	0% (0/4)	0% (0/9)
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)

GameFactory File Code:

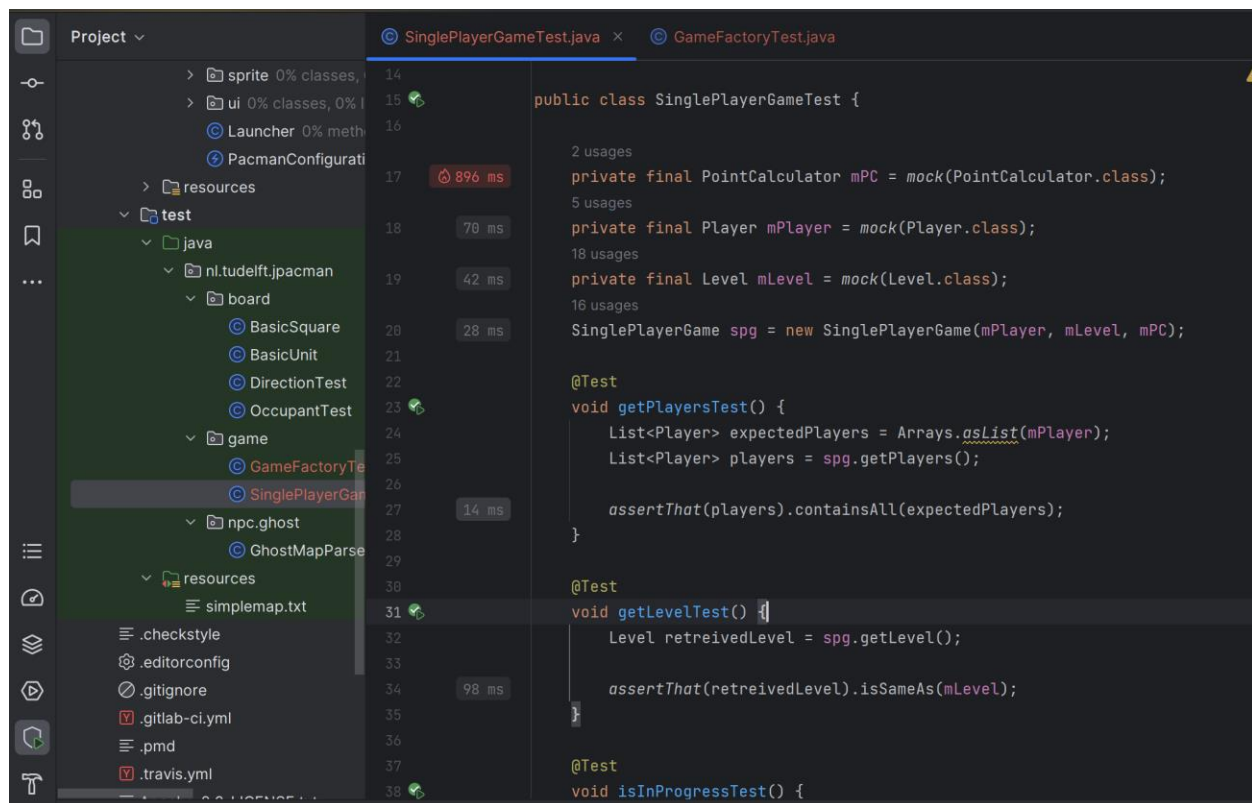


```
public class GameFactoryTest {
    1 usage
    private final Board mBoard = mock(Board.class);
    19
    1 usage
    private final Ghost mGhost = mock(Ghost.class);
    20
    1 usage
    private final Square mSquare1 = mock(Square.class);
    21
    1 usage
    private final Square mSquare2 = mock(Square.class);
    22
    1 usage
    private final CollisionMap mCollisions = mock(CollisionMap.class);
    23
    1 usage
    private final Level level = new Level(mBoard, Lists.newArrayList(mGhost), Lists.newArrayList(
    mSquare1, mSquare2), mCollisions);
    24
    1 usage
    private final PointCalculator pc = mock(PointCalculator.class);
    25
    3 usages
    private final PlayerFactory pf = mock(PlayerFactory.class);
    26
    2 usages
    27
    28 GameFactory gf = new GameFactory(pf);
    29
    30
    31
    @Test
    void gameGenerationTest() {
    32
    when(pf.createPacMan()).thenReturn(mock(Player.class));
    33
    Game game = gf.createSinglePlayerGame(level, pc);
    34
    assertThat(game).isExactlyInstanceOf(SinglePlayerGame.class);
    35
    }
    36
    37
    @Test
    void getPlayerFactoryTest() {
    38
    PlayerFactory gottenPf = gf.getPlayerFactory();
    39
    assertThat(gottenPf).isSameAs(pf);
    40
    }
    41
    42
}
```

There are only 2 methods to test for the GameFactory file excluding the constructor. The first method tested is the “createSinglePlayerGame” method. This method returns an instance of a SinglePlayerGame after providing the necessary parameters. The level parameter is not mocked but is instead made up of mocked elements, the PointCalculator is mocked, and the “createPacMan” method inside of PlayerFactory is stubbed and returns a mocked player class to avoid any possible null issues associated with needing to fake the data. To test we just make sure the return value is the correct type.

The second method is getPlayerFactory, which is very simple, and it just ensures that the playerFactory that the method retrieves is the same one that we used to create GameFactory with.

SinglePlayerGame File Code:



```
14
15
16
17 896 ms
18 70 ms
19 42 ms
20 28 ms
21
22
23
24
25
26
27 14 ms
28
29
30
31
32
33
34 98 ms
35
36
37
38

public class SinglePlayerGameTest {

    2 usages
    private final PointCalculator mPC = mock(PointCalculator.class);
    5 usages
    private final Player mPlayer = mock(Player.class);
    18 usages
    private final Level mLevel = mock(Level.class);
    16 usages
    SinglePlayerGame spg = new SinglePlayerGame(mPlayer, mLevel, mPC);

    @Test
    void getPlayersTest() {
        List<Player> expectedPlayers = Arrays.asList(mPlayer);
        List<Player> players = spg.getPlayers();

        assertThat(players).containsAll(expectedPlayers);
    }

    @Test
    void getLevelTest() {
        Level retrievedLevel = spg.getLevel();

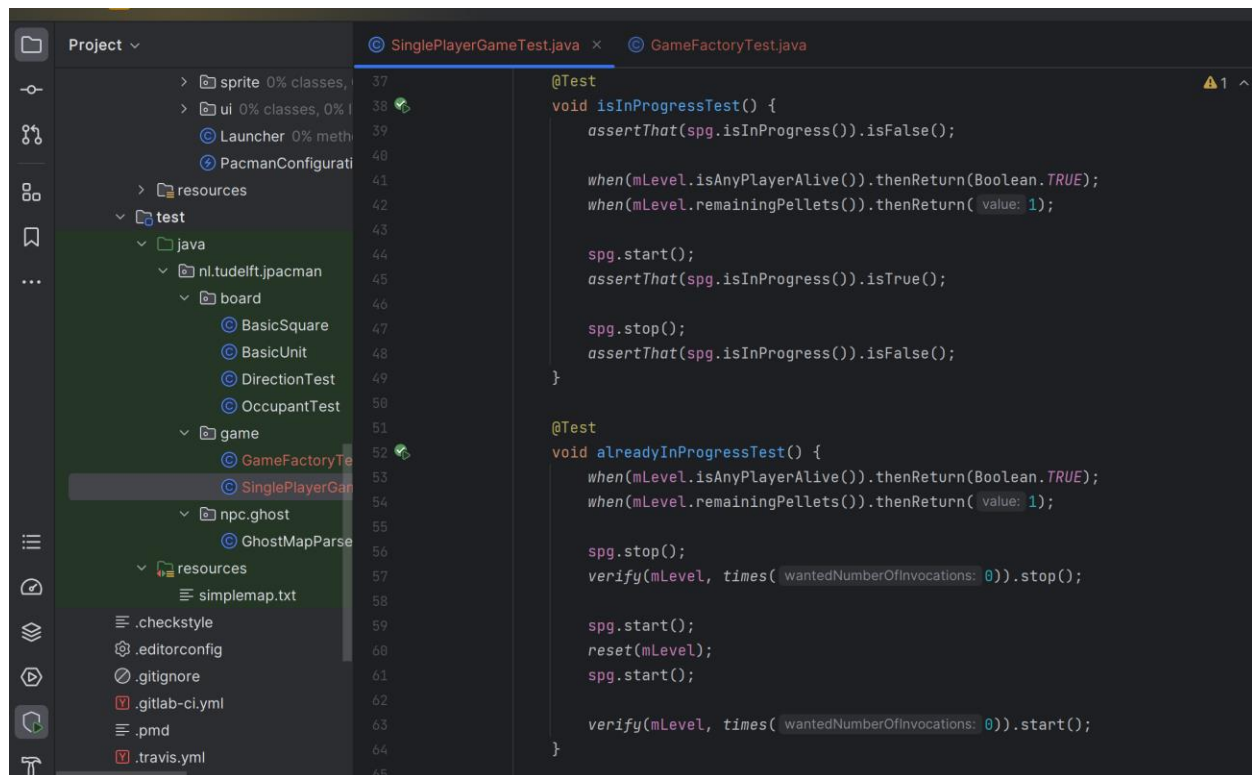
        assertThat(retrievedLevel).isSameAs(mLevel);
    }

    @Test
    void isInProgressTest() {
```

This file covers tests for both SinglePlayerGame class and the Game class. The reason for this is because we need an extended class in order to test the Game class.

The first method is getPlayers, which provides a list of players. So, to test this I created a SinglePlayerGame object with the 3 parameters mocked. Then to ensure that the data is the same and not the object, I created an expectedPlayer list that only has the player we used to make the game and tested what the method retrieves with the data that should be expected.

The second method is easier, as it just retrieves the level. In this case we want to make sure it is the same object, so we test to make sure that it is the same level object as what we used to create it the SinglePlayerGame.



The next test is the `isInProgressTest` which makes sure that the `inProgress` Boolean changes after starting and stopping. To test this, we need to ensure that the mocked level methods return to make the if condition true. Then we can call to start and test that `inProgress` is true and then call stop and test that `inProgress` is false.

The following test `alreadyInProgressTest` is similar but the opposite direction. We first make sure that the mocked level methods will make the start condition true. Then we instead called the stop method. Since we have not started then this will fall in the condition that immediately returns from the stop function. To test this we can verify that the `level.stop()` function is not invoked, since it is only invoked when it can stop. The same is true for the start condition. We first start it and then reset the mocked level to ensure that we can count the invocations properly. Then we call start again and ensure that the `level.start()` method is not called.



```
63         verify(mLevel, times(wantedNumberOfInvocations: 0)).start();
64     }
65
66     @Test
67     void moveTest() {
68         Direction north = Direction.valueOf(name: "NORTH");
69         when(mLevel.isAnyPlayerAlive()).thenReturn(Boolean.TRUE);
70         when(mLevel.remainingPellets()).thenReturn(value: 1);
71
72         spg.start();
73         spg.move(mPlayer, north);
74
75         verify(mLevel, times(wantedNumberOfInvocations: 1)).move(mPlayer, north);
76         verify(mPC, times(wantedNumberOfInvocations: 1)).pacmanMoved(mPlayer, north);
77     }
78
79     @Test
80     void levelWonTest() {
```

The next test is the `moveTest`. This one is simple. We just need to first start and then call the move function with the specified parameters. To test it we verify that the mocked level and mocked point counter call the correct methods with the correct parameters.

```
@Test
void levelWonTest() {
    when(mLevel.isAnyPlayerAlive()).thenReturn(Boolean.TRUE);
    when(mLevel.remainingPellets()).thenReturn(value: 1);
    spg.start();

    spg.levelWon();
    verify(mLevel, times(wantedNumberOfInvocations: 1)).stop();
}

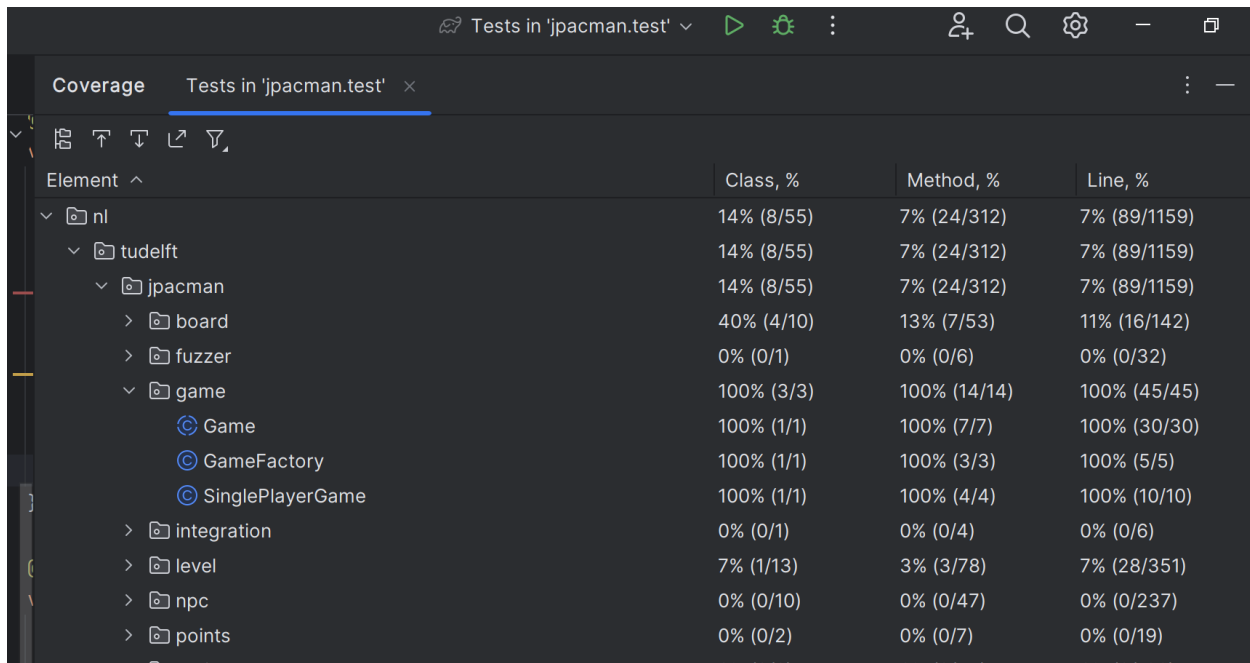
@Test
void levelLostTest() {
    when(mLevel.isAnyPlayerAlive()).thenReturn(Boolean.TRUE);
    when(mLevel.remainingPellets()).thenReturn(value: 1);
    spg.start();

    spg.levelLost();
    verify(mLevel, times(wantedNumberOfInvocations: 1)).stop();
}
```

The last 2 tests, levelWonTest and levelLostTest, are both essentially the same just for 2 different methods. We first need to start the game and then call either levelLost or levelWon. Both methods call the class's stop method so we just make sure that the level.stop() method is called. We cannot mock the Game class's stop method so instead we use the level's stop method since we can mock that class.

That is the end of all the tests for SinglePlayerGame and Game.

Here is the coverage after adding the above tests:



The screenshot shows the 'Coverage' window in IntelliJ IDEA for the test suite 'Tests in 'jpacman.test''. The window displays a tree view of the project structure on the left and a table of coverage data on the right. The table has four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The 'Element' column shows a hierarchy starting from 'nl' down to 'game' and its sub-classes. The 'Class, %', 'Method, %', and 'Line, %' columns show the percentage of coverage for each element, along with the number of classes, methods, and lines covered out of the total.

Element	Class, %	Method, %	Line, %
nl	14% (8/55)	7% (24/312)	7% (89/1159)
tudelft	14% (8/55)	7% (24/312)	7% (89/1159)
jpacman	14% (8/55)	7% (24/312)	7% (89/1159)
board	40% (4/10)	13% (7/53)	11% (16/142)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	100% (3/3)	100% (14/14)	100% (45/45)
Game	100% (1/1)	100% (7/7)	100% (30/30)
GameFactory	100% (1/1)	100% (3/3)	100% (5/5)
SinglePlayerGame	100% (1/1)	100% (4/4)	100% (10/10)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	7% (1/13)	3% (3/78)	7% (28/351)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)






As we can see every line has been covered for GameFactory, SinglePlayerGame, and Game.

Task 3: JaCoCo Report

- Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?
 - Not entirely. The report on JaCoCo looks like this:

jpacman > nl.tudelft.jpacman.game

nl.tudelft.jpacman.game

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
SinglePlayerGame		77%		50%	5	9	0	10	0	4	0	1
Game		100%		80%	2	12	0	30	0	7	0	1
GameFactory		100%	n/a	n/a	0	3	0	5	0	3	0	1
Total	10 of 154	93%	7 of 20	65%	7	24	0	45	0	14	0	3

It does show full coverage on GameFactory but SinglePlayerGame is a massively different story and Game has a few missed branches. Though it seems to be because JaCoCo takes asserts into a much more serious account as well as branches. Here is the file regarding SinglePlayerFactory:

```
15.  */
16. public class SinglePlayerGame extends Game {
17.
18.     /**
19.      * The player of this game.
20.      */
21.     private final Player player;
22.
23.     /**
24.      * The level of this game.
25.      */
26.     private final Level level;
27.
28.     /**
29.      * Create a new single player game for the provided level and player.
30.      *
31.      * @param player
32.      *         The player.
33.      * @param level
34.      *         The level.
35.      * @param pointCalculator
36.      *         The way to calculate points upon collisions.
37.      */
38.     protected SinglePlayerGame(Player player, Level level, PointCalculator pointCalculator) {
39.         super(pointCalculator);
40.
41.         assert player != null;
42.         assert level != null;
43.
44.         this.player = player;
45.         this.level = level;
46.         this.level.registerPlayer(player);
47.     }
48.
49.     @Override
50.     public List<Player> getPlayers() {
51.         return ImmutableList.of(player);
52.     }
53.
54.     @Override
55.     public Level getLevel() {
56.         return level;
57.     }
58. }
```

Here is the file regarding Game:

```
37.     *
38.     * The way to calculate points upon collisions.
39.     */
40.     protected Game(PointCalculator pointCalculator) {
41.         this.pointCalculator = pointCalculator;
42.         inProgress = false;
43.     }
44.
45.     /**
46.     * Starts or resumes the game.
47.     */
48.     public void start() {
49.         synchronized (progressLock) {
50.             if (isInProgress()) {
51.                 return;
52.             }
53.             if (getLevel().isAnyPlayerAlive() && getLevel().remainingPellets() > 0) {
54.                 inProgress = true;
55.                 getLevel().addObserver(this);
56.                 getLevel().start();
57.             }
58.         }
59.     }
60.
61.     /**
62.     * Pauses the game.
63.     */
64.     public void stop() {
65.         synchronized (progressLock) {
66.             if (!isInProgress()) {
67.                 return;
68.             }
69.             inProgress = false;
70.             getLevel().stop();
71.         }
72.     }
73. }
```

Clearly JaCoCo checks asserts more thoroughly and wants test for them when they fail. It also seems to want tests for if statement conditions failing. Because of these reasons it has discrepancies from the IntelliJ testing.

2. Did you find helpful the source code visualization from JaCoCo on uncovered branches?
 - a. It is very helpful to see uncovered branches and very important. IntelliJ showed lines that were tested but not branches.

3. Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?
 - a. I like both, IntelliJ is easy to use because it's built in and JaCoCo shows more details. However, if I had to pick, I would choose JaCoCo's report because it shows more details. When testing the ease of use is a nice factor but the overall goal is to have as much coverage as possible, so JaCoCo's reporting and being more detailed with branches is more important than IntelliJ's ease of use. Testing branches are important and even though in my tests the asserts don't matter to test as much since the code would error if they failed, it could be important in other situations.

Task 4: Python Code Coverage

This section is all about the code I used for the python code coverage section. This encompasses 4 methods with 5 tests.

From_Dict Test:

```
new *
def test_from_dict(self):
    """ Test from dict """
    account = Account()
    data = {
        "id": 720,
        "name": "testUser",
        "email": "testUser@test.com",
        "phone_number": "123987654",
        "disabled": False
    }
    account.from_dict(data)
    self.assertEqual(account.id, data["id"])
    self.assertEqual(account.name, data["name"])
    self.assertEqual(account.email, data["email"])
    self.assertEqual(account.phone_number, data["phone_number"])
    self.assertEqual(account.disabled, second: False)
```

This method is the entirety of the from_dict test. The method takes a dictionary and converts it into an account object with all the proper values populated. This test creates a new account which starts as empty and uses a new dictionary described a test user. It then calls from_dict with this test user dictionary and tests to ensure that the account values are the same as those provided by the dictionary.

Update Test (Success):

```
new *
def test_update_success(self):
    """ Test update success """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.id = 1
    account.create()

    new_phone_number = "987654321"

    account.phone_number = new_phone_number
    account.update()

    db_data = Account.find(1)

    self.assertEqual(db_data.id, second: 1)
    self.assertEqual(db_data.phone_number, new_phone_number)
```

This method tests 1 aspect of the update method. It tests when the method succeeds. We first take some random data from the premade ACCOUNT_DATA value and create an account starting with that data and commit it to the database. We then decide to change a value, in this case the phone number, and call to update the account. This will update the data inside of the database. To test this all happened correctly we query the database for the account and test to make sure it is the correct account we retrieved and that the data we wanted to update is updated.

Update Test (Failure):

```
new *
def test_update_fail(self):
    """ Test update fail """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    self.assertRaises(DataValidationError, account.update)
```

This test is the opposite of the previous test. Now we want the test to fail. So, we again make an account with the base data. The base data has no id on it, so when we call update, an error should be thrown. The assertRaise tests that the DataValidationError error is what occurs when account.update is called by this.

Test Delete:

```
new *
def test_delete(self):
    """ Test delete """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    account.delete()
    self.assertEqual(len(Account.all()), second: 0)
```

For the delete test, we know that when we create an account there will be a length of 1 of all the accounts (this comes from the create test). So, if we call delete right after a create we should get 0 accounts.

Test Find:

```
new *
def test_find(self):
    """ Test find """
    data = ACCOUNT_DATA[0]
    data2 = ACCOUNT_DATA[1]

    account = Account(**data)
    account2 = Account(**data2)
    account.id = 1
    account2.id = 2
    account.create()
    account2.create()

    db_data = Account.find(1)
    self.assertEqual(db_data.id, second: 1)
    self.assertEqual(db_data.name, data["name"])
    self.assertEqual(db_data.email, data["email"])
    self.assertEqual(db_data.phone_number, data["phone_number"])
```

The final test we need is for find. So, we take 2 different accounts, give them a separate id, and create them in the database. We then query for 1 of them using Account.find(id) and test to ensure that the data we retrieved from the database is the correct account data.

As a final overview here is running the full test suite:

```
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 0.182s

OK

Process finished with exit code 0
```

Task 5: TDD

For this section we had to implement tests first and then write the source code that would make those tests pass.

Put Request Test:

```
new *
def test_update_a_counter(self):
    """It should update a counter by +1"""
    post = self.client.post('/counters/coo')
    self.assertEqual(post.status_code, status.HTTP_201_CREATED)
    put = self.client.put('/counters/coo')
    self.assertEqual(put.status_code, status.HTTP_200_OK)
    self.assertEqual(put.json["coo"], post.json["coo"] + 1)
```

This is the test I wrote for put or update. I created a new counter and then to test that update (put) works, I test that the status code is correct and that the new response from the put endpoint is 1 more than when we created it.

```
@app.route(rule: '/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Updates a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_409_CONFLICT
    COUNTERS[name] = COUNTERS[name] + 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

This is the source code to make the test work. Now there is an extra thing I did here which is that we want to ensure that the name is inside of the COUNTERS dictionary we have, otherwise the counter doesn't exist, and we cannot update it. If the name does exist (which it should) then we just increment the counter by 1, return it and the OK status code.

Get Request Test:

```
new *
def test_read_a_counter(self):
    post = self.client.post('/counters/dar')
    self.assertEqual(post.status_code, status.HTTP_201_CREATED)

    res = self.client.get('/counters/read/dar')
    self.assertEqual(res.json["message"], second: "Counter: dar reads at: 0")
    self.assertEqual(res.status_code, status.HTTP_200_OK)

    self.client.put('/counters/dar')
    res = self.client.get('/counters/read/dar')
    self.assertEqual(res.json["message"], second: "Counter: dar reads at: 1")
    self.assertEqual(res.status_code, status.HTTP_200_OK)
```

This Get request didn't have a specification written down. The way I interpreted it is that we care to retrieve a message that reads the counter we asked for. So, to test this I created a new counter and then tried to get that counter. I wanted a message with what the counter reads at as well as the status code being 200. To make sure we get the right count for the counter I also made a put request which would increment the counter by 1 and made another get request. The message is the same except for the section that indicates what the counter is at. It should be at 0 when created and then at 1 after the put request.

```
32 @app.route(rule: '/counters/read/<name>', methods=['get'])
33 def get_counter(name):
34     """Gets a counter"""
35     app.logger.info(f"Request to get counter: {name}")
36     global COUNTERS
37     if name not in COUNTERS:
38         ⚡ return {"message": f"Counter {name} does not exist"}, status.HTTP_409_CONFLICT
39     return {"message": f"Counter: {name} reads at: {COUNTERS[name]}"}, status.HTTP_200_OK
```

This is the source code that makes the test pass. Again, if there is no counter then we cannot really get it. Otherwise, we can send a message that says the counter name and what it is currently at.

Summary of exceptions while writing tests:

The main exception I had while going back and forth with the test and source code was that the response object, I got from the endpoint would be in a way I did not expect. I initially thought it would be accessed like "response.data" but it is actually accessed by "response.json". Otherwise, all the tests broke without the source code and worked with the source code.

Joshua Holdridge

GitHub Fork: https://github.com/Linkaloo/CS472_Group1