CS472: Testing Report

Kyle Rainey
Fork repository links:
https://github.com/krain0/tdd
https://github.com/krain0/test_coverage
https://github.com/krain0/jpacman
https://github.com/krain0/CS472_LABS

Task 2

For task 2, the first test I added was for isAlive() in test/java/nl/tudelft/jpacman/npc/ghost/ClydeTest.java, following the step-by-step guide.

```
public class PlayerTest {
    @Test
    void testIsAlive() {
        Player p = (new PlayerFactory(new
            PacManSprites())).createPacMan();
        assertThat(p.isAlive()).isTrue();
    }
}
```

testIsAlive()

I added tests for two methods in the Clyde class: nextAiMove() and randomMove(). Creating these tests was more difficult because I had to create some simple maps so that I could see if Clyde was finding/not finding moves when it was intended to.

The test I added for randomMove() specifically tests that no move should be possible when Clyde is surrounded by inaccessible squares, so randomMove() should return null.

```
void testRandomMoveNull() {
    PacManSprites sprites = new PacManSprites();
    LevelFactory levelFactory = new LevelFactory(
        sprites,
        new GhostFactory(sprites),
        new DefaultPointCalculator());
    MapParser parser = new MapParser(
        levelFactory, new BoardFactory(sprites));
    Level level = parser.parseMap(
        new char[][]{{'#', '#', '#'},
                     {'#', ' ', '#'},
                     {'#', '#', '#'}});
    Board board = level.getBoard();
    Clyde c = new
Clyde(sprites.getGhostSprite(GhostColor.CYAN));
    c.occupy(board.squareAt(1,1));

    assertThat(c.randomMove()).isNull();
}
```

testRandomMoveNull()

For testNextAiMove(), I also had to add a player so that it's path-finding algorithm would function properly. This just tests if some move is found, not any particular move.

```
void testNextAiMove() {
    PacManSprites sprites = new PacManSprites();
    LevelFactory levelFactory = new LevelFactory(
      sprites,
      new GhostFactory(sprites),
      new DefaultPointCalculator());
    PlayerFactory playerFactory = (new PlayerFactory(new
        PacManSprites()));
    MapParser parser = new MapParser(levelFactory, new
        BoardFactory(sprites));
    Level level = parser.parseMap(new char[][]{{' '}, {'P'}});
    level.registerPlayer(playerFactory.createPacMan());
    Board board = level.getBoard();
    Clyde c = new
        Clyde(sprites.getGhostSprite(GhostColor.CYAN));

    c.occupy(board.squareAt(0,0));
    assertThat(c.nextAiMove().isPresent()).isTrue();
}
```

testNextAiMove()

I also added a test for the consumedAPellet method in DefaultPointCalculator. This just tests that when the player consumes a pellet with a certain value, the players score goes up by the appropriate value.

```
void consumedAPelletTest() {
    PacManSprites pacManSprites = new PacManSprites();
    Player player = new PlayerFactory(pacManSprites)
        .createPacMan();
    int originalScore = player.getScore();

    int pelletValue = 1;
    Pellet pellet = new Pellet(pelletValue,

pacManSprites.getPelletSprite());

    DefaultPointCalculator pointCalculator =
        new DefaultPointCalculator();
    pointCalculator.consumedAPellet(player, pellet);
    assertThat(player.getScore())
        .isEqualTo(originalScore + pelletValue);
}
```

consumedAPellet

Task 3

Are the coverage results from `JaCoCo` similar to the ones you got from `IntelliJ` in the last task? Why so or why not?
The coverage results I got from JaCoCo were different from the one's I got from IntelliJ because the JaCoCo report includes the coverage from the tests run in default-tests. So the JaCoCo coverage was higher.

Did you find helpful the source code visualization from `JaCoCo` on uncovered branches?
Yes, I found it very helpful. I liked how it was clearly marked when a branch was not covered, and that it showed how many cases were hit.

Which visualization did you prefer and why? `IntelliJ`'s coverage window or `JaCoCo`'s report?
There were parts of both that I liked. JaCoCo's coverage report was more thorough, but IntelliJ's was more convenient, because you can view it in your editor. For the highlighting of coverage in the source code, IntelliJ's was more informative, showing which cases of a branch were hit and how many times. But for JaCoCo, I liked how missed/hit branches were clearly shown with diamonds, while in IntelliJ, you had to hover over the gutter to see if it was counting branch or line coverage.

Task 4

For this task, I increased the test coverage of a python program to 100%. I only included snippets of the tests I added, not the ones provided.

```
def test_from_dict():
    result = dict();
    result["name"] = "John"
    result["email"] = "john@mail.com"
    result["phone_number"] = "1234567"

    account = Account()
    account.from_dict(result)

    assert account.name == result["name"]
    assert account.email == result["email"]
    assert account.phone_number == result["phone_number"]


def test_update_withID():
    # creating example info, with an ID
    newAccountInfo = dict()
    newAccountInfo["name"] = "Sam Sam"
    newAccountInfo["email"] = "sam@sam.sam"
    newAccountInfo["phone_number"] = "123.456.789"
    newAccountInfo["disabled"] = False
    newAccountInfo["id"] = 123

    account = Account(**newAccountInfo)
    userId = account.id
    account.create()

    newEmail = "sam_" + account.email

    account.email = newEmail
    account.update()

    newAccount = Account.find(userId)
    assert newAccount.email == newEmail
```

test_account.py added tests part 1

```
def test_update_noID():
    rand = randrange(0, len(ACCOUNT_DATA))  # Generate a random
index
    data = ACCOUNT_DATA[rand]  # get a random account
    account = Account(**data)
    try:
        # example data has no ID, so this should throw an
exception
        account.update()
        assert False
    except DataValidationError:
        assert True

def test_delete():
    # creating example info, with an ID
    newAccountInfo = dict()
    newAccountInfo["name"] = "Sam Sam"
    newAccountInfo["email"] = "sam@sam.sam"
    newAccountInfo["phone_number"] = "123.456.789"
    newAccountInfo["disabled"] = False
    newAccountInfo["id"] = 123

    account = Account(**newAccountInfo)
    userId = account.id
    account.create()
    # make sure it was created
    assert Account.find(userId) != None
    account.delete()
    # make sure it is gone
    assert Account.find(userId) == None
```

test_account.py added tests part 2

Task 5

Updating Counter

My first step was writing the test, so that I could actually get to a red phase. This took some time because I was not too familiar with this library, but once my test didn't have any errors, I was in a red phase.

During the red phase, my assertions about the status code were failing, so I added the update_counter function in counter.py.

After my status was ok, I was still failing the assertion because I was not updating my count, so I added that, and I passed the test.

To refactor, I deleted some extraneous prints and assertions that I had included to see how things work. I also added a check to see if the name actually exists in my list of counters.

```
    def test_update_a_counter(self, client):
        result = client.post('/counters/testingUpdate')
        assert result.status_code == status.HTTP_201_CREATED
        original_count = result.json["testingUpdate"]
        result = client.put('/counters/testingUpdate')
        assert result.status_code == status.HTTP_200_OK
        new_count = result.json["testingUpdate"]
        assert original_count + 1 == new_count
```

test_update_a_counter

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Create a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if not name 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
```

update_counter

Reading Counter

To get started on adding the ability to read the counter value, I again had to look at some documentation, and it took some time to get a test without runtime errors.

Once I did, I was in the red phase because I was getting the wrong status code. So I added the get_counter function in counter.py to add the appropriate functionality, and I was passing the test.

To refactor, again I removed extra check and assertions, and added a check for the existence of the counter before trying to access it in the dict.

```
    def test_get_counter(self, client):
        result = client.post('/counters/testingGet')
        assert result.status_code == status.HTTP_201_CREATED
        original_count = result.json["testingGet"]
        result = client.get('/counters/testingGet')
        assert result.status_code == status.HTTP_200_OK
        returned_count = result.json["testingGet"]
        assert original_count == returned_count
```

test_get_counter

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

get_counter