Elliot Wesoff
Prof. John Businge
CS 472, UNLV
February 5th, 2024

Software Testing Lab
https://github.com/elliotwesoff/Barbell

Task 2.1

The three methods I chose to test were:

1. src/main/java/nl/tudelft/jpacman/board/Board.getHeight

I tested this method by creating a BoardFactory instance and then using that to create a board instance with two squares in a single row. I then asserted that the height should be equal to 1.

```
@Test
void testGetHeight() {
    PacManSprites sprites = mock(PacManSprites.class);
    factory = new BoardFactory(sprites);
    var s1 = new BasicSquare();
    var s2 = new BasicSquare();
    var b = factory.createBoard(new Square[][]{{s1}, {s2}});
    assertThat(b.getHeight()).isEqualTo( expected: 1);
}
```

2. src/main/java/nl/tudelft/jpacman/board/Board.withinBorders

I tested this method by initializing a board in the same fashion as the previous test. I then made six different assertions to make sure that the method returns true when x and y coordinates within range are provided, and false when out-of-bounds coordinates are provided.

```java
@Test
void testWithinBorders() {
    PacManSprites sprites = mock(PacManSprites.class);
    factory = new BoardFactory(sprites);
    var s1 = new BasicSquare();
    var s2 = new BasicSquare();
    var b = factory.createBoard(new Square[][]{{s1}, {s2}});
    assertThat(b.withinBorders( x: -1,  y: 0)).isEqualTo( expected: false);
    assertThat(b.withinBorders( x: 0,  y: -1)).isEqualTo( expected: false);
    assertThat(b.withinBorders( x: 0,  y: 0)).isEqualTo( expected: true);
    assertThat(b.withinBorders( x: 1,  y: 0)).isEqualTo( expected: true);
    assertThat(b.withinBorders( x: 2,  y: 0)).isEqualTo( expected: false);
    assertThat(b.withinBorders( x: 0,  y: 1)).isEqualTo( expected: false);
}
```

3. src/main/java/nl/tudelft/jpacman/level/Level.isAnyPlayerAlive

I tested this method by first creating a new Level instance. I then asserted that Level.isAnyPlayerAlive() returns false, as there should be no players associated with a new Level instance.

```java
@Test
void testIsAnyPlayerAlive() {
    var level = new Level(board, Lists.newArrayList(ghost),
                          Lists.newArrayList(square1, square2),
                                             collisions);
    assertFalse(level.isAnyPlayerAlive());
}
```

Task 3

At first glance, it is difficult to empirically determine the amount of coverage on the Player class when comparing JaCoCo and IntelliJ's coverage results, as both coverages show different results for the same class. For example, IntelliJ shows 25% coverage for methods, or 2/8 methods covered. The JaCoCo results show that 4/8 methods are missing coverage, or 50% coverage for methods. IntelliJ shows 8/24 coverage for lines, while JaCoCo shows 4/24 lines missed, i.e. 20/24 coverage for lines. The results are completely inconsistent thus making it difficult to discern the actual state by simply comparing the two. However, in my single method to test Player.isAlive(), I am creating a Player instance (constructor method), as well as the Player.isAlive() method, so it appears that IntelliJ's count is more accurate than JaCoCo as far as direct code coverage is concerned.

I personally preferred the coverage details from JaCoCo over the IntelliJ coverage. It was better organized and shows much more detail on the tests covered.

Task 4

Creating the tests for the test_account.py file was straightforward, although getting the configuration exactly right for the database calls was tricky. A mock was required on one of my tests, while a specific ordering of create() and delete() calls were required for another. In order to get all of the lines in the update() function, I had to create two separate tests: one where an account has an ID in the database, and one where it does not. This is because the update() function contains a branch (if statement), so two calls to that function under different conditions were necessary to test it completely

```
def test_from_dict(self):
    """ Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account()
    account.from_dict(data)
    self.assertEqual(account.name, data["name"])
    self.assertEqual(account.email, data["email"])
    self.assertEqual(account.phone_number, data["phone_number"])
    self.assertEqual(account.disabled, data["disabled"])

def test_update1(self):
    """ Test account update """
    name = "steve"
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    account.name = name
    account.update()
    self.assertEqual(account.name, name)

def test_update2(self):
    """ Test account update without id raises an error """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    with self.assertRaises(DataValidationError):
        account.update()

def test_delete(self):
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    account.delete()
    account2 = Account.find(account.id)
    self.assertEqual(None, account2)

def test_find(self):
    """ Test find class method """
    data = ACCOUNT_DATA[self.rand] # get a random account
    query = MagicMock(name="query")
    Account.query = query
    Account.query.get.return_value = data
    result = Account.find(1)
    self.assertEqual(result["name"], data["name"])
    self.assertEqual(result["email"], data["email"])
    self.assertEqual(result["phone_number"], data["phone_number"])
    self.assertEqual(result["disabled"], data["disabled"])
```

Task 5

Update a counter

For the “red” phase of red/green/refactor I started by writing the unit test for updating a counter.

```
def test_update_a_counter(self):
    client = app.test_client()
    result = client.post('/counters/foo1')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    client.put('/counters/foo1')
    value = COUNTERS['foo1']
    self.assertEqual(1, value)
```

The test begins by posting to counter “foo1”. We ensure that the request was processed successfully, and that the counter was created by asserting its status code is 201. Then, the test makes a PUT request to “foo1”, indicating that the counter should be incremented. When the counter is created with the POST request, it should have been initialized to 0. After the PUT request completes successfully, we should expect for the value of the “foo1” counter to be 1, which is exactly what we are asserting in the last line of this unit test.

The original error produced from this unit test is:

```
======================================================================
FAIL: test_update_a_counter (test_counter.CounterTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/elliot/code/472/tdd/tests/test_counter.py", line 48, in test_update_a_counter
    self.assertEqual(1, value)
AssertionError: 1 != 0
```

Clearly, the actual value of the counter is not incremented, because there is no code written to implement this functionality. I implemented the update_counter function as such:

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

With this implementation, the unit test now passes, thus completing the “green” phase of the red/green/refactor phases. To refactor this, I added a conditional to ensure that the counter exists before trying to increment it, preventing server errors in these cases.

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    if name in COUNTERS:
        COUNTERS[name] += 1
        return {name: COUNTERS[name]}, status.HTTP_200_OK
    else:
        return {"Message":f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
```

Get a counter

For the “red” phase of red/green/refactor I started by writing the unit test for updating a counter.

```
def test_get_a_counter(self):
    client = app.test_client()
    result = client.post('/counters/foo2')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = client.get('/counters/foo2')
    data = json.loads(result.data)
    self.assertEqual(data['foo2'], 0)
```

The test begins by creating a counter “foo2” in the same fashion as the previous tests. I then make a GET request to retrieve the counter. The response is a JSON encoded string, so I use json.loads() to parse the string into a dict object. Since “foo2” is a newly created counter, we should expect its value to be 0, which is exactly what the assertion of this test does.

Since there is no implementation for this request, the error generated when running the tests is:

```
======================================================================
ERROR: test_get_a_counter (test_counter.CounterTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/elliot/code/472/tdd/tests/test_counter.py", line 55, in test_get_a_counter
    data = json.loads(result.data)
  File "/home/elliot/.asdf/installs/python/3.9.5/lib/python3.9/json/__init__.py", line 346, in loads
    return _default_decoder.decode(s)
  File "/home/elliot/.asdf/installs/python/3.9.5/lib/python3.9/json/decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/home/elliot/.asdf/installs/python/3.9.5/lib/python3.9/json/decoder.py", line 355, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

We get an error because the response isn't in a JSON format. Because we get an error from the test, the test fails, thus completing the "red" phase.

To make the test pass, I added this implementation:

```python
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name):
    """Get a counter"""
    app.logger.info(f"Request to get counter: {name}")
    return {name: COUNTERS[name]}, status.HTTP_200_OK
```

The test_get_counter unit test now passes, completing the "green" phase. To refactor, I added an if statement to prevent a KeyError from being thrown when a non existent key is requested:

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

The test now passes, thus completing the "refactor" phase of red/green/refactor for this unit test.