Unit Testing Report: johnxu21/jpacman

Within this repository, I have identified and created tests for three methods within the `Game` class that had 0% coverage. The following code has been implemented as a set of unit tests to improve coverage to 73%:

```java
package nl.tudelft.jpacman.game;

import nl.tudelft.jpacman.Launcher;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.AssertionsForClassTypes.assertThat;

public class GameTest {
    @Test
    void testStart() {
        Launcher l = new Launcher();
        Game g = l.makeGame();
        g.start();
        assertThat(g.isInProgress()).isEqualTo(true);
        g.start();
        assertThat(g.isInProgress()).isEqualTo(true);
        g.stop();
        g.start();
        assertThat(g.isInProgress()).isEqualTo(true);
        g.start();
        assertThat(g.isInProgress()).isEqualTo(true);
    }

    @Test
    void testStop() {
        Launcher l = new Launcher();
        Game g = l.makeGame();
        assertThat(g.isInProgress()).isEqualTo(false);
        g.start();
        g.stop();
        assertThat(g.isInProgress()).isEqualTo(false);
        g.stop();
        assertThat(g.isInProgress()).isEqualTo(false);
    }

    @Test
    void testIsInProgress() {
        Launcher l = new Launcher();
        Game g = l.makeGame();
        assertThat(g.isInProgress()).isEqualTo(false);
        g.start();
        assertThat(g.isInProgress()).isEqualTo(true);
        g.stop();
        assertThat(g.isInProgress()).isEqualTo(false);
    }
}
```

`testStart()`:

`Game.start()` is responsible for setting the state of the game to “In Progress”. There are a few outcomes that have been tested here:

1. The game has just been initialized.
2. The game has started already.
3. The game is resuming after pausing.
4. The game has started after being paused.

`testStop()`:

Similarly to `Game.start()`, `Game.stop()` is responsible for setting the state of the game to “Paused”. The outcomes that have been tested here are as follows:

1. The game is initially paused.
2. The game is pausing after starting.
3. The game is already paused.

`testIsInProgress()`:

`Game.isInProgress()` is responsible for reporting whether the game is in the “In Progress” or “Paused” states. As it is directly related to the other tested functions in this report, I thought it necessary to include it in this set of tests. The required behavior tested here is as follows:

1. On first initialization, the game is “Paused”.
2. When `Game.start()` is invoked, it should now be “In Progress”.
3. When `Game.stop()` is invoked, it should now be “Paused”.

Before Test Creation:

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)
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)

After Test Creation:

Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	56% (31/55)	41% (129/312)	36% (433/1186)
board	70% (7/10)	56% (30/53)	59% (86/144)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	100% (3/3)	64% (9/14)	77% (35/45)
Game	100% (1/1)	57% (4/7)	73% (22/30)
GameFactory	100% (1/1)	66% (2/3)	80% (4/5)
SinglePlayerGame	100% (1/1)	75% (3/4)	90% (9/10)

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman

Test Coverage w/ JaCoCo

Using the JaCoCo utility to analyze coverage among the entirety of the repository, I was able to uncover the following data after creating the tests previously listed:

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		58%	73	155	103	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		88%		62%	29	70	10	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		89%		65%	9	24	3	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,201 of 4,694	74%	289 of 637	54%	290	590	226	1,039	51	268	6	47

Created with JaCoCo 0.8.3.201901230119

The table above shows how well the current set of unit tests cover the source code within the repository. Out of the entire repository, the unit tests miss 26% of the total instructions in 46% of all branches. This shows that new unit tests should still be created as certain edge cases are likely missed because of the incomplete coverage.

The packages that should receive the most attention are:

- `nl.tudelft.jpacman.level`
- `nl.tudelft.jpacman.npc.ghost`
- `nl.tudelft.jpacman.ui`
- `nl.tudelft.jpacman.board`
- `nl.tudelft.jpacman.sprite`

These packages have significant branch occlusion in the unit testing in proportion to the amount of code written in them.

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman

Unit Testing Report: johnxu21/test_coverage

From the provided state of the repository, `nosetests` reports an initial coverage percentage of 77% with four tests. With four remaining methods in the `Account` class to create unit tests for, I was able to bring the total coverage up to 100%, with all eight tests passing.

```
Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data
- Test account delete from database
- Test account find in database
- Test account from dict
- Test the representation of an account
- Test account to dict
- Test account update in database

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 0.337s

OK
```

The methods for which I created unit tests for consist of the following:

`Account.from_dict():`

This method is responsible for initializing an `Account` instance using a dictionary which maps its key/value pairs to attribute/value pairs in the instance.

`Account.update():`

This method is responsible for updating an `Account` instance's data within the database.

`Account.delete():`

This method is responsible for removing the entry corresponding to an `Account` instance from the database.

`Account.find():`

This method is responsible for retrieving an `Account` instance from the database using a provided ID value for the account detail.

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman

Code Snippets

```python
def test_from_dict(self):
    """Test account from dict"""
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)

    dict_val = {
        "name": "Test Name",
        "email": "test_email@gmail.com",
        "phone_number": "1234567890",
        "disabled": False,
        "date_joined": "01/01/2024"
    }

    account.from_dict(dict_val)
    self.assertEqual(account.name, dict_val["name"])
    self.assertEqual(account.email, dict_val["email"])
    self.assertEqual(account.phone_number, dict_val["phone_number"])
    self.assertEqual(account.disabled, dict_val["disabled"])
    self.assertEqual(account.date_joined, dict_val["date_joined"])

def test_update(self):
    """Test account update in database"""
    data = ACCOUNT_DATA[self.rand] # get a random account
    account1 = Account(**data)
    account2 = Account(**data)
    account1.create()

    # update success; account1 in DB
    with self.assertLogs(logger, level="INFO"):
        account1.update()

    # update failure; account2 not in DB
    with self.assertLogs(logger, level="INFO"):
        with self.assertRaises(DataValidationError):
            account2.update()
```

```python
def test_delete(self):
    """Test account delete from database"""
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()

    with self.assertLogs(logger, level="INFO"):
        self.assertEqual(len(Account.all()), 1)
        account.delete()
        self.assertEqual(len(Account.all()), 0)

def test_find(self):
    """Test account find in database"""
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()

    with self.assertLogs(logger, level="INFO"):
        locate = Account.find(account.id)

        self.assertEqual(account.name, locate.name)
        self.assertEqual(account.email, locate.email)
        self.assertEqual(account.phone_number, locate.phone_number)
        self.assertEqual(account.disabled, locate.disabled)
        self.assertEqual(account.date_joined, locate.date_joined)
```

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman

Test Driven Development (TDD) Report: johnxu21/tdd

For this report, I followed the outline of Test Driven Development to create the required methods of a counter application using REST guidelines and Flask. After adding the provided unit tests and definition for `create_counter()`, I started creating tests for future methods, namely `update_counter()` and `read_counter()`.

Following the cycle outlined in TDD, I first created the unit test method `test_update_a_counter()` to outline the requirements for `update_counter()`. This placed my unit testing into the RED phase. Then, I created the definition of the method `update_counter()` and followed the unit test for its requirements. My unit testing was then in the PASS phase. Below is the unit test and method definition for `update_counter()`.

```python
def test_update_a_counter(self):
    """It should update a counter or return error for unknown name"""
    result = self.client.put('/counters/NULL')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

    result = self.client.post('/counters/foobar')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    base_value = result.get_json()['foobar']
    result = self.client.put('/counters/foobar')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    new_value = result.get_json()['foobar']
    self.assertEqual(base_value + 1, new_value)
```

```python
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update 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
```

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman

Moving on, I then created the unit test method `test_read_a_counter()` to outline requirements for `read_counter()`. This also placed my unit testing into the RED phase. Then, I created the definition of the method `read_counter()` and followed the unit test again for its requirements. Since no requirements were outlined for this method, I decided to use the following:

1. If the provided name in the app request does not match any counter, return a message and a `404_NOT_FOUND` response.
2. If the provided name does exist, return the counter with a `200_OK` response.

Below is the unit test and method definition for `read_counter()`.

```python
def test_read_a_counter(self):
    """It should read a counter or return error for unknown name"""
    result = self.client.get('/counters/NULL')
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)

    result = self.client.post('/counters/barfoo')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    base_value = result.get_json()['barfoo']

    # update counter 15 times
    for _ in range(15):
        result = self.client.put('/counters/barfoo')
        self.assertEqual(result.status_code, status.HTTP_200_OK)

    result = self.client.get('/counters/barfoo')
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    new_value = result.get_json()['barfoo']
    self.assertEqual(base_value + 15, new_value)
```

```python
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    """Read a counter"""
    app.logger.info(f"Request to read 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
```

With all four unit tests implemented, `nosetests` reports a 100% coverage of the counter methods.

Report Source: https://github.com/VyProductions/SeniorDesign/tree/master/jpacman