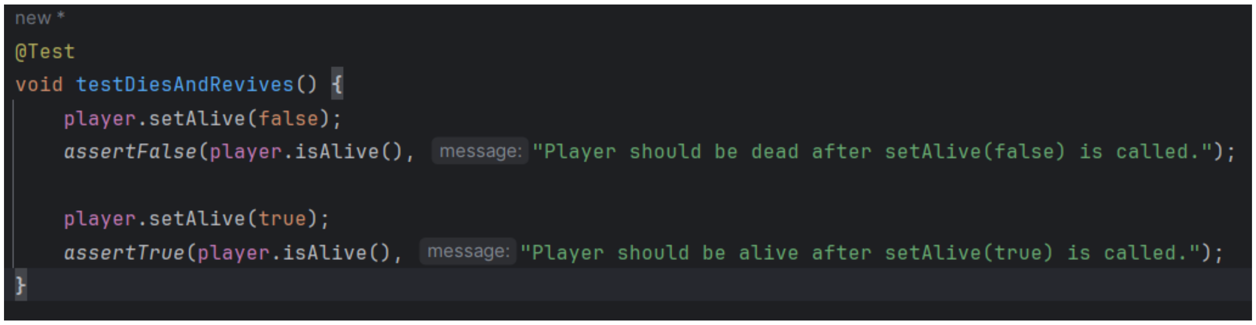
Report on Unit Testing and Coverage Analysis

Task 2.1 and Task 3: Unit Testing and Coverage with IntelliJ and JaCoCo

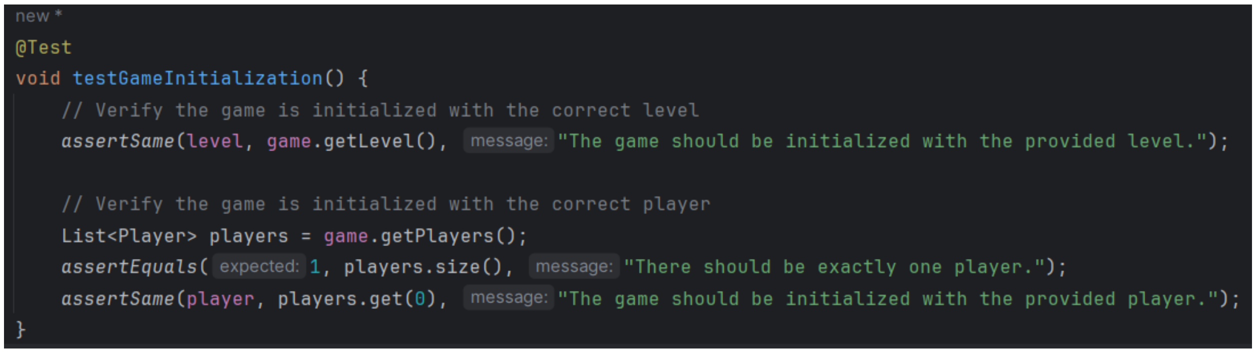
Test Case: testDiesAndRevives



```
new *
@Test
void testDiesAndRevives() {
    player.setAlive(false);
    assertFalse(player.isAlive(),  message: "Player should be dead after setAlive(false) is called.");

    player.setAlive(true);
    assertTrue(player.isAlive(),  message: "Player should be alive after setAlive(true) is called.");
}
```

This test verifies the functionality of the setAlive method for a player object. Initially, it asserts that after setting the player's alive status to false, the player is indeed considered dead (assertFalse is used to check that player.isAlive() returns false). Then, it tests that after setting the player's alive status back to true, the player is considered alive (assertTrue is used to ensure that player.isAlive() returns true). This test ensures that the setAlive method correctly updates and reflects the player's alive status.

Test Case: SinglePlayerGameTest



```
new *
@Test
void testGameInitialization() {
    // Verify the game is initialized with the correct level
    assertSame(level, game.getLevel(),  message: "The game should be initialized with the provided level.");

    // Verify the game is initialized with the correct player
    List<Player> players = game.getPlayers();
    assertEquals( expected: 1, players.size(),  message: "There should be exactly one player.");
    assertSame(player, players.get(0),  message: "The game should be initialized with the provided player.");
}
```

This test case focuses on verifying the initialization process of the **SinglePlayerGame** class, ensuring that the game is correctly set up with a given level and player. The test validates that the correct level and player are assigned during the game's initialization process.

Question 1: Coverage Results Comparison

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 between IntelliJ and JaCoCo differed significantly, with IntelliJ showing 100% coverage for **SinglePlayerGameTest**, while JaCoCo reported only 33%. This discrepancy can be attributed to how each tool calculates and interprets code coverage. IntelliJ's coverage metrics might focus more on the lines of code executed during the test, without distinguishing between different branches or conditions within those lines. On the other hand, JaCoCo provides a deeper analysis, considering paths and conditions within the code.

Question 2: Source Code Visualization

Did you find helpful the source code visualization from JaCoCo on uncovered branches?

Yes, the source code visualization feature in JaCoCo, which highlights uncovered branches directly in the source code, proved immensely helpful. It allows me to quickly identify which specific parts of their code were not executed during testing.

Question 3: Visualization Preference

Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?

While both IntelliJ's coverage window and JaCoCo's report offer valuable insights into test coverage, my preference leans towards JaCoCo's report for its detailed and comprehensive visualizations but favors IntelliJ for its ease of access and integration within the development environment.

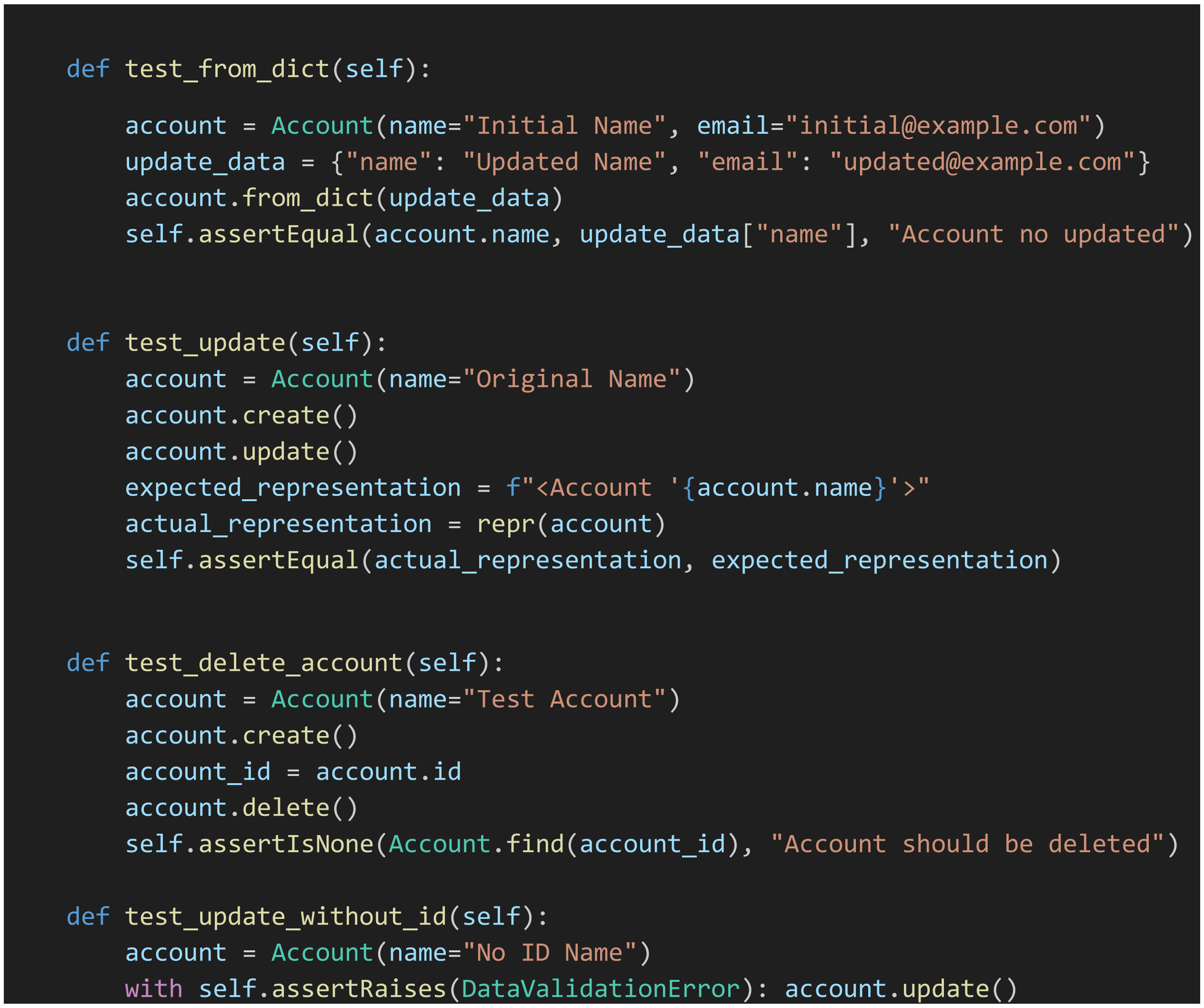
Task 4 Report: Achieving 100% Test Coverage with Python

Overview

This report documents the process of achieving 100% test coverage for a Flask application managing account entities, utilizing the test coverage repository "https://github.com/johnxu21/test_coverage". The task involved writing and refining tests to ensure coverage across all the Account model.

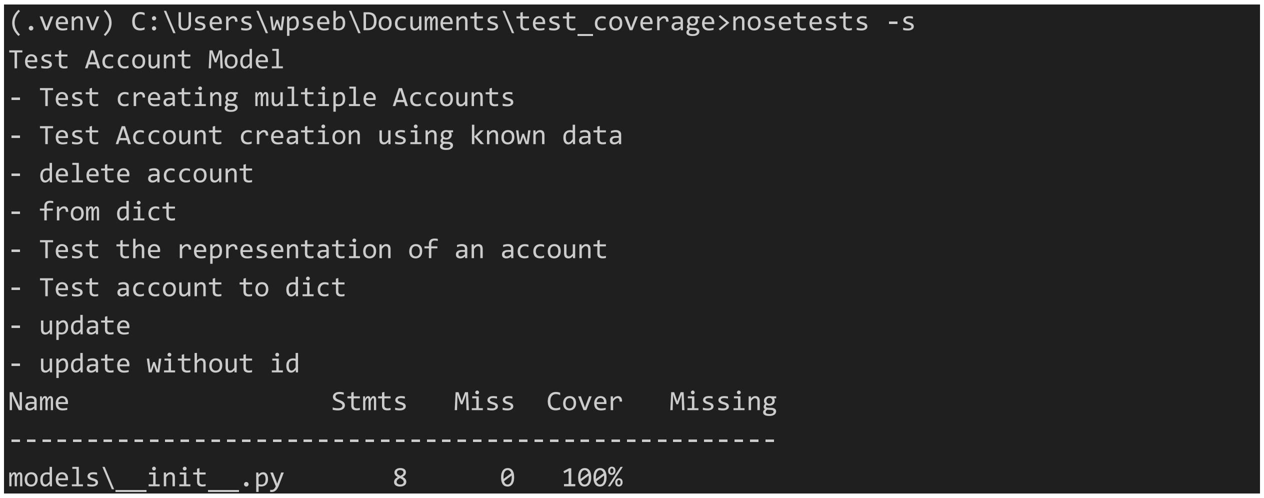
- Red Phase: Write tests that fail to highlight missing or incorrect functionality.
- Green Phase: Implement or correct the functionality to pass the tests.
- Refactor Phase: Refine the code to improve structure and readability without altering its behavior.

VS Code



```python
    def test_from_dict(self):

        account = Account(name="Initial Name", email="initial@example.com")
        update_data = {"name": "Updated Name", "email": "updated@example.com"}
        account.from_dict(update_data)
        self.assertEqual(account.name, update_data["name"], "Account no updated")


    def test_update(self):
        account = Account(name="Original Name")
        account.create()
        account.update()
        expected_representation = f"<Account '{account.name}'>"
        actual_representation = repr(account)
        self.assertEqual(actual_representation, expected_representation)


    def test_delete_account(self):
        account = Account(name="Test Account")
        account.create()
        account_id = account.id
        account.delete()
        self.assertIsNone(Account.find(account_id), "Account should be deleted")

    def test_update_without_id(self):
        account = Account(name="No ID Name")
        with self.assertRaises(DataValidationError): account.update()
```

Output For nosetests:



```
(.venv) C:\Users\wpseb\Documents\test_coverage>nosetests -s
Test Account Model
- Test creating multiple Accounts
- Test Account creation using known data
- delete account
- from dict
- Test the representation of an account
- Test account to dict
- update
- update without id
Name                 Stmts   Miss  Cover   Missing
---------------------------------------------------
models\__init__.py       8      0   100%
```

```
models\account.py        41       0    100%
--------------------------------------------------
TOTAL                    49       0    100%
----------------------------------------------------------------------
Ran 8 tests in 0.440s
OK
```

Final Results

Running nosetests -s confirmed 100% test coverage for the Account model, verifying the effectiveness of the tests implemented:

Task 5 Report: Flask Application Testing Report with Results

Overview

This report summarizes the process and outcomes of developing and testing a Flask application designed for managing counters through RESTful endpoints.

Implementation Details

The application provides three key functionalities:

- **Creating a Counter**: POST to **/counters/<name>** to create a new counter.
- **Updating a Counter**: PUT to **/counters/<name>** to increment an existing counter.
- **Reading a Counter**: GET to **/counters/<name>** to retrieve the current value of a counter.

Code Snippets

- **Create**:

```
@app.route('/counters/<name>', methods=['POST'])
def create_or_reject_counter(name):
    """Create a new counter if it does not exist. If it exists, reject with a
conflict error."""
    if name in COUNTERS:
        # Counter exists, return a conflict response
        return jsonify({"Message": f"Counter {name} already exists"}),
status.HTTP_409_CONFLICT
    else:
        # Create and return a new counter
        COUNTERS[name] = 1
        return jsonify({name: COUNTERS[name]}), status.HTTP_201_CREATED
```

- **Update**:

```
@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Increment an existing counter by 1. If it doesn't exist, return a not
found error."""
    if name not in COUNTERS:
        # Counter does not exist, return a not found response
        return jsonify({"Message": f"Counter {name} does not exist"}),
status.HTTP_404_NOT_FOUND
    else:
        # Update and return the counter
        COUNTERS[name] += 1
        return jsonify({name: COUNTERS[name]}), status.HTTP_200_OK
```

- **Read**:

```
@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    """Return the current value of an existing counter. If it doesn't exist,
return a not found error."""
    if name not in COUNTERS:
        # Counter does not exist, return a not found response
        return jsonify({"Message": f"Counter {name} does not exist"}),
status.HTTP_404_NOT_FOUND
    else:
        # Return the counter
        return jsonify({name: COUNTERS[name]}), status.HTTP_200_OK
```

Testing Approach

Testing was executed using the **nosetests** framework, targeting the correctness and reliability of each endpoint.

Challenges Encountered

- **Python Version Compatibility**: The project initially used Python 3.12, which lacked the **imp** module required by **nosetests**. This necessitated a downgrade to Python 3.9 and reinstallation of dependencies.
- **Flask Request Handling**: Gaining a thorough understanding of Flask's handling of multiple requests of the same request type was essential for effective API design.

Test Results

```
(.venv) C:\Users\wpseb\Documents\tdd>nosetests
Counter
```

```
- It should update a counter and return the new value

Name              Stmts   Miss  Cover   Missing
---------------------------------------------
src\counter.py       21      3    86%   13, 24, 35
src\status.py         6      0   100%
---------------------------------------------
TOTAL                27      3    89%
----------------------------------------------------------------------
Ran 1 test in 0.139s
OK
```