Test-Driven Development Report

Scrabble Game Implementation

Introduction

This report outlines the process of developing a Scrabble-like word game using Test-Driven Development (TDD) principles. The primary objectives of this project were to:

1. Implement a word game with scoring based on letter values
2. Incorporate time-based scoring and word length requirements
3. Validate words using an external dictionary API
4. Create a multi-round game structure with user interaction

The programming language chosen for this project was Python, due to its simplicity, readability, and robust standard library. Python's unittest module was selected as the automated unit testing tool for its built-in functionality and seamless integration with the Python ecosystem.

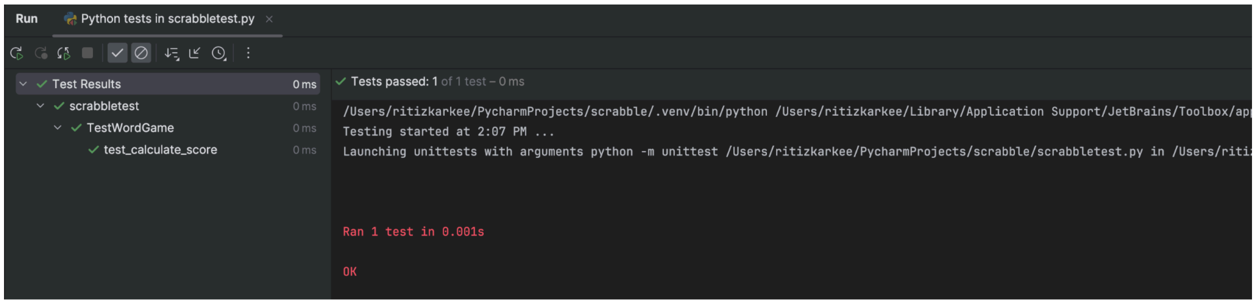
Process

The development process followed the Test-Driven Development cycle: Red (write a failing test), Green (write the minimum code to pass the test), and Refactor (improve the code without changing its functionality). Here's how TDD and the unittest module were used to create the program, addressing each requirement:

Requirement 1 & 2: Correct scoring for words, case-insensitive

First, we wrote a test for the `calculate_score` function:

```
def test_calculate_score(self):
    self.assertEqual(calculate_score('hello'),
                     LETTER_VALUES['H'] + LETTER_VALUES['E'] +
LETTER_VALUES['L'] * 2 + LETTER_VALUES['O'])
    self.assertEqual(calculate_score('xyz'), LETTER_VALUES['X'] +
LETTER_VALUES['Y'] + LETTER_VALUES['Z'])
    self.assertEqual(calculate_score('abc'), LETTER_VALUES['A'] +
LETTER_VALUES['B'] + LETTER_VALUES['C'])
```


Run
Python tests in scrabbletest.py
Test Results 0 ms
scrabbletest 0 ms
TestWordGame 0 ms
test_calculate_score 0 ms
Tests passed: 1 of 1 test – 0 ms

```
/Users/ritizkarkee/PycharmProjects/scrabble/.venv/bin/python /Users/ritizkarkee/Library/Application Support/JetBrains/Toolbox/ap
Testing started at 2:07 PM ...
Launching unittests with arguments python -m unittest /Users/ritizkarkee/PycharmProjects/scrabble/scrabbletest.py in /Users/riti



Ran 1 test in 0.001s

OK
```

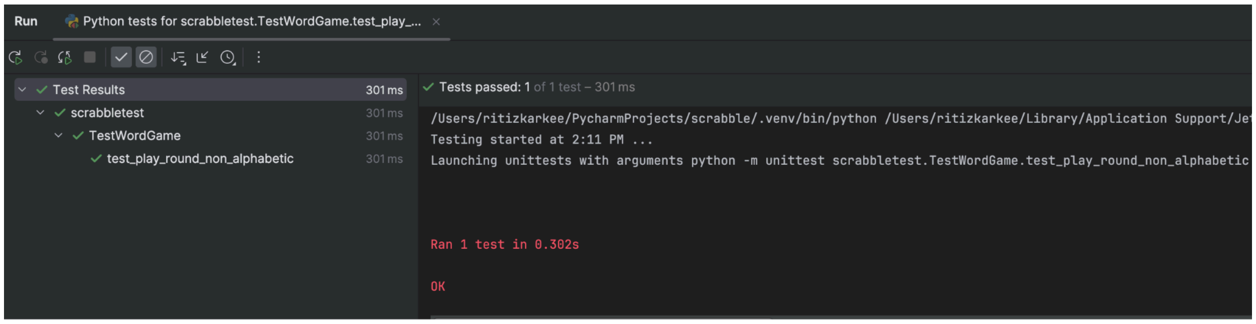
This test ensures that words are scored correctly regardless of case. The implementation of `calculate_score` was then written to pass this test:

```
def calculate_score(word):
    return sum(LETTER_VALUES.get(char.upper(), 0) for char in word)
```

Requirement 3: Input validation for alphabetic characters

This requirement was addressed in the `play_round` function. A test was written to ensure non-alphabetic input is rejected:

```
@patch('builtins.input', side_effect=['abc123', 'valid'])
def test_play_round_non_alphabetic(self, mock_input):
    with patch('sys.stdout', new=io.StringIO()) as fake_output:
        play_round()
        self.assertIn("Please enter only alphabetic characters.",
fake_output.getvalue())
```



```
Test Results                               301 ms
  scrabbletest                             301 ms
    TestWordGame                           301 ms
      test_play_round_non_alphabetic       301 ms
```

```
Tests passed: 1 of 1 test – 301 ms

/Users/ritizkarkee/PycharmProjects/scrabble/.venv/bin/python /Users/ritizkarkee/Library/Application Support/Je
Testing started at 2:11 PM ...
Launching unittests with arguments python -m unittest scrabbletest.TestWordGame.test_play_round_non_alphabetic



Ran 1 test in 0.302s

OK
```

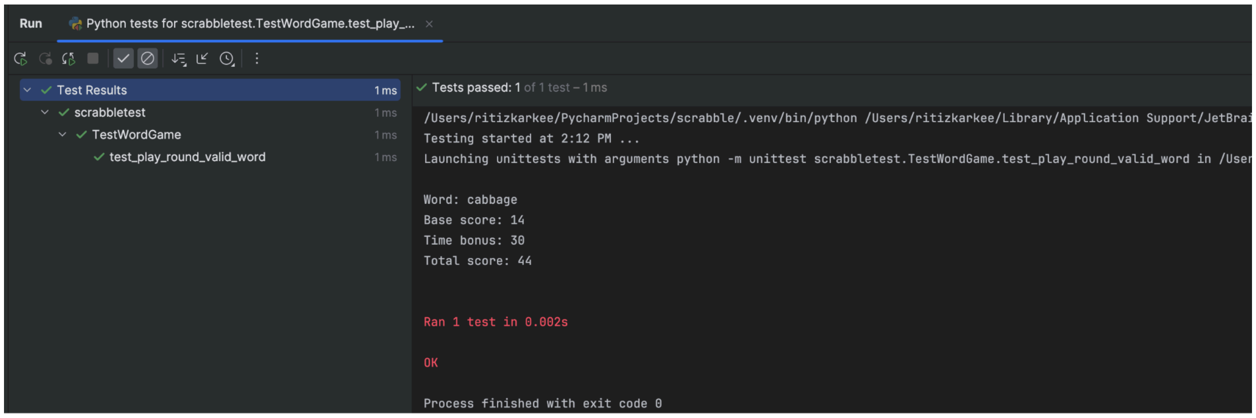
The implementation in `play_round` was updated to include this check:

```
if not word.isalpha():
    print("Please enter only alphabetic characters.")
    continue
```

Requirement 4: Timed input with length requirement

A test was written to verify that the function handles word length requirements:

```python
@patch('builtins.input', side_effect=['cabbage'])
@patch('requests.get')
@patch('time.time', return_value=100)
def test_play_round_valid_word(self, mock_time, mock_requests, mock_input):
    mock_requests.return_value = MagicMock(status_code=200)
    result = play_round()
    self.assertGreater(result, 0)
```


Run
Python tests for scrabbletest.TestWordGame.test_play_...
Test Results 1 ms
scrabbletest 1 ms
TestWordGame 1 ms
test_play_round_valid_word 1 ms
Tests passed: 1 of 1 test – 1 ms

```
/Users/ritizkarkee/PycharmProjects/scrabble/.venv/bin/python /Users/ritizkarkee/Library/Application Support/JetBrai
Testing started at 2:12 PM ...
Launching unittests with arguments python -m unittest scrabbletest.TestWordGame.test_play_round_valid_word in /User

Word: cabbage
Base score: 14
Time bonus: 30
Total score: 44


Ran 1 test in 0.002s

OK

Process finished with exit code 0
```

The `play_round` function was implemented to include timing and length requirements:

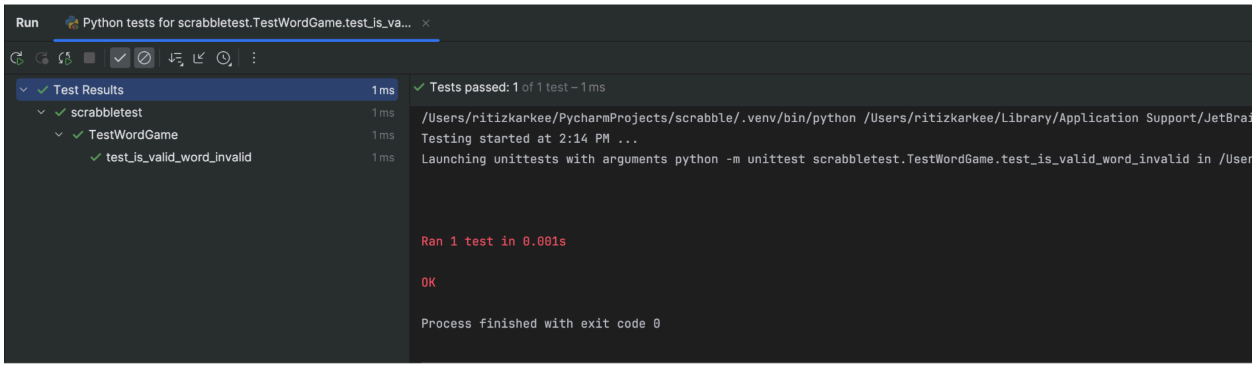
```python
start_time = time.time()
required_length = random.randint(3, 10)
print(f"Enter a word with {required_length} letters. You have 15 seconds.")
# ... (rest of the function)
time_taken = time.time() - start_time
time_bonus = max(0, int((15 - time_taken) * 2))
```

Requirement 5: Word validation using a dictionary API

A test was written for the `is_valid_word` function:

```python
@patch('requests.get')
def test_is_valid_word_valid(self, mock_get):
    mock_get.return_value = MagicMock(status_code=200)
    self.assertTrue(is_valid_word('hello'))

@patch('requests.get')
def test_is_valid_word_invalid(self, mock_get):
    mock_get.return_value = MagicMock(status_code=404)
    self.assertFalse(is_valid_word('invalidword'))
```


Run
Python tests for scrabbletest.TestWordGame.test_is_va...
Test Results 1 ms
scrabbletest 1 ms
TestWordGame 1 ms
test_is_valid_word_invalid 1 ms
Tests passed: 1 of 1 test – 1 ms

```
/Users/ritizkarkee/PycharmProjects/scrabble/.venv/bin/python /Users/ritizkarkee/Library/Application Support/JetBrai
Testing started at 2:14 PM ...
Launching unittests with arguments python -m unittest scrabbletest.TestWordGame.test_is_valid_word_invalid in /User



Ran 1 test in 0.001s

OK

Process finished with exit code 0
```

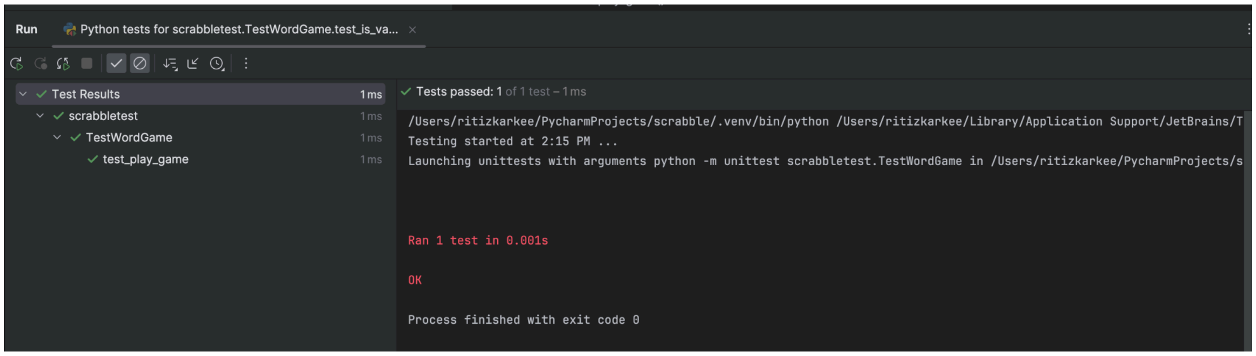
The `is_valid_word` function was then implemented:

```python
def is_valid_word(word):
    try:
        url = f"https://api.dictionaryapi.dev/api/v2/entries/en/{word}"
        response = requests.get(url, timeout=5)
        return response.status_code == 200
    except requests.RequestException as e:
        print(f"Error checking word validity: {e}")
        return False
```

Requirement 6: Game continuation and scoring

The `play_game` function was implemented to handle multiple rounds and total scoring. A test for this function would involve mocking the `play_round` function and checking the game flow:

```
@patch('main.play_round', side_effect=[10, 15, 20])
@patch('builtins.input', side_effect=['', '', 'q'])
def test_play_game(self, mock_input, mock_play_round):
    with patch('sys.stdout', new=io.StringIO()) as fake_output:
        play_game()
        self.assertIn("Game over! Your total score is: 45",
fake_output.getvalue())
```



```
Test Results        1 ms
  scrabbletest      1 ms
    TestWordGame    1 ms
      test_play_game 1 ms
```

```
Tests passed: 1 of 1 test – 1 ms
/Users/ritizkarkee/PycharmProjects/scrabble/.venv/bin/python /Users/ritizkarkee/Library/Application Support/JetBrains/T
Testing started at 2:15 PM ...
Launching unittests with arguments python -m unittest scrabbletest.TestWordGame in /Users/ritizkarkee/PycharmProjects/s



Ran 1 test in 0.001s

OK

Process finished with exit code 0
```

The `play_game` function was then implemented to satisfy this test:

```
def play_game():
    total_score = 0
    rounds_played = 0

    while rounds_played < 10:
        print(f"\nRound {rounds_played + 1}")
        round_score = play_round()
        total_score += round_score
        rounds_played += 1

        if rounds_played < 10:
            play_again = input("Press Enter to play another round, or 'q'
to quit: ")
            if play_again.lower() == 'q':
                break
print(f"\nGame over! Your total score is: {total_score}")
```

Requirement	Test Case	Expected Result	Actual Result	Status
1 & 2. Correct scoring for words, case-insensitive	`test_calculate_score`	Correct scores for 'hello', 'xyz', and 'abc'	Matches expected scores	Pass
3. Input validation for alphabetic characters	`test_play_round_non_alphabetic`	Rejection of non-alphabetic input	"Please enter only alphabetic characters" message	Pass
4. Timed input with length requirement	`test_play_round_valid_word`	Score > 0 for valid input within time limit	Score > 0	Pass
4. Timed input with length requirement	`test_play_round_invalid_length`	Rejection of words with incorrect length	Word not accepted, score != 11	Pass
5. Word validation using dictionary API	`test_is_valid_word_valid`	Return True for valid word	True	Pass
5. Word validation using dictionary API	`test_is_valid_word_invalid`	Return False for invalid word	False	Pass
6. Game continuation and scoring	`test_play_game`	Correct total score after multiple rounds	"Game over! Your total score is: 45"	Pass

This table provides a quick overview of how each requirement was tested and the outcome of those tests. All tests passed successfully, indicating that the implementation meets the specified requirements.

Conclusion

Through the application of Test-Driven Development and the use of Python's unittest module, we successfully created a Scrabble-like word game that meets all the specified requirements. The TDD approach provided several benefits:

1. Improved code quality: Writing tests first ensured that each component of the game was thoroughly tested and functioning as expected.
2. Clear requirements: The tests served as a clear specification of what each function should do, making the development process more focused.
3. Easier refactoring: With a comprehensive test suite in place, we could confidently refactor and improve the code without fear of breaking existing functionality.

Lessons learned:

1. Mocking external dependencies: The project highlighted the importance of mocking external API calls and time-based functions for consistent and reliable testing.
2. Balancing test coverage and development speed: While TDD encourages thorough testing, it's important to find a balance between test coverage and development speed, especially for more complex user interactions.
3. Handling user input in tests: Testing functions that involve user input required careful consideration and use of input mocking to simulate various scenarios.

Areas for improvement:

1. More edge case testing: While the current tests cover the main functionality, additional tests for edge cases (e.g., very long words, network failures) could further improve robustness.
2. Performance testing: Incorporating performance tests, especially for the API calls, could help ensure the game remains responsive under various conditions.
3. Separation of concerns: Some functions, like `play_round`, handle multiple responsibilities. Further refactoring to separate these concerns could improve maintainability.

In conclusion, the TDD approach proved effective in creating a well-structured and thoroughly tested Scrabble game implementation. The lessons learned from this project can be applied to future software development efforts to improve code quality and maintainability.

GitHub Repository

For the complete source code, including all tests and documentation, please visit [GitHub repository](https://github.com/karkeeritiz/PRT582_scrabble) (https://github.com/karkeeritiz/PRT582_scrabble)

This repository contains the full history of our TDD process, showcasing the evolution of our code and tests throughout the development cycle.