CS5031 - Software Engineering Practice

P1 - Wordle

Matriculation Number: 210025499

Computer Science
University of St Andrews

1 Implementation

1.1 Overview

Wordle is a word guessing game in which a user may use up to 6 guesses to guess a word chosen at random from a collection of 5 letter words. In this implementation, wordle refers to the randomly chosen word that the user will try to guess, and the WordleApp is the object representing the game state. Only guesses that are in a dictionary of 5 letter words will be accepted. In the original Wordle, now hosted on the New York Times[1], a guessed letter that does not appear in the wordle would be coloured grey. In this implementation, this colour has been adapted to red for easier viewing on the command line.

The Rules of the wordle game:

- You have up to 6 guesses to guess the Wordle
- A correct letter in the correct position turns green
- A correct letter in the incorrect position turns yellow
- An incorrect letter turns red
- Wordles are 5 letters long, so guesses must be 5 letters long

1.2 Design

To check a guess against the wordle, we first scan through both the wordle and the guess in parallel, marking any matching letters at the same positions as green. If we marked any, we can skip over these when trying to match other letters.

For the remaining letters in our guess, we check if they appear anywhere in the wordle, skipping over wordle characters we've already matched against. If they do appear somewhere in the wordle, then this letter in our guess should be marked yellow to indiciate it was found, but not in its equivalent position.

If any character in our guess hasn't been marked as green or yellow, then it can be marked grey to indicate that this letter does not appear in the wordle.

A user is prompted to enter a guess. They may enter "!quit" on any guess to exit. A guess will either be rejected, with an appropriate error message shown, or accepted, which will trigger the game board to be rerendered with coloured letters corresponding to the results of the guess. This process continues until a guess matches the wordle, in which case the final board is printed and a score is displayed, or until the user runs out of guesses, and the wordle is revealed.

2 Development Approach

2.1 Use of TDD and Version Control

To develop this project I used Git version control, Maven, and JUnit5.

For this project I used Test Driven Development with small and frequent iterations. I would think of use cases which could be translated to tests simply without being too reliant on class design and the structure of the implementation. This drove a bottom-up style approach, as the first tests were testing individual guesses against a wordle. By writing various tests for different guesses, wordles, and expected outputs, the high-level blueprint of the implementation became easier to see. I would write test cases which called not yet defined classes and their methods, knowing roughly what they should be expected to do. Then, for any methods that were required by a test, but not yet implemented in the source code, I would write stub methods, so that the methods could be called by a test and compile, but fail, as the implementation was not yet defined, and so was not passing the test.

At this point, I would push to the main branch of the git repository. Then, I would write the implementations for any methods which needed defining, and work on the implementation logic until the latest set of tests were all passing. Once all tests which previously which failed now passed, I would push these changes.

2.2 Refactoring

It would often be the case that while implementing a method, I realised that the method needed to do something else, and so the test that first referenced this method would have to change. One example of this is validation of a user's guess. I previously would return either true or false, and either reprompt for input or take the validated guess and proceed. I then realised that if a user's guess is invalid, the validation method doesn't give any information about the root cause of why the input is invalid, it just indicates that for some reason, the input was invalid. To capture information about why the input is not valid, I needed to refactor this method to use exceptions. This propagated changes into other methods which used this validation method. Fortunately, the methods were not tightly coupled, and these changes did not cascade too far into other parts of the code. I would then refactor the tests to fit the new design. As the purpose of the test had not changed (the use case from which the test was created) the name of the test remained the same, bu its implementation would need redefining. It was important that even though this change was prompted by working on the implementation, we must leave the implementation method as a stub until we refactor the test for the new design. I would then refactor the test, push the changes, and then begin to implement the stub method, with the new design in mind, and the tests amended.

3 Reflection

It was hard adjusting to the TDD approach as it requires you to write tests before design. However, by writing tests with stub implementation methods, I was able to approach the design in a bottom-up fashion, and ensure that each unit of code (often methods) performed as expected in isolation from other components. Using version control and refactoring via tests made the work flow efficient and manageable. The final product is extensible, and allows for modular additions of components, with ease of refactoring.

Word Count: 970 (including headers)

References

[1] Wordle

The New York Times

`https://www.nytimes.com/games/wordle/index.html`