Hangman Game Development Report

230008042

Hangman implementation

- Implementing the rules of the classic Hangman game.
    - Input and Feedback

    After each input by the user, if the guess is correct, the correct character will be filled in the word and displayed in the corresponding index bit, if a letter is repeated in a word, such as (p->apple),the letter will be displayed in all the index bits, e.g., ***pp***. Repeating a correctly entered letter will not count as an error.

```
Current state: _a_ia
Guessed letters: [a, v, g, i]
 +---++
 |   ||
 O   ||
 |   ||
     ||
     ||
=======
```

    If the user guesses incorrectly, an ASCII drawing of the hangman will show the 6 parts, head, torso, arms, legs, etc. Non-alphabetic inputs, such as numbers or special characters, will result in a failed attempt at input, but will not be considered guessing errors. If multiple letters are entered, only the first character is retained.

```
Current state: _____
Guessed letters: [v, g]
 +---++
 |   ||
 O   ||
 |   ||
     ||
     ||
=======
```

    After each entry, the history of letters entered by the user is displayed in a list.

    - Scoring

      The game starts with 120 points, incorrect inputs result in a deduction of 20 points, game fails with 0 points.

```
Game over! The word was: stare
 +---++
 |   ||
 O   ||
/|\  ||
/ \  ||
     ||
=======

Your score is 0
```

game fail

```
 +---++
 |   ||
     ||
     ||
     ||
     ||
=======
Guess a letter: o

Congratulations! You guessed the word: hundo

Your score is 120
```

game successful

Test-driven approach

The development approach followed TDD, breaking down the game functions into minimal components based on game rules, such as input methods for the Logic class and methods for acquiring game status, as well as output methods for the UI class. The writing of unit test cases is oriented towards these objects and their respective methods, with the test cases essentially covering all conditional branches.

The unit tests for the Logic class include the verification of GameLogic class properties (win/lose status, score, error count) under different states. A GameLogic instance is set as a stub[^1], with test cases including assertions on the game state after initialization, correct or incorrect letter input, input exceptions, and game success or failure.

```java
/**
 * Sets up the testing environment before each test. This includes initializing the game
 * logic and preparing a list of characters that are not in the secret word.
 */
@BeforeEach
void setUp() {
    gameLogic = new HangmanGameLogic();
    for (char ch = 'a'; ch <= 'z'; ch++) {
        errorInputAlphabet.add(ch);
    }
    for (Character cc : gameLogic.getSecretWord().toCharArray()) {
        errorInputAlphabet.remove(cc);
    }
}
```

Detection of exceptions thrown by abnormal user inputs was also implemented.[^1]

```java
@Test
void testNonAlphabeticInputThrowsException() {
    Executable action1 = () -> gameLogic.makeGuess( input: '1');
    Executable action2 = () -> gameLogic.makeGuess( input: '*');
    assertThrows(NonAlphabeticInputException.class, action1,  message: "Non-alphabetic input should throw NonAlphabeticInputException.");
    assertThrows(NonAlphabeticInputException.class, action2,  message: "Non-alphabetic input should throw NonAlphabeticInputException.");
}
```

The unit tests for the UI class adopt a mock approach to verify the UI outputs under different game states, including correct, incorrect, or exceptional input, and output assertions for game success or failure.

In the test case for input and output, a spy was created using `verify` to ensure that `makeGuess()` was called correctly[^2].

```java
@Test
public void testGameEndsAfterThreeWrongGuesses() {
    when(userInput.nextLine()).thenReturn( t: "a",  ...ts: "z", "b");
    when(gameLogic.isGameLost()).thenReturn( t: false,  ...ts: false, false, true);
    when(gameLogic.isGameWon()).thenReturn(false);
    when(gameLogic.getSecretWord()).thenReturn("apple");
    //ui = new HangmanUI(gameLogic, userInput);
    ui.play();


    verify(gameLogic, times( wantedNumberOfInvocations: 1)).makeGuess( input: 'a');
    verify(gameLogic, times( wantedNumberOfInvocations: 1)).makeGuess( input: 'z');
    verify(gameLogic, times( wantedNumberOfInvocations: 1)).makeGuess( input: 'b');
}
```

After completing the development of the unit test cases, the methods for the Logic and UI classes were implemented in turn, with the test cases transitioning from fail to success[^1].

```
[ERROR]   HangmanLogicTest.testNonAlphabeticInputThrowsException:75 Non-alphabetic input should throw NonAlphabeticInputException. ==> Expected hangman.exception.NonAlphabeticInputExcept
ion to be thrown, but nothing was thrown.
[ERROR]   HangmanLogicTest.testRepeatedGuess:120 Wrong guesses count should not change after repeating a guess. ==> expected: <1> but was: <2>
[ERROR]   HangmanLogicTest.testWinningGame:135 Game should be won when all letters are guessed ==> expected: <true> but was: <false>
[ERROR]   HangmanUITest.testGameLoss:111 expected: <true> but was: <false>
[ERROR]   HangmanUITest.testGameWin:96 expected: <true> but was: <false>
[ERROR]   HangmanUITest.testIncorrectGuess:68 Expected game over message in output. ==> expected: <true> but was: <false>
[ERROR]   HangmanUITest.testNonAlphabeticInput:82 Expected NonAlphabeticInputException detected output. ==> expected: <true> but was: <false>
[INFO]
[ERROR] Tests run: 12, Failures: 10, Errors: 0, Skipped: 0
```

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running HangmanLogicTest
[INFO]
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Refactoring approach throughout development

The design of the game followed OOP principles, combining game logic-related methods and attributes into the HangmanGameLogic class according to different functions[^3]. Methods within the HangmanGameLogic class are designed to minimize responsibility, each handling specific tasks such as submitting user input through makeGuess() and obtaining the current game state with getCurrentState().

Methods related to user input and output interactions are aggregated into the HangmanUI class. By defining the UserInput member variable of HangmanUI as an interface, polymorphism for input sources was achieved, such as ScannerInput[^4].

After passing all unit tests, several refactorings were conducted.The first refactoring involved splitting and extracting methods from HangmanUI's play() method.

The play() method was divided into several single-responsibility methods such as displayGameState(), processUserGuess(), displayEndGameMessage(), promptUserForGuess(), replay()[^5].

The second refactoring changed the HangmanGameLogic class to a singleton pattern.

All game attribute-related variables (MAX_TRIES, MAX_SCORE) are defined as public static constants in GlobalReference class, avoiding magic numbers[^6].

Method names follow the Intention-Revealing principle, and class names are all named with nouns[^7].

All refactorings passed the unit tests.

Other software development practices

Branch merging was used when modifying HangmanGameLogic to a singleton pattern. A new branch named singleton was created for this transformation. After passing the unit tests, this branch was merged into the main branch

Extra functionality

A replay mode was added, allowing users to input 'y' at the end of the game to continue to the next round.

References

^12.1 Unit Testing - CS5031 Software Engineering Practice (st-andrews.ac.uk)

^22.2 Test Objects - CS5031 Software Engineering Practice (st-andrews.ac.uk)

[^3][Combine Functions into Class (refactoring.com)]Combine Functions into Class (refactoring.com)

^4Collapse Hierarchy (refactoring.com)

^5Extract Function (refactoring.com)

^6Replace Magic Literal (refactoring.com)

^7Catalog of Refactorings