Introduction to Homework 10 : Hangman

For HW10, you may work as a group (no more than 2 students). Note: As an alternative to this assigned homework, you can define a final project of your own design.

This is a *2-part* homework assignment.

This is just an introduction to Homework 10. More details and instructions will be provided once the assignment is officially "assigned".

Part 1: Traditional Hangman

Traditional Hangman

Hangman is a 2-player word-guessing game generally played by having one player think of a word and the other player trying to guess that word letter by letter. Read more information about the game here.

Similar to Battleship, we want you to program a computer versus user version of the game. The computer will pick a random word from a dictionary that we will provide to you in a file. Note that we want you to use this dictionary file and only this dictionary file. It makes it easier for our testing.

- Represent the word to guess by using a row of underscores, representing each letter of the word (e.g., 'dog' would be shown to the user as _ _ _).
- The user guesses one letter at a time. Every correct letter that they guess should be shown in its correct location(s). Every guess that is incorrect will be shown in a list right underneath the word.

Your Task

1. Read and parse the provided dictionary .txt file in Java, cleaning it up, if/when needed.

2. Design the traditional hangman program

We are not going to provide you with a specific design for this homework. However, we will provide some hints about the design and implementation.

Part 2: Evil Hangman

Evil Hangman

The evil version of this game basically exploits the computer's ability to store a large amount of information. In the traditional version of the game, the computer has to stick to the original word as the user guesses. In the evil version, the computer keeps changing the word in order to make the user's task harder.

The algorithm that drives Evil Hangman is fairly straightforward. The computer begins by picking a random word from the entire clean list of words, and maintaining a list of only the words of that length. Whenever the player guesses, the computer partitions the words into "word families" based on the positions of the guessed letters in the words of a particular length. For example, if the full word list is HEAL, HELP, BELT, HAIR, and MAKE and the player guesses the letter 'E', then there would be four word families:

- E E -, containing H**EE**L
- E - -, containing H**E**LP and B**E**LT
- - - -, containing HAIR
- - - E, containing MAK**E**

Once the words are divided into these groups, the computer picks the largest of the groups (with the most words) to use as its remaining word list. By doing so, it gives itself the maximum chance to dodge the user's guesses.

It then reveals the letters in the positions indicated by the word family. In this case, the computer would pick the family - E - - (because it has 2 words and the other families containing an E only have 1 word each). The computer would keep only the - E - - family of words and reveal an E in the second position of the word to the user.

Your Task

Add the evil hangman functionality (as described above) to your traditional hangman program. When you launch the game from the `HangmanGame` class, the computer should decide which version of the game to play with the user. Your goal as the programmer is to ensure that, for the entire game, the user never knows what version of hangman is being played (evil or traditional).

Testing

Regardless of your design, we expect you to write unit tests for every public method. The only public methods that you can leave untested are those that perform file I/O, and simple getters and setters. You should keep the file I/O in as few methods as possible.

Evaluation

You will be graded out of 40 points :

- Code writing, functionality, and design (*20 pts*)
    - Does traditional hangman work as expected?
    - Does evil hangman work as expected?
    - Does the computer choose which version of the game to play and can the user tell?
    - Did you make good design decisions about code re-use?
        - How much code is being shared between traditional and evil hangman?
        - How is this code sharing being achieved in your design?
    - Does the code take too long to run? (10 seconds would be too long!)
- Loading, parsing, and navigating dictionary (*10 pts*)
    - Did you keep file I/O in as few methods as possible?
    - Did you accurately clean the dictionary file based on the provided guidelines?
    - What data structure(s) did you use?
- Unit testing (*5 pts*)
    - Did you write unit tests for every public method (excluding methods that perform file I/O and simple getters and setters)?
- Style & Javadocs (*5 pts*)
    - Adding Javadocs to all methods and variables, and comments to all non-trivial code
    - Generating HTML files from the Javadocs in your code using Eclipse