Assignment 4 Design

Alec Bargas and Connor Lewis

Perry EE422C

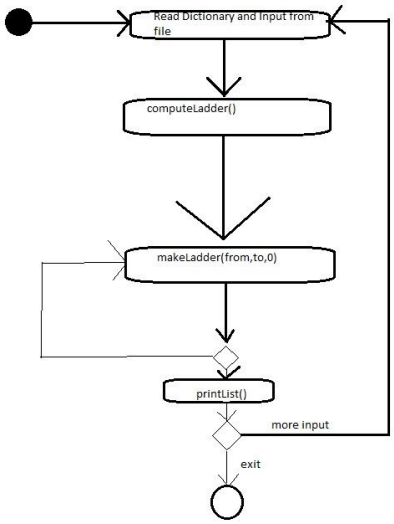
Algorithm for Driver

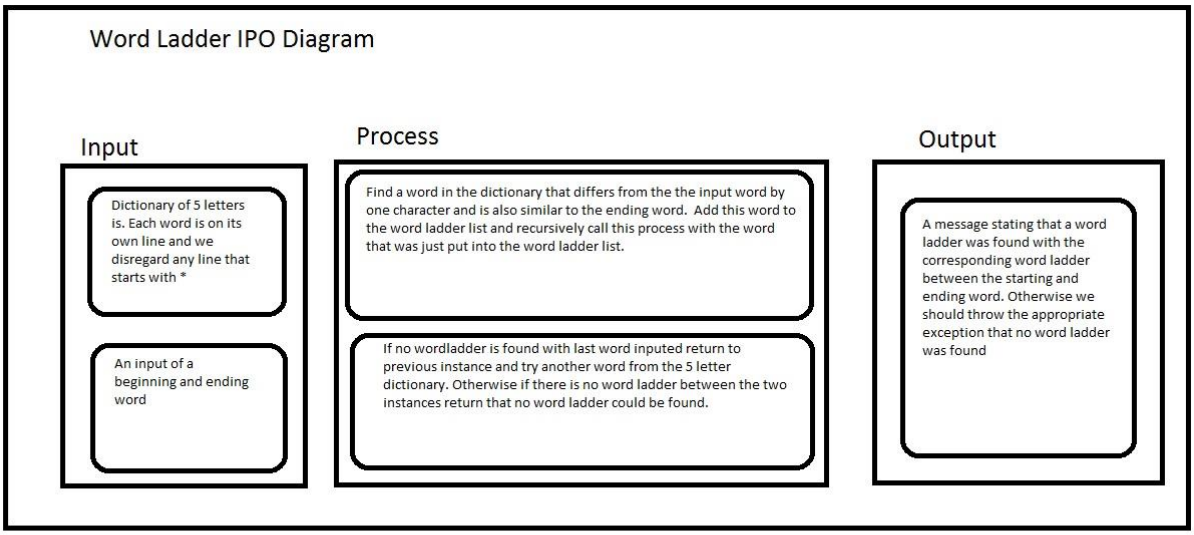
The algorithm for the driver is fairly basic for this program. First, we take the file that has all of the members of the dictionary in it and then separate the dictionary entries from the rest of the file and store them. The program just takes the first five letters of each line of input that is not a comment and a part of the program will later decide if they are valid. Then we take all of the input word pairs and organize them in a way that can be easily recognized by the program. The program ignores all spaces and will show an error if there is too much input in one of the lines. During this, we check to make sure that all of the input is valid. Then, we call the parts of the program that will construct the word ladder. If a word ladder does not exist, then the program tells the user that no word ladder can be found. The main function continues to process the inputs until there are no more pairs. The program then checks the word ladder to make sure that it is a valid word ladder.

Rationale Behind Design

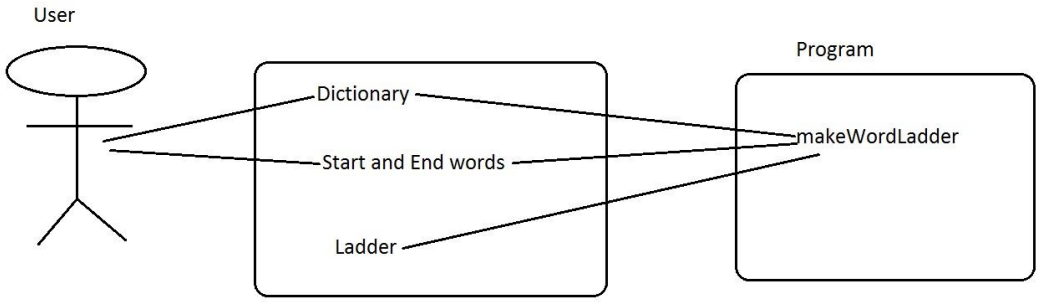
The reason for the design choice of the makeWordLadder function is because that is what the document originally told us to do. The main makeWordLadder function is divided up into different levels of searching for matching words. The function first checks for words that are the same as the input word, but a specific letter is different, which is the same as the letter in the end word. Then, if the program can't find one, it looks for words that are the same as the starting word but any the letter in any location of the starting word is the same as the corresponding letter in the ending word. Lastly, if the program can't find any of those, it looks for a word that is one letter off from the current word in a location other than the one that it just changed. The recursive nature of makeWordLadder function reflects the actual structure of word ladders because each "rung" in the ladder completely depends on the one that precedes it. The structure of dictionary (an ArrayList) reflects the nature of an actual dictionary, just a list of words. We considered doing an iteration based breadth first search method, but it seemed more difficult and longer to implement than the recursive depth first search. The disadvantages from a user perspective are that it could take longer to create the ladder and the ladder will not be the shortest possible one. From a designer perspective, it is more difficult and confusing to debug a recursive method. The method could be expanded to encompass ladders between many different types of objects that all have one common characteristic. The code can be enhanced further by adding different methods that could reduce the execution time of the program/make the search more efficient. The dictionary could be expanded so that each entry would contain more information: creating a dictionary entry class and not just storing them as lists of strings. The design uses an interface class that hides the implementation of the majority of the program. This allows other users to understand what the program does without knowing how it exactly works. In addition, there are several instances in which we attempt to catch potential errors. The use of multiple classes (especially the exception class) add a level of modularity to the program so that pieces can be reused/tested without the whole program.

Functional Block Diagram





Use Case Diagram



Test Plan

Black box testing is accomplished by inputting in the words that I know will work for the given inputs and observing if these corresponding inputs agree with what is returned. I am able to observe this without going through every step in my methods. The black box testing will allow me to check the various corner cases and ensure that the program works for a variety of different inputs, not only those that are guaranteed to work. With black box testing, no matter what the program looks like and how it functions, I can test for a specific form and accuracy of the outputs. With the test cases I wrote I am using white box testing because I input a start and end word and put these into our computeWordLadder function which implements all of our methods and gives us complete branch coverage of our program. By implementing a bad test case that tries to find a word ladder from atlas to zebra. I am able to check to see if our program will recognize that there is no such word ladder which will fail in our makeWordLadder method. By checking to see if there is a word ladder from smile to okay I am able to check that the program

will recognize that there is no possible way that there should be a word ladder between these two inputs because okay is a four letter word this should be caught in computeWordLadder. By implementing these two bad test cases I am able to check the case in which either one of our branches fail. I also checked the validateResult method first because if that method fails then I would be unable to check if the previous results from computeWordLadder are correct. With all of these methods I am using black box testing to check the inputs and outputs and am able to check all of the branches from my functional block diagram.

UML Class Diagram