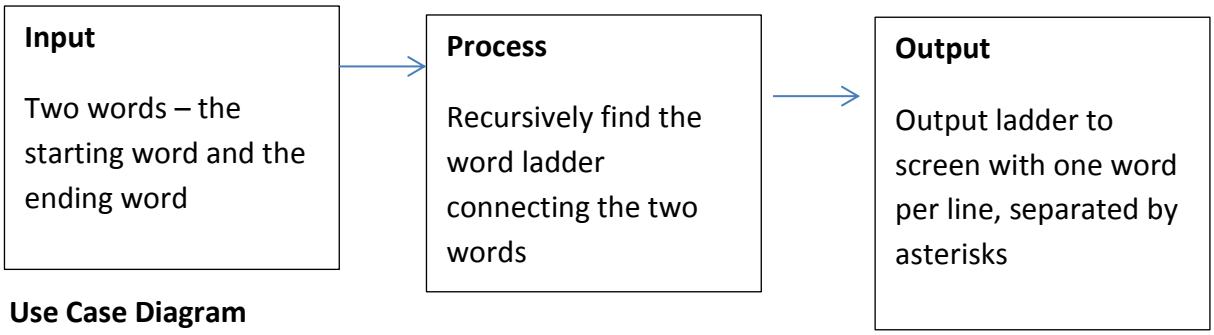
Sneha Vasantharao and Jai Bock Lee
Sv8398 and jbl932

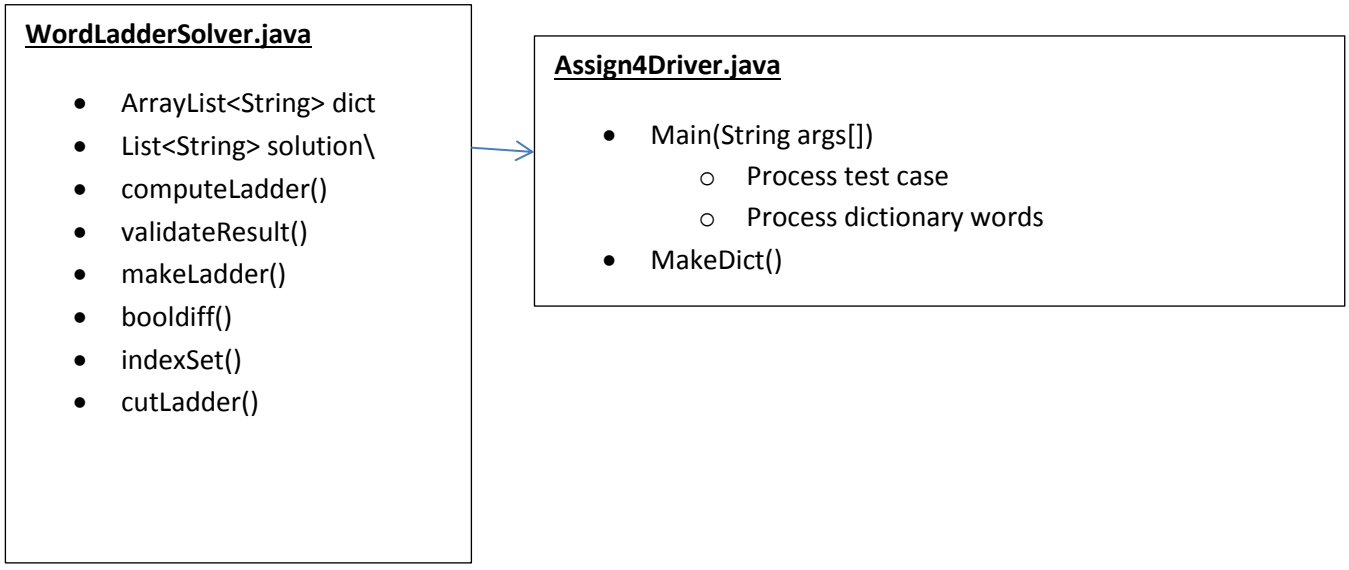
Design

1. System IPO Diagram

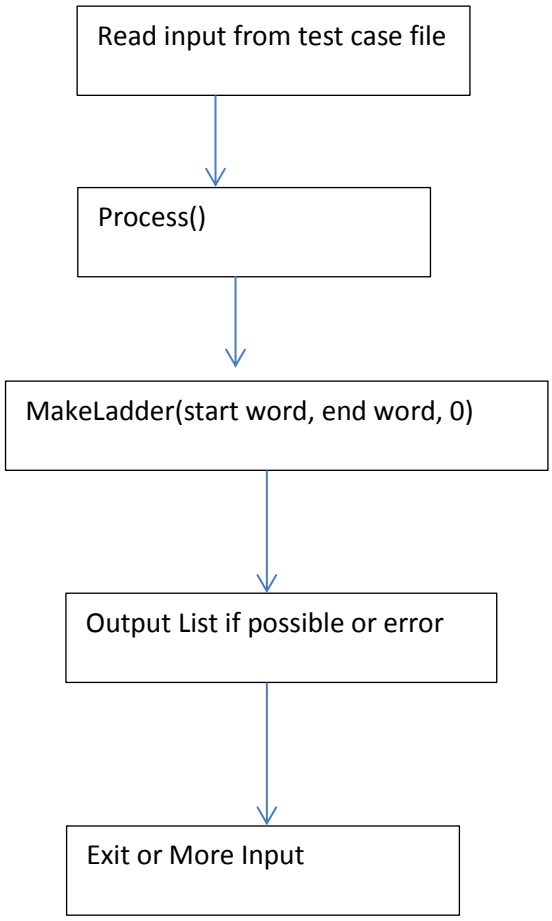


2. Use Case Diagram

3. UML Model



4. Functional Block Diagram



5. Algorithm

The algorithm needed for the driver logic in the main method was explained to us in the pdf instructions. The algorithm was required to use recursion to translate a five letter word into another five letter word by manipulating or changing one letter at a time. Our recursive method is makeLadder() in the WordLadderSolver.java file. The three inputs given to this method are the starting word, the ending word and the index of the last changed letter. We initially set the index to -1 because there might be words that have a difference of one letter which can be easily translated with one iteration. The length of both the start words and the end word are checked to make sure that they equal 5. Then, both the words are searched in the dictionary to make sure that they are valid words. After this, we take the start word and parse through the dictionary to find another word that has a difference of one letter from this word. All of these words are gathered and checked to see the number of letters of difference these words have with the end word. The one with the least letters of difference is then picked as the next word in the ladder. We used a list called “soln” to store the solution ladder and print this at the end if a complete solution is found.

Main method in Assign4Driver.java:

- Process input from test case file
- Check to make sure if input has any errors
- Separate input line based on two words
- Process dictionary words input
- Make dictionary ArrayList of words later used to compare the two input words to

6. Rationale

Our program follows the OOD guidelines by separating the functions into their appropriate classes and separating the classes into appropriate methods. Changes in the WordLadderSolver class will not affect the Assign4Driver class because there is low coupling and therefore, project maintenance is significantly easier. Assign4Driver class handles input error handling and creating the dictionary while the WordLadderSolver class actually creates the ladder. This means that we have high cohesion because each class manages a significant component of the program. We also have high information hiding because we segregated the designs in different classes and different methods, so that if one part of the program changes, we will not have to make significant modifications to the rest of the program. One of the main alternatives that were considered was making the computeLadder() method iterative instead of recursive. This change would make it significantly easier to debug and the logic that was used in this method would be much easier to follow. But an iterative method would not be as optimal as a recursive method. We also considered using HashMaps instead of an arraylist for the dictionary. This would reduce the time complexity because HashMaps have a time complexity of O(1) when it comes to search functions. We decided against this method because the output returned relatively quickly either way. So although HashMaps are more beneficial with a program perspective, it did not matter as much when it came to the user perspective. For future enhancements, we could use a HashMap instead of an ArrayList to reduce the time complexity. We could also expand the program to accommodate words of different sizes instead of just five letter words. Our design reflects the interactions and behavior of the real world objects that it models because the Assign4Driver class represents a dictionary in the real world because it contains a list of all of the five letter words in the dictionary.