Names: Ronald Macmaster, Horng-Bin Justin Wei

UT EID: rpm953 and hjw396

PA4: Fun with Word Ladders

1) Analysis

Problem Statement:

Design a program to traverse a word ladder for any given pair of 5-lettered words.

Utilize the word classifications from a given dictionary of legal English 5-letter words.

Print a warning message if the word ladder does not exist.

The word ladder must not necessarily be the shortest word ladder.

Input:

Word pairs are input from a given file. The filename is specified through the command line.

General Word Pair input
dears fears
stone money
money smart
devil angel
atlas zebra
heart heart
babes child
mumbo ghost
ryan joe
hello buddy
hello world
heads tails

Valid input : 2 fiver-letter words in the English language.

Input is not necessarily valid! Report errors using exceptions.

Output:

Output to the console a word ladder from the starting word to the ending word.

The words in the ladder must be valid English words

If a ladder does not exist, Output: “There is no word ladder between … (word1) and (word2).”

If any of the input words are not valid English words, print

Questions?

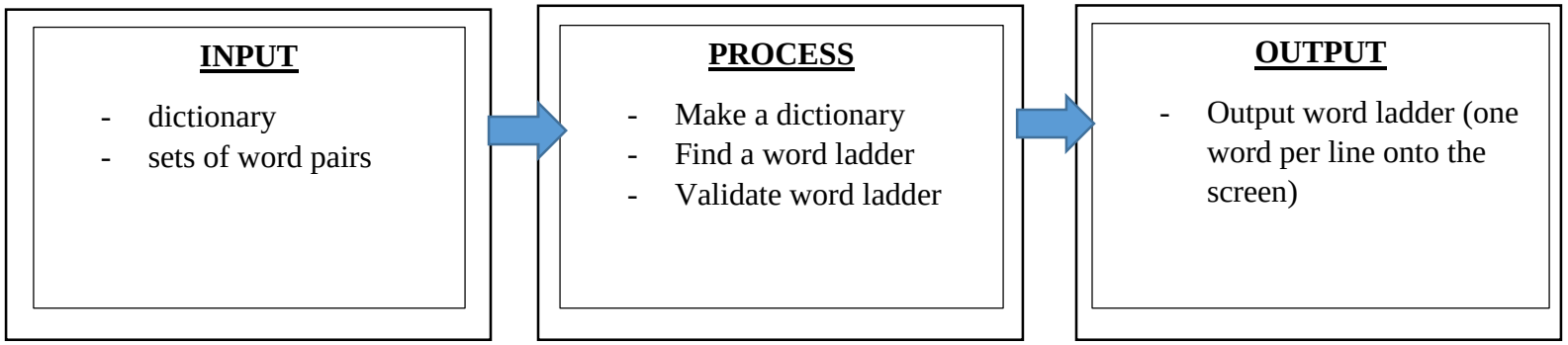
Should we map every single 5-letter words in a single graph?

How do we perform a search for the words?

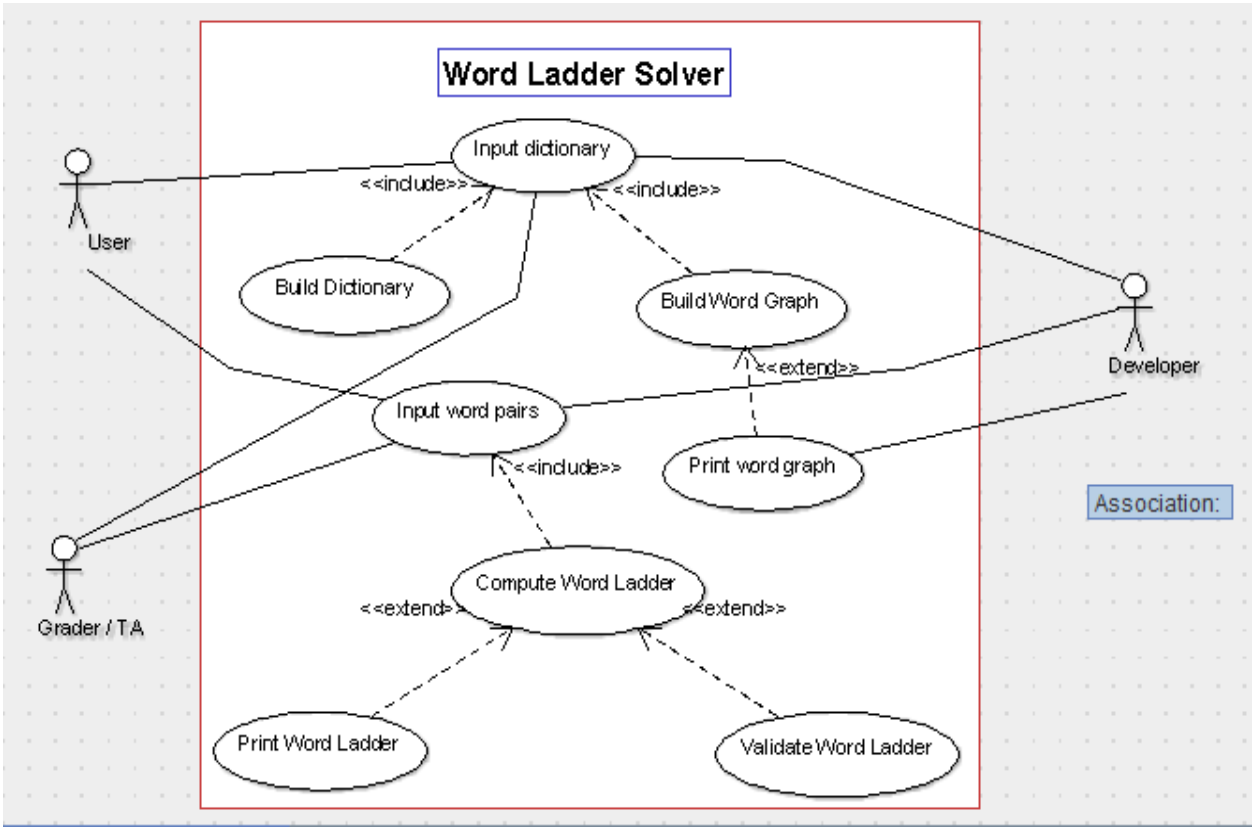
2) Design

Architecture Models:

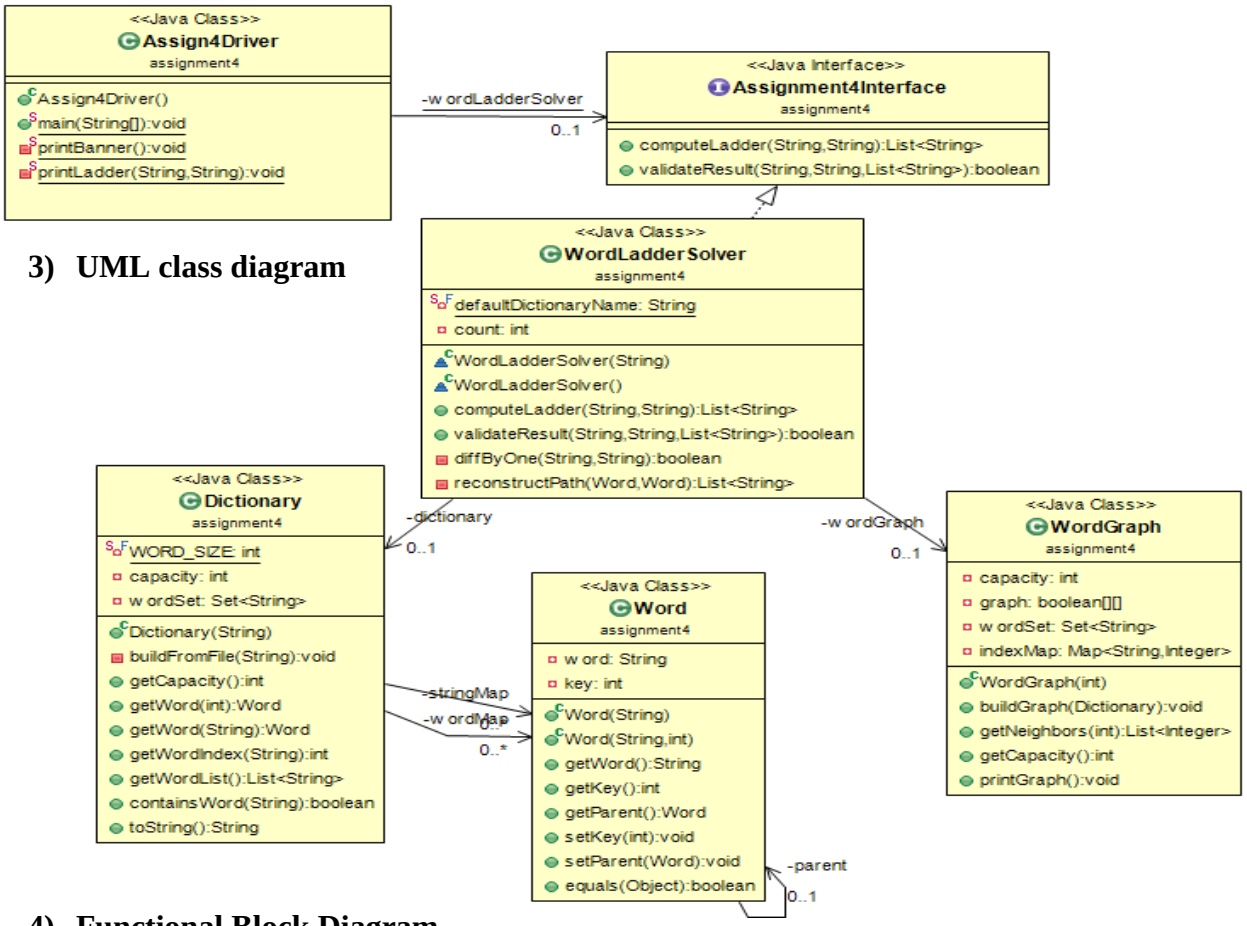
1) System IPO Model



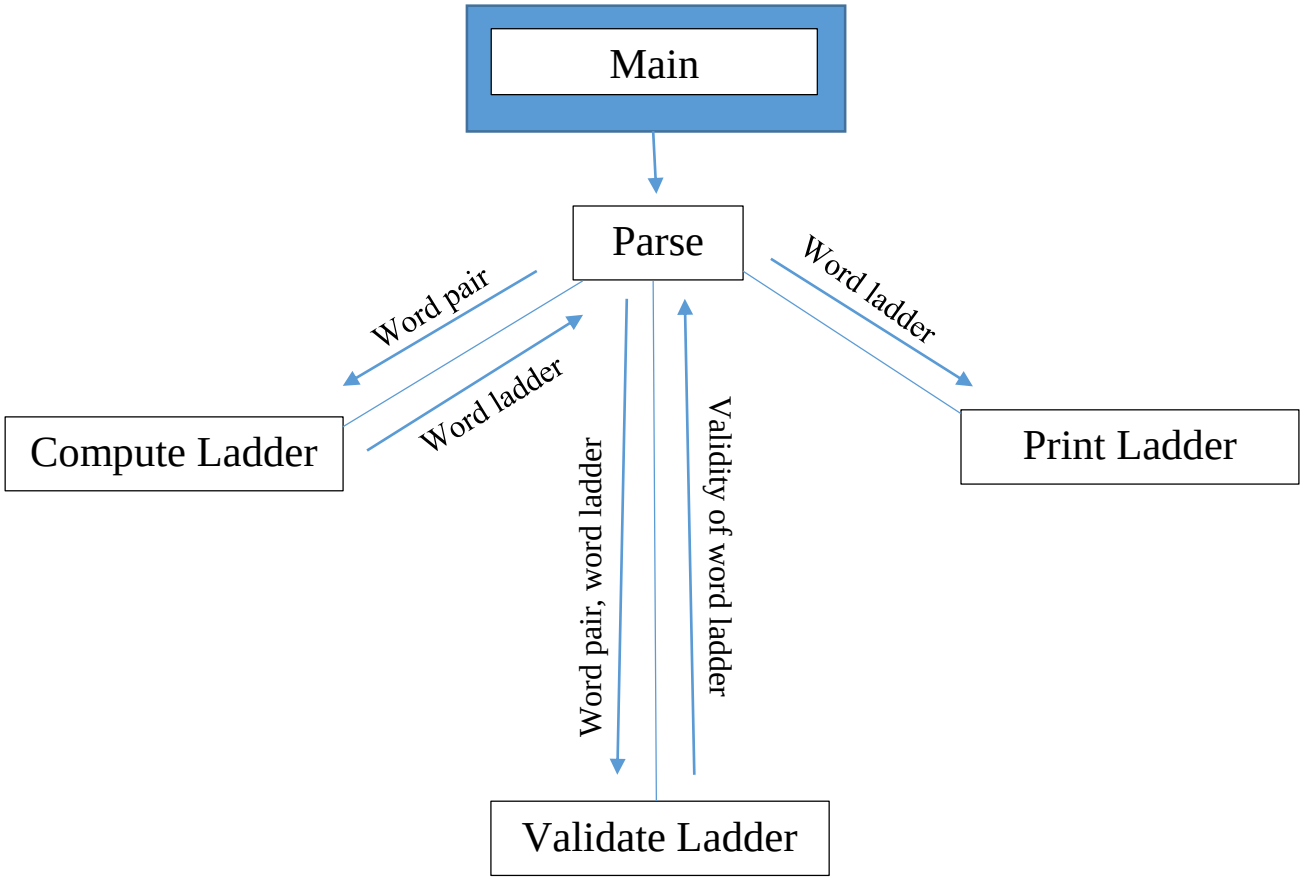
2) Use-Case Diagram



3) UML class diagram



4) Functional Block Diagram



Algorithms

Driver Algorithm: (Assign4Driver)

1) Read / Clean word list from the dictionary
2) Build the Word Graph from dictionary data
3) Loop:
    a. Read in word pair
    b. Output delimiter **********
    c. If word data invalid: throw exception and repeat
    d. Compute Ladder from start to end
    e. If no ladder exists: Print no ladder exists! And repeat
    f. Else: print word ladder and repeat
4) Output delimiter **********
5) End driver.

Probable Graph Search Algorithms:

Breadth-first search

```
BFS(Graph, roof){
        For each node in G
                N distance = infinity
                N parent = null
        Empty Queue Q
        Root distance = 0
        Q enque(roof)
        While(not empty){
                For each node adjacent to current
                If(n.distance = invfinity)
                        n.distance = current.distance+1
                        n.parent = current
                        Q.enqueue(N)
```

Depth-first search

```
DFS(Graph, root)
        Empty Stack S
        S.push(root)
        While(S, not empty)
                V = s.pop()
        If v not discovered
                Label v discovered
                For all adjacent edges (v, w)
                        s.push(w)
```

Rationale:

Our OOD reflects the interaction and behavior of the real-world objects in certain aspects. For example our dictionary class is a collection of words just like a physical dictionary. Our dictionary class also allows us to determine if a string of letters is a real word just like how a physical dictionary would. We chose to do a breadth first search (BFS) design and an alternative that we considered was a depth first search (DFS) design. The advantages and disadvantages of either designs is the time it would take to complete the search or the amount of memory needed in a programming perspective. If the tree is deep and solutions are rare, then it would be most likely faster to pick a BFS over a DFS. On the other hand, if the tree is very wide, then a BFS might take up too much memory so a DFS would be better. If solutions are frequent and the tree is deep then a DFS might be better. Comparing BFS and DFS, the advantage of DFS is that it has much lower memory requirements than BFS. On a user perspective one may want to do the program iteratively instead of recursively, which would make the code easier to read and help with white box testing. Although in a programming perspective a recursive function may be more optimal. We could expand our program and design by allowing for more than 5 character words and/or more than 2 word input pairs. Allowing more than 2 word inputs would work by creating a word ladder from one word to another, but at least containing the third word inside of the word ladder. Our design adheres to principles of good OOD design by separating functions into their appropriate classes. By doing so we also have good cohesion in our project by only having elements of a class together that actually belong together. We also have low coupling which means our classes do not depend on each other that much and are more independent. Lastly, we incorporated good info hiding with our segregated design decisions in the project that are most likely to change which protects other parts of the program from extensive modification.

A paragraph describing the rationale behind your design. This would include:
a) How does your OOD reflect the interaction and behavior of the real-world objects that it models
b) What alternatives did you consider? What were the advantages/disadvantages of each alternative both from a programming perspective and a user perspective?
c) What are some expansions or possible flexibilities that your design offers for future enhancements?
d) How does your design adhere to principles of good design: OOD, cohesion, coupling, info hiding,

Test Plan

Summary

We will utilize the JUnit testing framework to encompass our program as a black box. The black-box testing is done through the Assingment4Interface. The white-box testing / branch coverage is done through unit tests in JUnit. **Junit Tests can be found in the DictionaryTester, GraphTestser, and WordLadderTester files.**

Black Box

We need to black-box test the computeLadder and validateResult methods. First, we need to confirm the correctness of the validateResult method. This can be done through unit tests with known start, end, and word ladder lists. Then, we can apply the validateResult method to test the correctness of the computeLadder method. This will allow us to ensure the correctness of the ladders generated by our WordLadderSolver.

White-box

We can use branch coverage to unit test our Dictionary, Word, and WordGraph Data structures. The static methods and constructor building will be covered under the black-box testing in integration tests. However, we need to test the various get and set methods for each data structure. Additionally, we have developed a printGraph method for the WordGraph class that will allow us to view a .csv of the AdjacencyMatrix in Excel. The most important method to unit test is the getWord method from the Dictionary and WordGraph.

Possible Black-Box tests

/***** Regular Use ***/
gears gears
aargh zowie

/** improper word length **/
abc like
this time

/** Valid length, invalid word **/
aarrk llloo
slkkj sloke (Also include tests for null inputs and invalid alphanumeric strings)