Analysis and Design for Word Ladder Assignment
Team # 33:
SMITH, KASSANDRA kss2474 (16180)
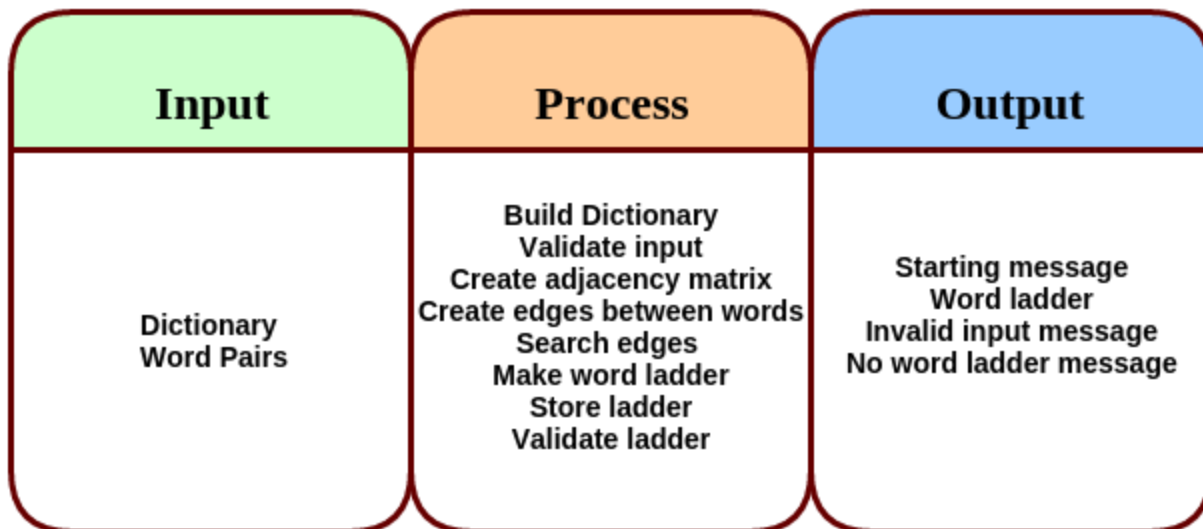HADIMOHD, AFTAB ah35368 (16180)

ANALYSIS

The Assignment asks us to create a "word ladder" between two given words. From the start

word, one letter is altered at a time to create another word that exists within the dictionary until

the end word is reached. These two words are input as strings with some sort of white space

separating the two. (Note that on Piazza Post @193 states that the input will actually be the same

format as the code stub).

Some given rules:

- The input should be five letters long

- The input should be a real word which exists within the dictionary of five letter words

- The word ladder can be as long as necessary

- Throw a NoSuchLadder Exception if the ladder does not exist

- Handle invalid input

DESIGN

System IPO diagram

Input	Process	Output
Dictionary Word Pairs	Build Dictionary Validate input Create adjacency matrix Create edges between words Search edges Make word ladder Store ladder Validate ladder	Starting message Word ladder Invalid input message No word ladder message

Analysis and Design for Word Ladder Assignment
Team # 33:
SMITH, KASSANDRA kss2474 (16180)
HADIMOHD, AFTAB ah35368 (16180)
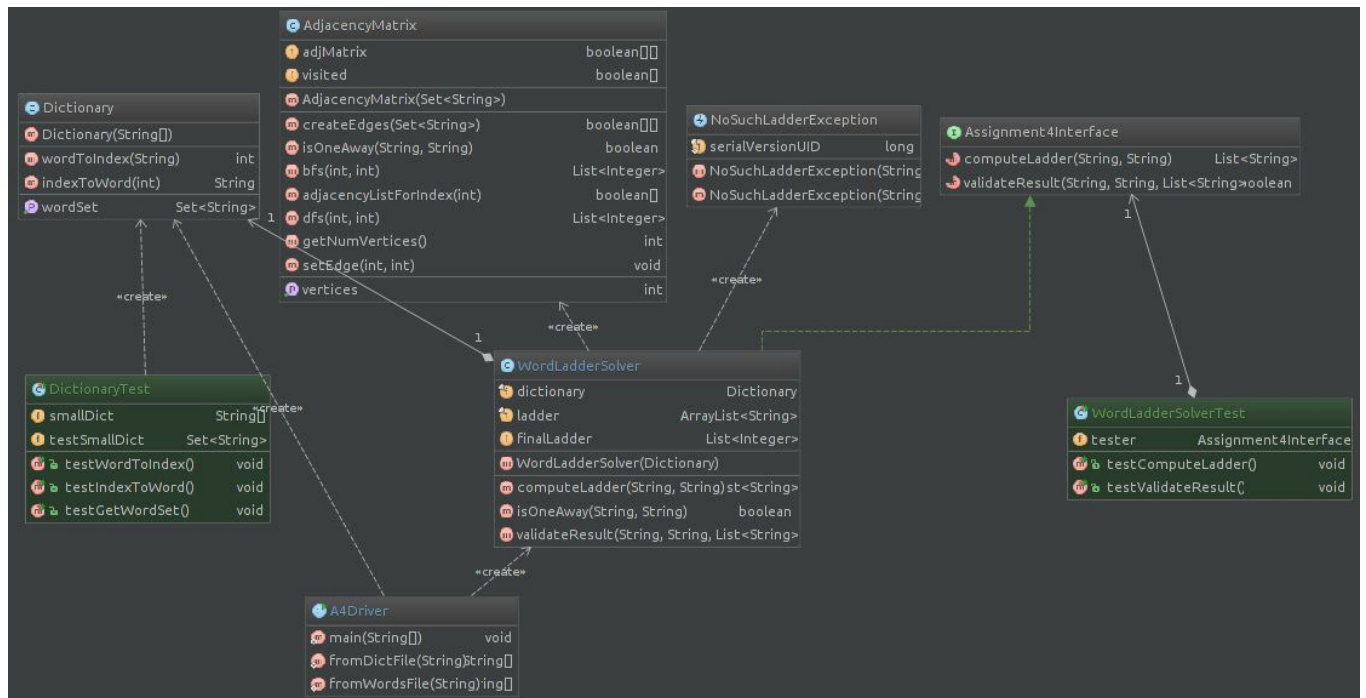
Use-case diagram
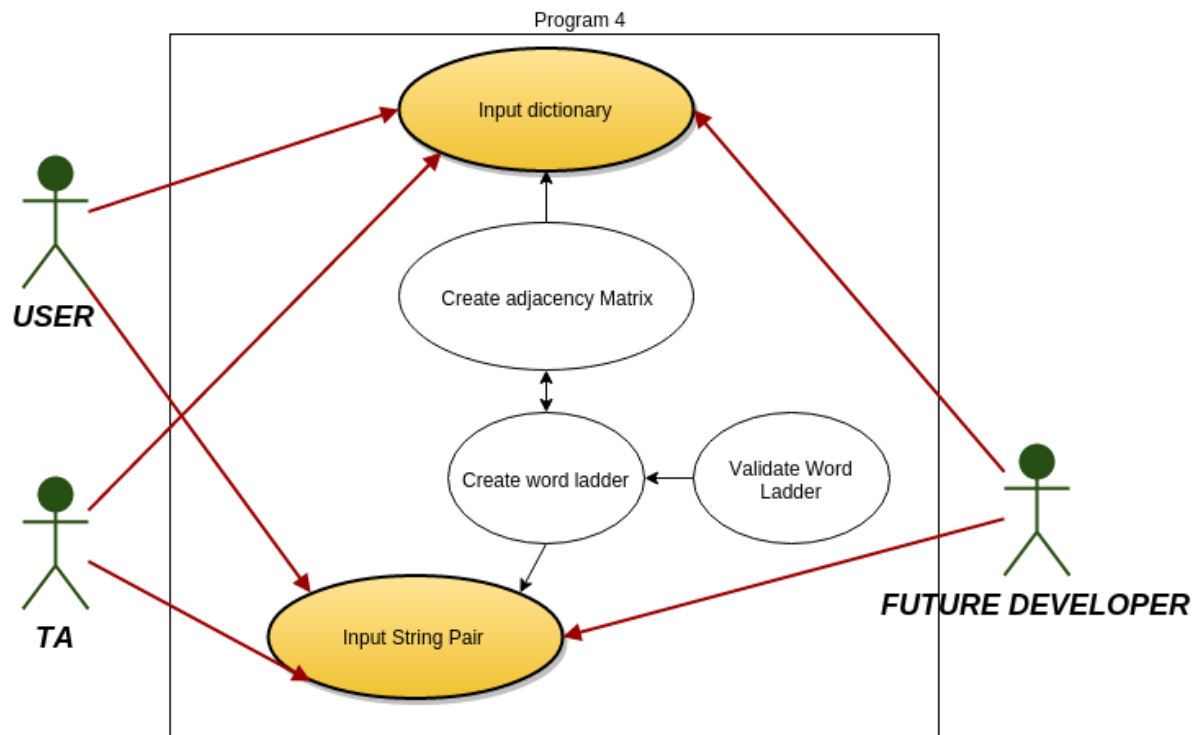
UML model of the needed classes and their relationships

Analysis and Design for Word Ladder Assignment
Team # 33:
SMITH, KASSANDRA kss2474 (16180)
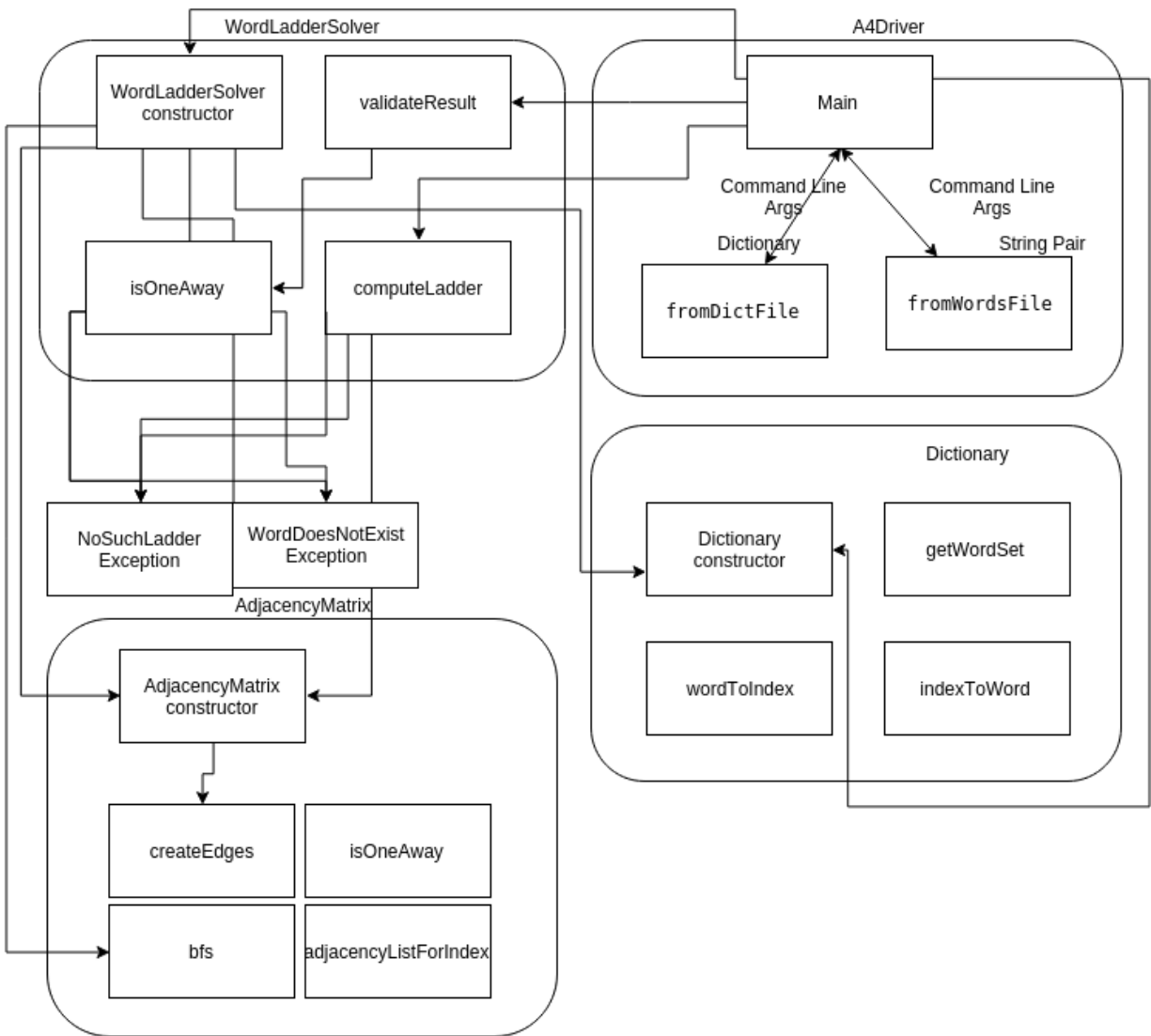HADIMOHD, AFTAB ah35368 (16180)

Functional block diagram showing the calling relationships between methods (library methods are not included)

Algorithm for the driver (main method)

1) make sure the command line input is valid

2) create a dictionary by reading from the dictionary

3) create a wordLadderSolver and interface

4) check that word pairs are valid

5) create a ladder

6) if invalid throw exception, catch and output message

7) clear and continue

Rationale Behind Design.

a) How does your OOD reflect the interaction and behavior of the real-world objects that it models

The Dictionary methods are useful the same way a real dictionary would be, and the adjacency matrix and the word checking mimic real-world checks.

b) What alternatives did you consider? & What were the advantages/disadvantages of each alternative both from a programming perspective and a user perspective?

We initially decided to create a depth-first-search on our adjacency matrix but realized the ladders were very long and time consuming and thus switched to a breadth-first-search.

c) What are some expansions or possible flexibilities that your design offers for future enhancements?

The design could be expanded by using a full dictionary or adding functionality to create ladders between differently sized words.

d) How does your design adhere to principles of good design: OOD, cohesion, coupling, info hiding, etc?

The correct methods are made private, public, static, and all information is shared properly. We break the several methods related to the Dictionary and the methods related to the Adjacency Matrix into their own classes to allow for modularity and cohesion.

Analysis and Design for Word Ladder Assignment
Team # 33:
SMITH, KASSANDRA kss2474 (16180)
HADIMOHD, AFTAB ah35368 (16180)

Test Requirements

Black Box Testing

We tested three main cases, string pairs that work, string pairs in which there existed no ladder, and string pairs which were invalid. This was done simply by running the program with a test file and documenting what we expected the output to be and what it was after running.

an example valid string pair: "honey", "stone"

an example string with no ladder: "atlas", "zebra"

an example string with invalid input: "joe", "garbl"

White Box Testing

The majority of our testing was white box testing. During the testing phase breadcrumbs were placed to make sure that each function was behaving as expected "System.Out.println("Entering the constructor for Dictionary")" is one example. We also set strategic breakpoints to test the values at certain points (for example that the length of the dictionary was 5757 as it should be).

JUnit tests

We write a test for the Driver, the WordLadderSolver, and the Dictionary.

Each of these tests checks that the end conditions of each method is correct in the following way:

WORDLADDERSOLVER:

testComputeLadder - tested that the ladders we expected to break did in fact break

testValidateResult - made sure that the ladders were not incorrect

DICTIONARY:

testWordToIndex - tests on a smaller dictionary to make sure that the method is properly getting indexes from strings

testIndexToWord - tests on a smaller dictionary to make sure that the method is properly getting strings from indexes

testGetWordSet - tests to make sure that the set returned differs in no way from the one input