Assignment IV – Word Ladder

Hari Kosuru – Kevin Yee

1 Test Requirements – BlackBox and WhiteBox Testing

Our test plan follows the two conventional styles of testing: black box testing and white box testing.

BlackBox:

Our black box testing approach is derived from pre-emptively knowing the results to certain inputs:

From the given solutions, we know that the words “heads” to “tails” has a possible word ladder. Thus, we tested the existence of such ladder and used a method known as “validateResult()” to confirm that the first and last entries of the list array contains “heads” and “tails” respectively. Furthermore, we confirmed that all the results between the first and last entry only differed by one character.

We also know that the words “angels” and “devils” have no word ladder between them. The method in which the solution was derived is no interest to us in blackbox testing. We simply confirmed that the list returned was empty.

WhiteBox:

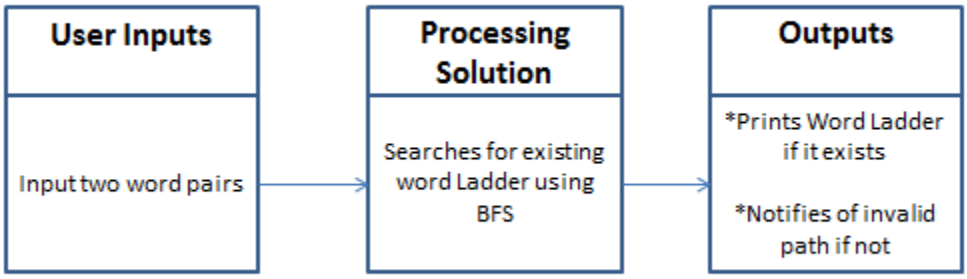
Our white box testing approach is derived from a programmer’s perspective and knowing the constraints.

We tested the wordLadder using special cases such as words that are note equal to five letters. Words with such properties were returned with a message prompting the user that there exist no paths between those two words.

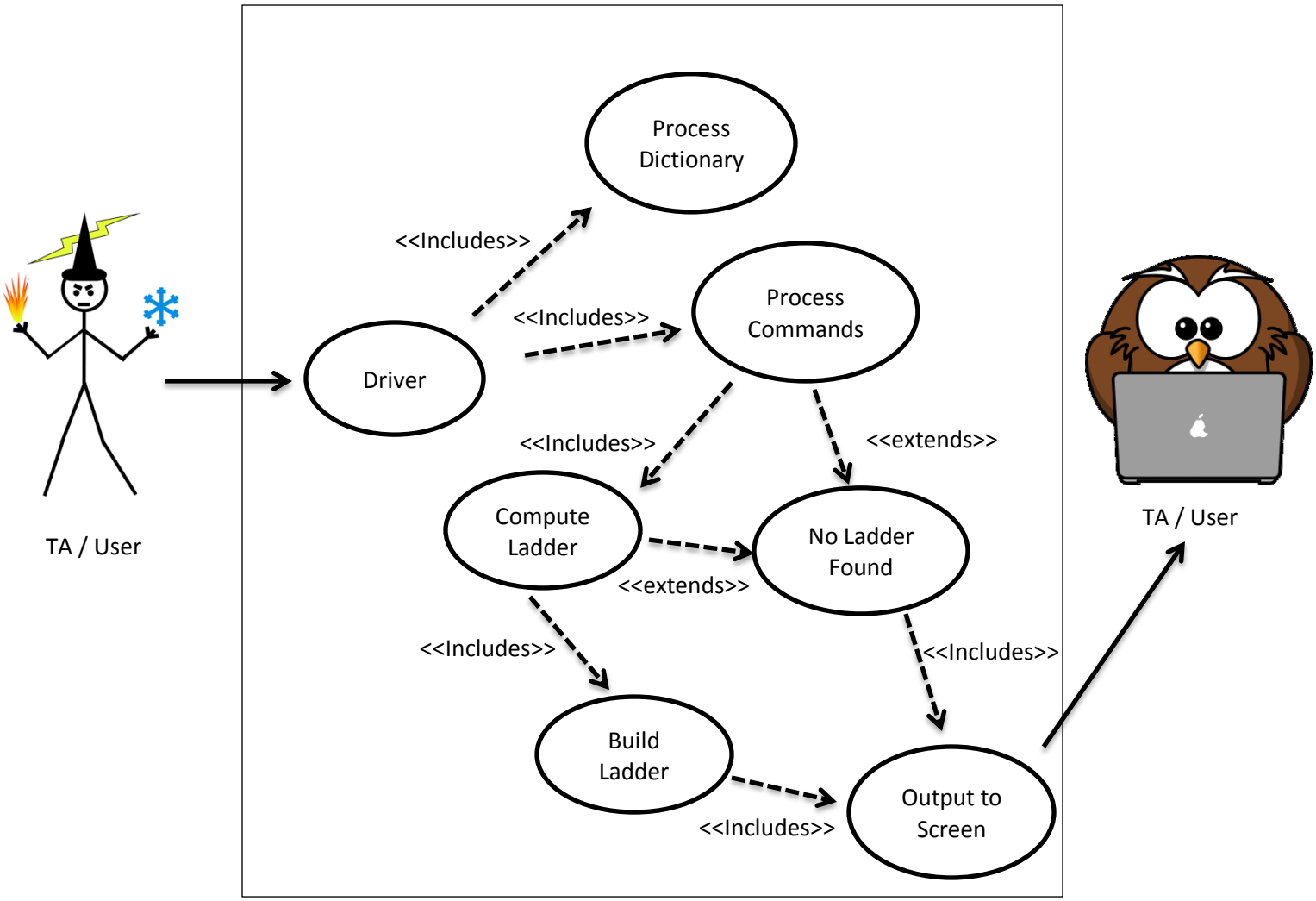
Furthermore, other knowledge and test cases we were able to derive is the input of null strings. The input of null strings returns an empty list as well.

To structure our whitebox testing further, every correct input will print a message indicating that the result is indeed correct using our “validateResult()” method.

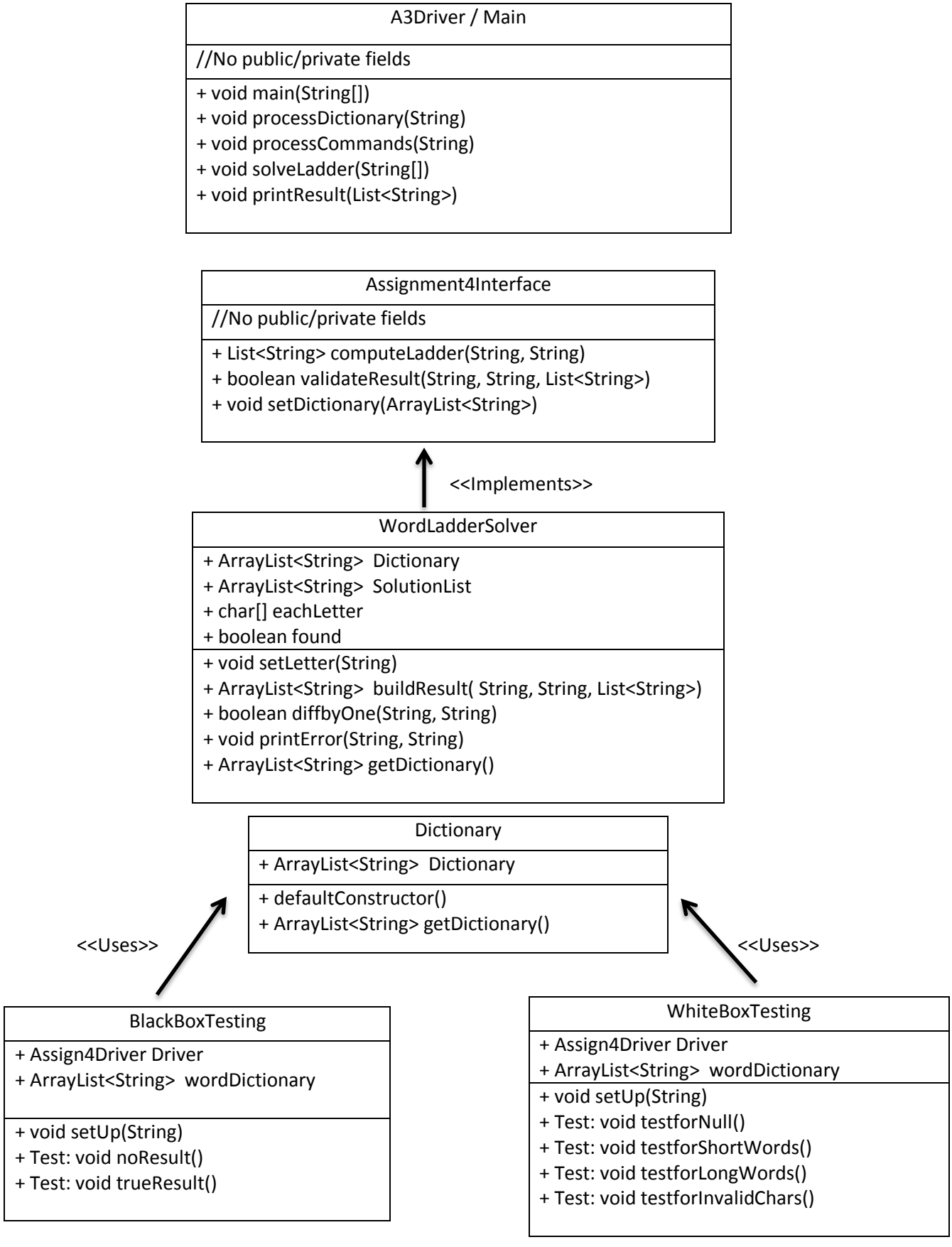
2 System IPO



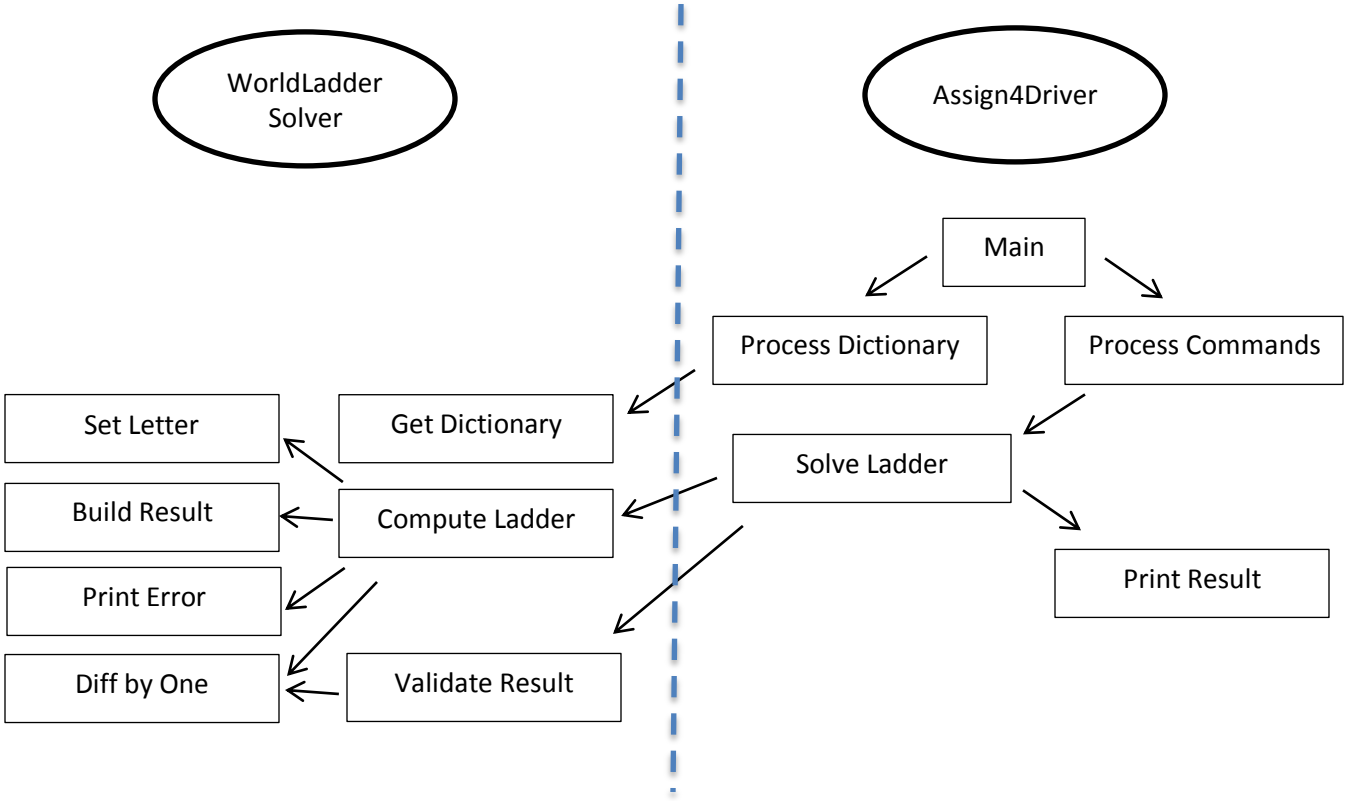
3 Use Case Diagram



4 UML Diagram



5 Functional Block Diagram



6 Algorithm of Driver

```
main(String[] args):
    If args.length is not 2:
            Display error message
            Exit program
    processDictionary(args[0])        // args[0] = dictionaryLocation
    processCommands(args[1]) // args[1] = testFileLocation
// main ends

processDictionary(String dictionaryLocation):
    dictionaryFile = FileReader (dictionaryLocation)
    for (s = FirstLine; s is not null; s = nextLine):
    if (s does not start with *)
```

```
            getDictionary = first five chars of s
if (file not found or IO failed)
    display error
return to main
// processDictionary ends

processCommands(String testFileLocation)
    testFile = FileReader (testFileLocation)
    for (s = FirstLine; s is not null; s = nextLine):
            String[] inputs
            inputs = s delimited by spaces
            if (inputs.length < 2 or inputs.length > 2)
                    display error
                    continue loop
            if (inputs.length = 2 and inputs[0] or inputs[1] are empty):
                    display error
                    continue loop
            solveLadder(inputs)
    if (file not found or IO failed)
            display error
    return to main
//pricessCommands ends

solveLadder(String[] inputs)
    startWord = inputs[0]
    endWord = inputs[1]
    if (getDictionary does not contain startWord or endWord)
            display error
            return to processCommands

    List<String> result = wordLadderSolver.computeLadder(startWord,
            endWord)

    if (result is not empty and validateResult(result) is true)
```

```
            print result (all elements in list)

        if (NoSuchLadderException was thrown)
            print message
        //return to processCommands
```

7 Description for Design choice

Our approach to the problem was using BFS, with internal implementations of hash maps and queues. From a programmer perspective, BFS was more appealing than DFS in terms of speed, since the problem had a rare solution (at most 1) and the solution could exist anywhere in the tree (not just the deeper levels). One downside, however, is that BFS will need higher memory requirements especially if the solution is deep in the tree, since the number of child pointers becomes larger as the tree becomes wider. From a user perspective, BFS provides the shortest word ladder, which usually allows for easy verification of the ladder by inspection. As for OO design, we ensured that access between the driver class and the solver class was limited and that all fields in both classes were private, with get and set methods as necessary. We also considered and tested many types of inputs (both valid and invalid) that were possible, and ensured that the program has a reliable mechanism to handle them properly. Our design also reflects appropriate interaction between objects as would be expected. For example, every instance of a solver class requires the same dictionary, and hence we opted for it to be static. We also supplemented the interface provided in our implementation order to specialize it for the algorithm we chose, while still adhering to the given framework. Whenever possible, we generalized our algorithm to allow for easy enhancements in the future. For example, if instead we were asked to implement a DFS, we could easily override the computeLadder method with a new algorithm. Even if we were given a dictionary 6-letter words, we would simply have to make small changes to our error handling checking and some internal data structures that would allow for 6-letter word ladders. As such, our program is generally quite flexible.