# CS21120 Assignment 1

## WordPlay

10/26/2012

Craig Heptinstall
Crh13

# Table of Contents

# Introduction

In this assignment I have been given, it has been asked of me to create a small program that can calculate and produce ‘Word-ladders’ which are a linking of different words by changing one letter at a time. For example, ‘look’ could too turn into ‘lock’ and so on. I have been told there should be two options of running the graphical or command line program, one where the user input a start word, and then input the size of the ladder the program should try and create. This first command will be named ‘Generation’. The second option the user should be given when starting the program should be a ‘Discovery’ section. This will allow users to input two words of equal length and have the program attempt to link these two words together in the shortest possible ladder. The program would have to have a dictionary available for it to check and link words together.

From looking at the specifications provided, I can see that I will have to complete the project in a number of steps. Firstly, gathering the dictionary data will be required in order for the program to be able to collect and ladder words. I plan to use the Linux dictionary for this, using the UNIX command line code to place a list of words into a text file. This will then be possible to read in from the program.

As for both sections of the program I will be implementing, I will have to look at different possible data structures I will be using in order to store the dictionary of words in a more organised and easy to ladder way. After looking at some different options, I have found the best way to structure my data for use in ladders would be either using graphs, Hash Tables or Hash Maps. Either one or a combination of these will provide me with the tools of sorting the data and making it readily available to search through.

But before I begin the implementation of my program using either of those options, ii will have to ensure I thoroughly design the program using the UML standard design diagrams. Starting with the use case in which I will be able to look at what the inputs and commands the user should be able to give will be. I will then construct a class diagram in order to plan what classes will need to be implemented, and their methods. Showing links between these classes will also help identify where data will be passed around if necessary. Other Possible UMLs I am considering within this project are state and activity, and also sequence diagrams. This would ensure I give a full-through design of how my system should work.

When it actually comes to the implementation, I have decided to go with a basic command line program, with the possibility of adding some kind of graphical user interface later if time allows. This will allow me to concentrate on the data side of the program ensuring the ladder system works. During the implementation, I will also be providing in this document my algorithm for the way the program will decide its ladders from word to word for the discovery section. I will present this in a Pseudo code style making it much more readable and understandable. It will be important to get an efficient algorithm for the sorting and calculation of ladders to ensure the ladders from one word to another are as short as possible, and also reliable.

On the topic of making my algorithm understandable, commenting my code will be a significant task within the project, as I will have to make sure that key lines of code are explained, and made so that the reader can see what is happening at different stages. I will consider JavaDoc for commenting also which will give a professional print out of the method listings and descriptions of what each should do.

In the penultimate section of this document, I will display the testing performed on my completed system showing the tests of input/ output, and most importantly the testing of my algorithm. I will choose to do some JUnit tests here to create more method specific tests, along with testing of exceptions such as invalid inputs. I will ensure that I test at least three different words and lengths for the first section of the program 'generation', and a number of different pairs of words for the other section. It will be very important to test all of the parts of my program that calculate the ladders to ensure it works as specified in the brief, and also that it does it efficiently without breaking. Any failed tests will also be noted, and the parts of the system possibly improved upon.

The final section of my program will consist of an evaluation of both my program and project in its entirety. I will look at how my program has met the requirements and how it possibly has not managed to do so. A 'possible improvements' section will explain how I think I could have improved the program if the timescale was larger/ had improved skills. When evaluating the project, I will look at the way I came across the planning and implementation, and if I felt I ran the project smoothly. At the end of this section I will write up a brief self-evaluation of how I think I performed.

## Designs and Justifications

In this section, I will outline the design of my system in attempting to meet the brief. Using a range of UML and Pseudo code for the algorithm, I should be able to provide a detailed view into how I will solve the problems involved in the creation of the system. The first of the design section will be the UML, beginning ultimately with the use case diagrams needed. For the UML section of the design, I have chosen to use StarUML, open source software which provides very useful tools on creating UML and allows the exportation of JPEG images.

### Use Cases

Below is an illustration of my use case diagrams for the projected system:

*Figure 1: After system start:*

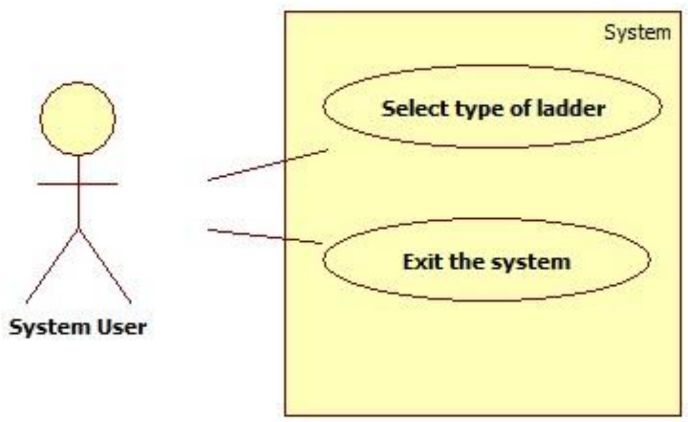


*Figure 2: 'Generation' section of program:*

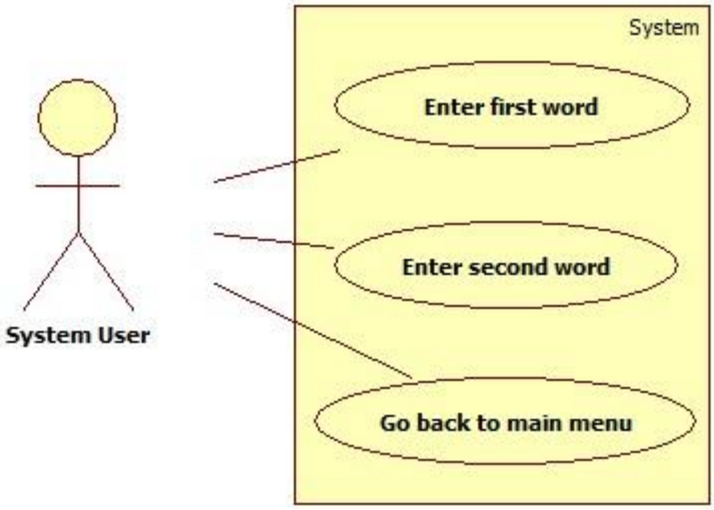
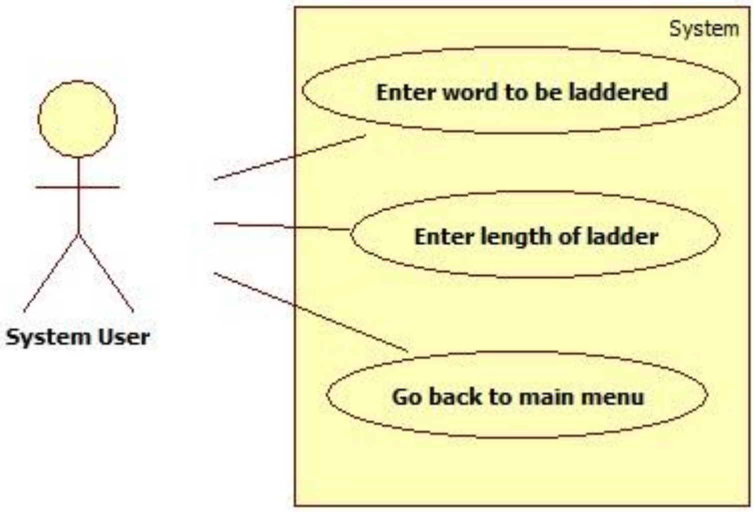



*Figure 3: 'Discovery' section of program:*




Now I have completed created my use case diagrams for the projected program, I will provide some short justifications of each use case. In the first (Figure 1.), this was a very simple one to establish, as all that the user should be able to do at this point of starting the program is to select which type of ladder they wish to create. The other option available here is to exit the program. Both these actions would be through the command line by just typing a specified String instruction.

As for figures two and three, these are very similar except for the difference that one allows users to input two words (discovery), whilst the second input from the program would be for the user to input an integer (length of desired ladder). Both ladder options allow the user to get back to the menu by typing a specific string such as 'x'.

## State diagrams

My state diagrams show the operations taken by the user in my system when performing different tasks:

*Figure 4: State diagram for word ladder*

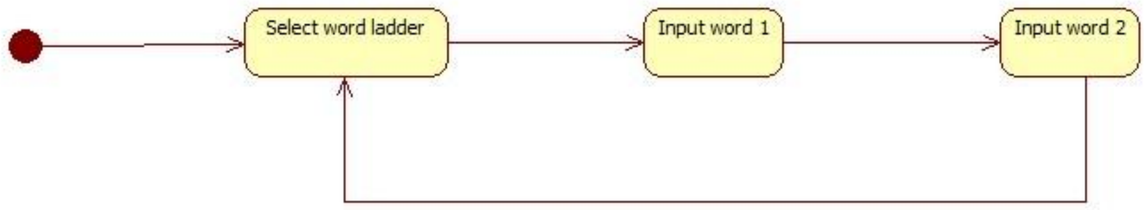



*Figure 5: State diagram for Step ladder:*

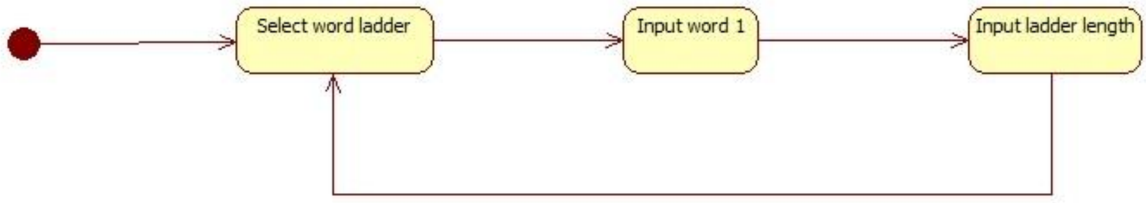
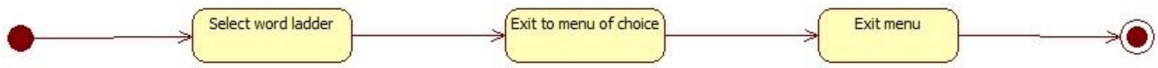



*Figure 6: State diagram for going back to menu/ exiting system:*




Alike the use case design justification I performed in the last section, I will now describe the state diagram justification for the above figures. Overall, these were relatively easy to create with only a few basic actions that take place within the projected program. For example, in both figure 4 and 5, it was a simple case of: from the starting position, the user would firstly choose a word ladder type, input the first word, then the length or second word, and then head back to the start where they then again inputted the first word.

The difference in the last state diagram (figure 6) is that this one does not loop as it shows an example of firstly selected a word ladder, and then using the command to exit to the menu, followed by exiting the program. This shows the simplistic action behind what should be starting the program and finishing the program for a general user. I find state diagrams very useful as a tool within UML 2 because of the clear way they illustrate the actions the user can go through to get to an end of an operation such as starting and exiting through a program.

## Activity diagrams

Following on from state diagrams, it is important I can show the way the program should handle the operations given by the user and plan what the system will do after and before operations. I have displayed state diagrams below:

*Figure 7: Activity diagram for the word ladder:*

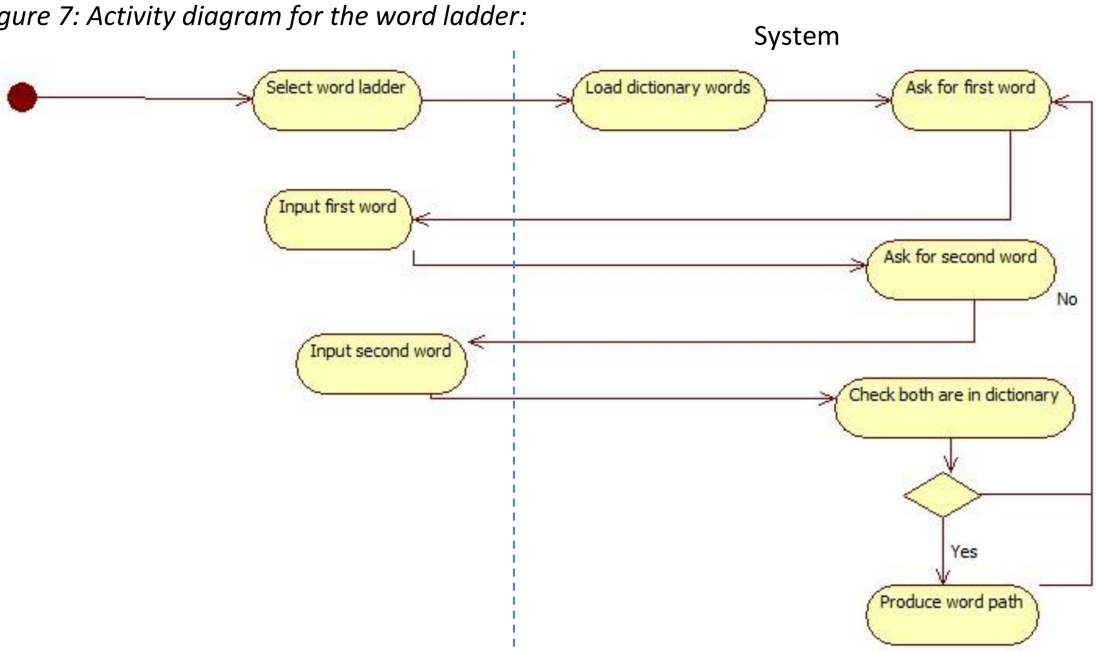



*Figure 8: Activity diagram for step ladder part of program:*

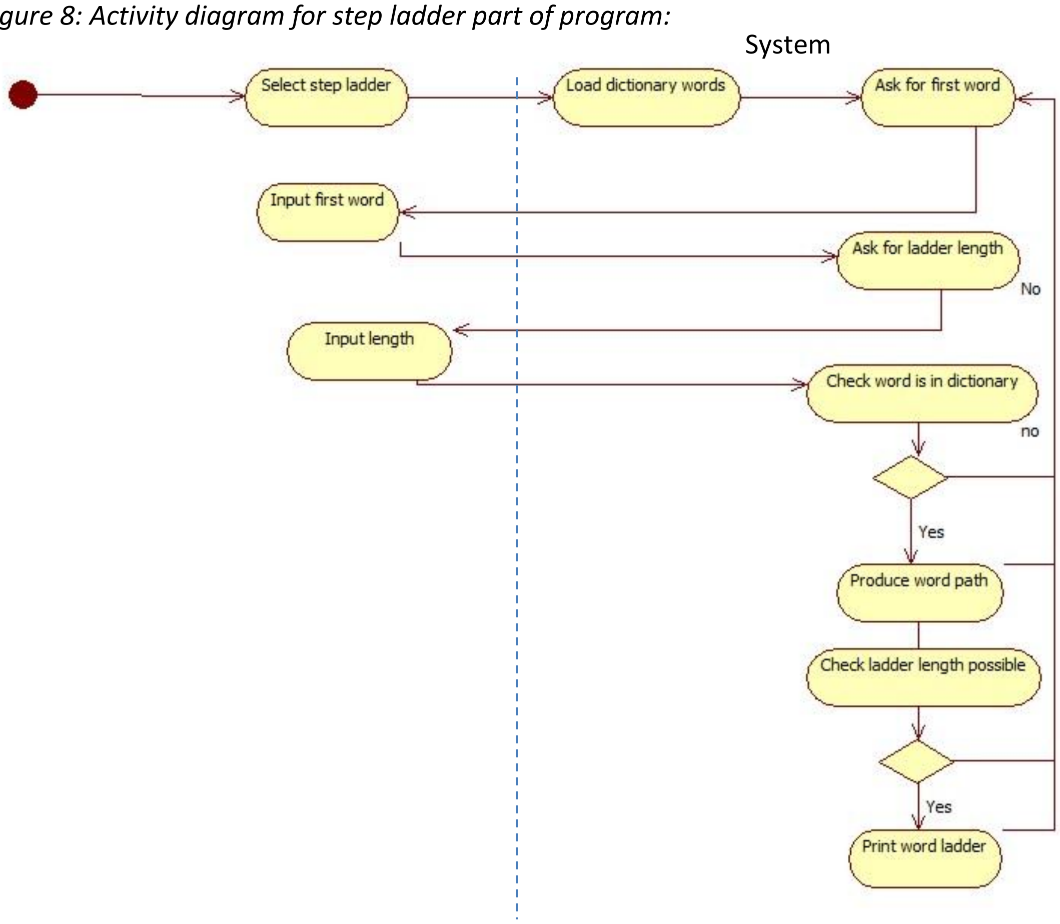
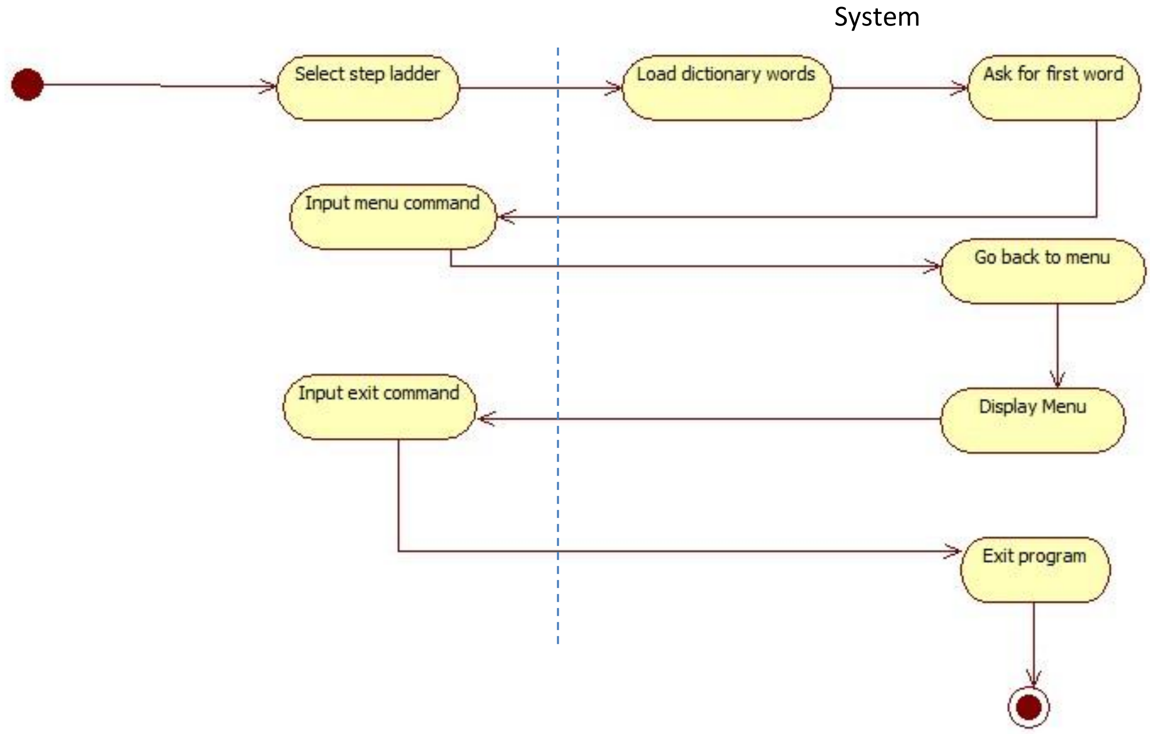


*Figure 9: Activity diagram for closing and using the menu:*




The penultimate section of my UML diagrams consisted of three activity diagrams, again one for each of the different ladder game types, and one for navigating the menu and exiting the program. In figure 7, it shows the relationship between the system and the users when entering in two different words during the discovery section of the program. As shown in the figure, the user inputs the choice of word ladder, followed by the loading of the dictionary words. Then the system asks the user for a word to be followed by an input by the user. It then shows the repeat of this for the second word, followed by a check. They system checks if both are in the dictionary, and then decide what to do next. If the user did input valid words, it then shows moving onto the next stage which is producing the word path. Otherwise, the check would redirect to asking for the user to input the first word in.

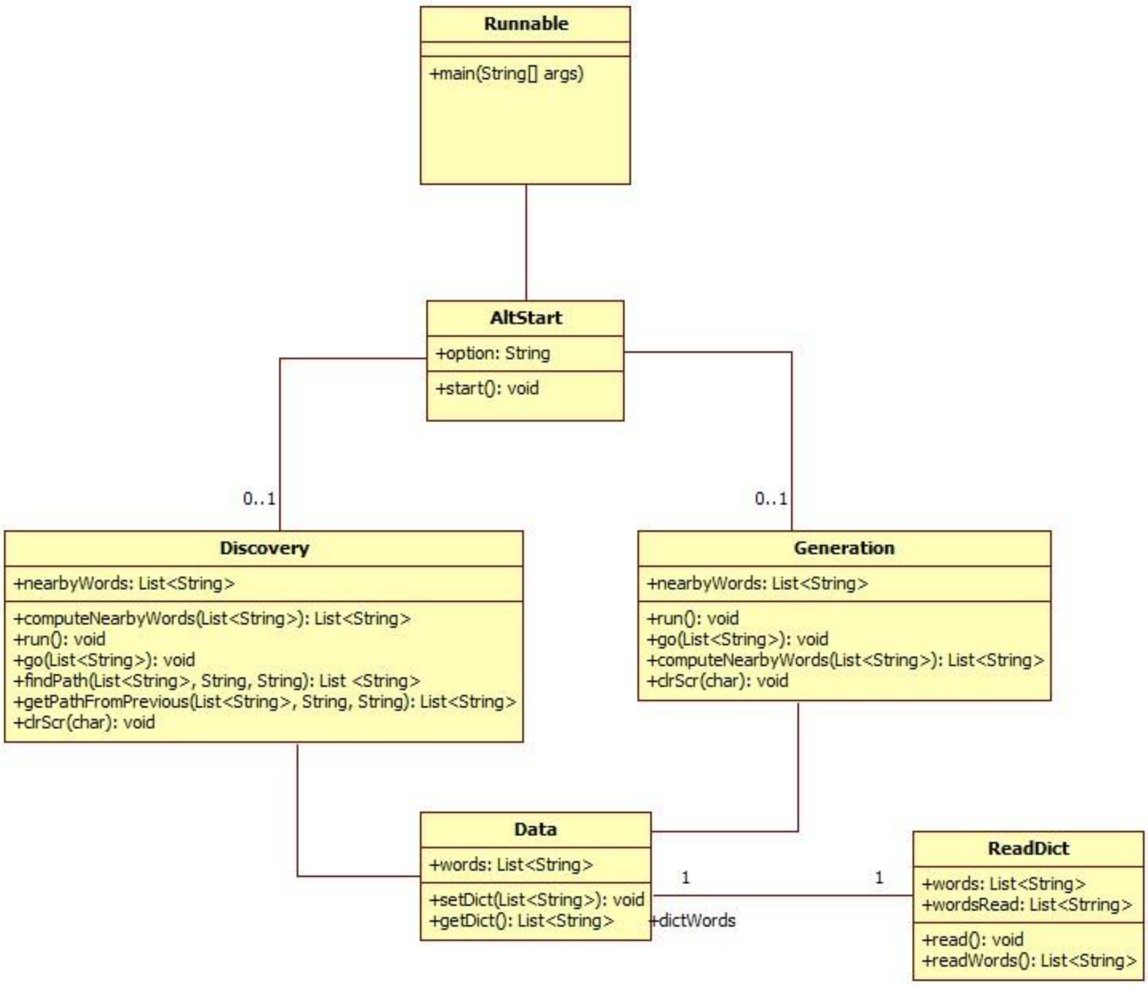
This is similarly defined in figure 8, where instead of asking for the second word, it then asks for the length input of the ladder. Here, it goes through the check of looking through the dictionary for the first word, and if successfully, then checks that the ladder length is possible for the word entered. If all is successful, it prints the word ladder. If either fail, it simply goes back to input of the first word.

In figure 9, this diagram contains no if statements, but just illustrates again the way the user can begin by choosing a ladder type game from the menu, and then by inputting the escape command to get back to the menu, followed by another command to exit the system. I have chosen not to include the link to each ladder from the menu within the diagram for simplicity and to show clearly the way out of the program. Activity diagrams are a very good tool in design and after choosing them I can see what the order of system-user interaction should appear like.

## Class Designs

As part of my final UML design section, I will now create a class diagram to show how my program will be divided into classes, and the methods that operate within the program. It will provide me with a basic skeleton of the program allowing me to fill in the methods.

*Figure 10: Class diagram:*




## Class descriptors

In figure 10 displayed above, I have created a program class diagram to fit the needs of my system and I will now provide some descriptors of each of the classes within the figure. This section is the most important of the UML as it gives a good idea of what methods I will be using to complete the required brief. As mentioned earlier, it also provides me with some 'skeleton code' for the implementation

**Runnable class**

This will be the class with the main method of the program in it, which starts the AltStart (alternative start) class. It main purpose is that it starts the program and by placing the inputs and menu in another class, it can be repeatedly called when users exit to the menu from either ladder class.

**AltStart class**

This class which is called from both the main runnable class at program start up and both ladder classes contains the main navigational unit of the system which asks for the user to select a ladder to use or the other option to exit. As seen from the figure, users insert their option and this is used a string to decide which is selected. It creates a new instance of either a Discovery or Generation class.

**Data class**

The data class as illustrated within the class diagram has a basic meaning of holding the dictionary list that comes from reading in the dictionary file. Its only attribute is the list of strings named words, and methods include the setter and getter for the dictionary list. The set method is implemented by the read dictionary class once the program has started up. Both the Generation and Discovery class will access the get method from the data class to get the dictionary list.

**ReadDict class**

This class will be responsible for getting the dictionary text file read into the system and then interpreting this as a list of strings. It will contain two attributes, one for the directly inputted value of the dictionary file, and one for the organised dictionary within the list of strings. Similarly, two methods will be implemented, one designed for the use of reading the words directly from the file, and then to read the files into the list. The readWords method will implement the setter of the dictionary words in the data class.

**Discovery class**

The first of the two ladder classes, this class will be responsible for creating the link of words between two given words from the user. As the only attribute of the class, the nearbyWords variable which will be a Hash Map of strings containing the list of words closely related after sorting. This attribute will be calculated during the computeNearbyWords method which will take in the words variable from the data class. It will then produce the Hash map to be used when finding the path between the words. Amongst this method, there will be firstly a main run method, which will introduce instructions for the user, and also run the computing of nearby words method. The ‘go’ method will be used in this class for getting user input, and also producing the output. This method will take in the dictionary words list. As for finding ladders, both the find path and getPathFromPrevious methods will work together to produce the path. The clear screen method will act as a way to clear the screen after an output is produced.

**Generation class**

The final class in the diagram is the Generation class which runs very similarly to the Discovery class except the way it will not require both the findPath and getPathFromPrevious methods in order to get a specific length ladder from one word.

## Description of Algorithm

Now I have shown the layout and required workings of my program in the planning section of this document, I will now present the algorithm (method my program will organise and calculate the route between words) I will use within my system. Shown below is a run through of how my program will do this:

*Figure 11: Algorithm run through*

***Get dictionary file from file system, place this into a List of Strings.***

***Then compute the words into a Hash Map that contains of words that differ by just one letter by:***

***Group the words by length, by creating a new key and a set of values in the map. (Key being the length integer, values being the words of that length)***

> ***Work on each group of words in the Hash Map:***
> ***For each string in a group of words, change one char, and creating a hash map value for each changed word***
>
> ***Then check each section within these maps whether they contain more than one string in. If so:***
>> ***For each string gotten from the map,***
>>> ***For each next string gained,***
>>>> ***Compare the two, if they are not the same, add them to a final map containing nearby words.***

***Then to get path using the nearby words map, and two inputs from user:***

***Create a new empty hash map to hold the previous_word in the path.***
***Create a new queue of strings, and add the first word in.***
***And while the queue is not empty,***

> ***Get the first item out of the queue, (remove this from the queue) and get the map associated with it in the nearby words map.***
>
> ***If this map list is not empty:***
>> ***For each string from the list:***
>>> ***If the previous_word map value for the certain string is empty:***
>>>> ***Create a new section in the map for this string as the key, and the value from the queue as the value in the section.***

***When queue becomes empty, overwrite the first key value in the map to null***

***Then, create the linked list which will represent the path between the two words***

***Check if the values in the previous_word map associated with the second word as the key is not null. If they are not null:***

***Create a new string (str) assigned to the second word inputted by user,***

***While it is not null,***

***Assign it to the value in the previous_word map using the str as the key***

***And add this to the front of the linked list representing the result of the ladder***

***Return the resulting path.***

# Implementation

Now I have completed the design section of the project, I can implement the system as I have planned. Within this part of the document, I will provide the printed code along with any required comments complete within a JavaDoc document. This will provide a more clear to read program and an explanation of how to complete program works.

*Addition: Added a graphical user interface to the program allowing users to use the program easier and provide a better looking system. I have added and completed the JavaDoc for these classes.*

## Printed new and changed java code with comments

*See appendix 1 & 2 for printed java code with JavaDoc.*

# Testing

The testing of my program is as follows. I have used JUnit testing for this part of the project where possible, and included this within the source files of the project. JUnit was a good choice for some of the classes and methods involved within the system, as I found that it gave good clarification that data was being passed around the program correctly as it should. For instance, in the data class, which handled the dictionary list of words needed to calculate paths, I had to ensure that the words were being saved properly and reliably for when the ladder classes were using the get () method to pull out the information.

I also chose to create a JUnit test suite within my testing package of the source, allowing me to run all my tests at once and ensure ALL the system worked as intended. From my JUnit testing criterion, I found that they all passed, so as extra test material, I performed a full test of the system in terms of input within the system using a test plan and table as shown on the next table. This has meant I could test extreme and boundary data along with other types in sections such as inputting the integer number of ladders in the generation section of the program.

## Test plan and test table

This test plan outlines the tests I am to perform on the system in all the elements appropriate. There will be a test number for reference, the name of the test element, along with a description to say how I am testing and an expected result. I will be able to then compare this to the actual result in the test table following this.

*Figure 12: Test plan:*

| Test Number | Test Element | Test Description | Expected Result |
|---|---|---|---|
| 1 | Menu selection | Input “1” into the menu to select Discovery | Should load up the discovery section of system |
| 2 | | Input “2” into menu to select Generation | Should load up generation section of system |
| 3 | | Input “x” into menu to exit | Should exit program |
| 4 | | Input “X” into menu to exit | Should exit program |
| 5 | | Input “3” into menu | Should do nothing and repeat instruction |
| 6 | Generation inputs | Input valid word “test” and length “3” into generation | Should accept inputs and produce appropriate path |
| 7 | | Input valid word “test” and length “1000” into generation | Should accept inputs and attempt to produce path. Tell user not long enough if it cannot find word ladder |
| 8 | | Input invalid word “t” and length “3” into generation | Should tell user word not in dictionary and wait for new input |
| 9 | | Input valid word “test” and invalid length “t” into generation | Should inform user of invalid integer input and wait for new input |
| 10 | | Input “x” to exit to the menu | Should take user back to the menu |
| 11 | | Input “X” to exit to the menu | Should take user back to the menu |
| 12 | | Input invalid word “(empty string)” and length “3” into generation | Should tell user work is not in dictionary and wait for new input |
| 13 | | Input valid word “test” and length “0” into generation | Should tell user of invalid input of length |
| 14 | | Input valid word “test” and length “1” into generation | Should produce path of just the word |
| 15 | | Input valid word “test” and length “1.5” into generation | Should tell user of invalid input of length and await new input |

| Test Number | Test Element | Test Description | Expected Result |
|---|---|---|---|
| 16 | Discovery inputs | Input valid words “test” and “belt” into discovery | Should accept inputs and create appropriate path between the words |
| 17 | | Input valid word “test” and invalid word “t” into discovery | Should accept first and then tell user second word is invalid and await new input |
| 18 | | Input invalid word “t” and valid word “test” into discovery | Should tell user first input is wrong before getting to the second input, then await new input |
| 19 | | Input invalid word “(empty)” and valid word “test” into discovery | Should tell user first input is wrong and await for new input |
| 20 | | Input word “short” and “longer” into discovery | Should tell user inputs are different length and await new input |
| 21 | | Input word “unique” and “number” into discovery | Should tell user that It could not bridge the words and await new input |
| 22 | | Input “x” to go back to menu | Should take user back to menu |
| 23 | | Input “X” to go back to menu | Should take user back to menu |
| 24 | | Input word “car” and “rap” into discovery | Should accept inputs and create appropriate path between the words |
| 25 | | Input word “traps” and “cries” into discovery | Should accept inputs and create appropriate path between the words |

Now I have created the test plan for my system testing, I can now test the system directly for the actual results to see whether they did as predicted and passed, or if they did anything unexpected and failed. Any failed tests will be placed into the revised tests section following this to explain why they failed if they did.

*Figure 13: Test table:*

| Test Number | Test Element | Test Description | Expected Result | Actual Result | Pass/ Fail? |
|---|---|---|---|---|---|
| 1 | Menu selection | Input “1” into the menu to select Discovery | Should load up the discovery section of system | Loaded up discovery | Pass |
| 2 | | Input “2” into menu to select Generation | Should load up generation section of system | Loaded up generation | Pass |
| 3 | | Input “x” into menu to exit | Should exit program | Exited the system | Pass |
| 4 | | Input “X” into menu to exit | Should exit program | Did not exit, asked again | Fail- See Revised |
| 5 | | Input “3” into menu | Should do nothing and repeat instruction | Told of invalid input, asked for new input | Pass |
| 6 | Generation inputs | Input valid word “test” and length “3” into generation | Should accept inputs and produce appropriate path | Created and printed out the path | Pass |
| 7 | | Input valid word “test” and length “1000” into generation | Should accept inputs and attempt to produce path. Tell user not long enough if it cannot find word ladder | Told that path could not be made long enough | Pass |
| 8 | | Input invalid word “t” and length “3” into generation | Should tell user word not in dictionary and wait for new input | Told that word was not in dictionary and asked for new input | Pass |
| 9 | | Input valid word “test” and invalid length “t” into generation | Should inform user of invalid integer input and wait for new input | System terminated and produced exception | Fail- See Revised |
| 10 | | Input “x” to exit to the menu | Should take user back to the menu | Printed out the menu | Pass |
| 11 | | Input “X” to exit to the menu | Should take user back to the menu | Told word was not in dictionary and asked for input again | Fail- See Revised |
| 12 | | Input invalid word “(empty string)” and length “3” into generation | Should tell user work is not in dictionary and wait for new input | Told that word was not in dictionary and re-asked for input | Pass |
| 13 | | Input valid word “test” and length “0” into generation | Should tell user of invalid input of length | System terminated and produced exception | Fail- See Revised |

| | | | | | |
|---|---|---|---|---|---|
| 14 | | Input valid word “test” and length “1” into generation | Should produce path of just the users word | Printed out the ladder of just the word inputted | Pass |
| 15 | | Input valid word “test” and length “1.5” into generation | Should tell user of invalid input of length and await new input | System terminated and produced exception | Fail- See Revised |
| 16 | | Input valid words “test” and “belt” into discovery | Should accept inputs and create appropriate path | Accepted inputs and produced the path | Pass |
| 17 | Discovery inputs | Input valid word “test” and invalid word “t” into discovery | Should accept first and then tell user second word is invalid and await new input | Accepted the first word and told that second word was not in dictionary and re-input | Pass |
| 18 | | Input invalid word “t” and valid word “test” into discovery | Should tell user first input is wrong before getting to the second input, then await new input | Told first word was not in dictionary and asked for a new input. | Pass |
| 19 | | Input invalid word “(empty)” and valid word “test” into discovery | Should tell user first input is wrong and await for new input | Told first word was not in dictionary and asked for new input | Pass |
| 20 | | Input word “short” and “longer” into discovery | Should tell user inputs are different length and await new input | Told that words were different length and asked for new input | Pass |
| 21 | | Input word “unique” and “number” into discovery | Should tell user that It could not bridge the words and await new input | Attempt to calculate it anyway | Fail- See Revised |
| 22 | | Input “x” to go to menu | Should take user to menu | Taken back to the menu | Pass |
| 23 | | Input “X” to go back to menu | Should take user back to menu | Asked for new input after being told word was not in dictionary | Fail- See Revised |
| 24 | | Input word “car” and “rap” into discovery | Should accept inputs and create appropriate path between the words | Created correct path between words | Pass |
| 25 | | Input word “traps” and “cries” into discovery | Should accept inputs and create appropriate path | Created correct path between words | Pass |

## Revised tests

Now that I have completed the test plan and test table for the program, I can have a look at which tests failed, in a list of 'revised' tests. Below I have stated the tests that have failed from some of my inputs, including why I believe the tests failed, and how I came to fix the relevant code specific to the problem. Revising tests is vital to ensuring that the completed program has no errors or way the program could be broken by the user inputting invalid information, or general bugs happening. Below are the revised tests, which I have grouped together where appropriate:

**Test Fail- Tests 4/ 11/ 23 (Entering "X" but not exiting)**

In these set of tests, which in all I found that by entering "X" uppercase rather than lowercase did not exit the program, this was an unexpected but simple error in the code. To solve this simple issue, all that had to be done to my code was to add in within the 'if' statement that checking if the input was "x", to also check if it was "X" using an 'or' expression. I then ran this again and found no such problem when wanting to exit or go back to the menu inside the program.

**Test Fail- Tests 9/ 13/ 15 (Invalid length input)**

In this set of tests, the fail came from the inputting of the length integer within the generation section of the program and more specifically the section where the input read in was then converted from a string to an Integer. So after identifying this problem, all I simply had to do was add in a try catch block of code that firstly tried converting the string and in the catch block it would catch the exception caused by the invalid input, followed by setting the length to a default value of 0. This would then indicate to the system that the input was wrong, and it would now print a statement to the user that they inputted an invalid input and then re-ask for the input of the word and length.

**Test Fail- Test 21 (Could not find ladder)**

This was a simple print our error that I have now fixed. It was due to the print line saying the patch size was 0, rather than just saying that it could not find one. I easily fixed this by changing the print line in this section of code, resulting in the proper given error message now.

# Evaluation

In this, the final section of my project documentation, I will evaluate my completed program and overall project. I am going to look at how well I met the brief when it came to the system I had created for the client in terms of getting the word ladder working and overall how I came to perform this. Starting with the planning of this project, I will look at how I came about planning my system from the diagrams to the algorithm pseudo code I created. Following this I will look at the method of implementation I used, and how I tested the completed program. I will also be using this section to look at the possible improvements I believe could be made to my program, along with show changes if I actually implemented them.

## Evaluation of program

Before I look at the way I went about completed my program, I will firstly look at my completed system and how it meets the brief. By looking at a general run through of the system, I will be able to now look at how well I have met the requirements given at the start of the project. Starting by looking at the completed product and where I achieved the goal of its purpose, I can say that I have developed a system that allows users to choose from both generation and discovery options allowing the creating of 'word ladders'. In the discovery section, which I implemented first, I have created and used an efficient algorithm that calculates the routes between words reliably and also in the shortest possible ways. This applies the same with the generation section of the system, where I have created a reliable ladder creator allowing users to into a word from the dictionary, and requesting a ladder made to a length of their choice, although it has limitations of the length of generated word ladders. (I have included this note in my possible improvements next)

When it comes to any possible improvements in the system I have created, I can outline a few examples which I believe would make a generally better program if I had time to implement them, or if I ever created a program like this again knowing what I know now.

Looking at some possible improvements, which I will expand, could have included:

- Adding a graphical user interface to the program- adding this would be simple, and would make the program look more attractive.
- Moving the method that computes the list of nearby words to another class- would prevent duplicated code.
- In the generation section of the program, make it possible to create longer ladders of the inputted word- would add more possible use of the program.

I believe that if I had extra time, it would have been possible to make all of these improvements, and with the new skills I have picked up when completing my program, making my system easier to use, and with more capabilities.

## Evaluation of project

In the final section of the documentation for this Java based project, I will now look at the design, implementation and testing sections of this document and how I completed them. Starting with the design that followed the introduction, I found that by using a wide range of UML diagrams helped me explain and plan what the program layout should have been, and also helped greatly in defining the key methods and variables that would be used in calculating the ladders. Justifying each of these diagrams also provided good explanation into the reasons for creating the designs in such ways. If I could look at a way I could have improved this section of my project, it would be that I could have used a couple more diagrams to explain the plan better. When it came to the algorithm explanation, I felt that it explained in good detail of the steps taken by the program of how it would find ladders between words.

Once the planning section was complete, I then went onto implementing the projected system. Although I did not show how I implemented the system, the way I actually did this went very well. By getting the basic data Java files working first such as sorting the words from the dictionary into a hash map set up a good building base for sorting out the rest of the program.

Once the implementation was complete, I then went onto creating the JUnit tests for some of the data classes, mainly for the ones that held data was key to be tested and I felt that I went about this in an ideal way. Ensuring that the data held was reliably and conveniently held meant the rest of my program could grab the data easily for use in the calculation of the ladders. Along with the JUnit testing, I also ensured that the program met standards and did not have any errors by using a test plan and test table. This allowed me to test inputs such as boundary and extreme data.

# Appendices

## Appendix 1: Printed source code (Including JUnit)

## Appendix 2: JavaDoc