**Assignment 1: KWIC-KWAC-KWOC**

|  |
| --- |
| Code Repository URL: https://github.com/HyungJon/KWIC |

|  |  |  |
| --- | --- | --- |
| Name | Kim Hyung Jon | Jerrold Law |
| Matric number | A0115864B | A0097835M |

1. **Introduction**

This software is a simple program that generates a set of circularly shifted strings from input titles and words that are not keywords. It aims to meet the three non-functional requirements: user-friendliness, speed and extendibility.

A functional requirement set by our group was that the software should immediately display the results after the process is completed so that the user does not have to find and open it.

For both solutions, the data storage format was simplified to storing lines, titles and circular shifts as arraylists of individual words rather than strings and indices indicating the start of words, for faster execution.

**2. Design**

The program runs by executing the UI class, a component common for both solutions. Upon execution, UI displays a simple GUI containing 4 main parts: two text fields for specifying input and output file paths, and two buttons for selecting the solution. The UI component was implemented by Hyung Jon, in order to make it easy for users to learn the instruction.

Both methods require that the input file contains the data in the following format: the first line contains all words to ignore, and each of the following lines contains one title. An example is:

is the of and as a after

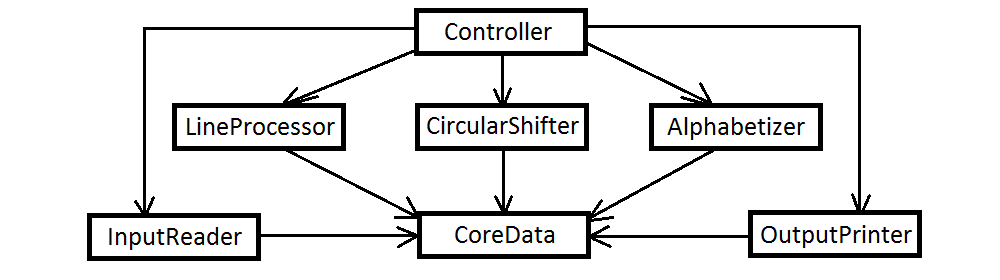
The Day after Tomorrow

Fast and Furious

Man of Steel

By clicking either button, the user can choose one of the following solutions to generate circular shifts. Solution 1 was implemented by Jerrold, while Solution 2 was implemented by Hyung Jon. As both solutions are very similar except for the difference in where the data is stored, some methods that can be used in common for both approaches were implemented by Hyung Jon as well for efficient development.

**2.1 Solution 1: Shared Data**



Solution 1 by Jerrold follows the Solution 1: Shared Data approach in Reading 2. It corresponds to the shared repository architecture and contains the following components:

**Controller**

Master control in charge of invoking the involved components in the correct order and storing the shared data.

**CoreData**

Component that stores the data produced by each involved component and offers methods for accessing and modifying the data.

**InputReader**

Reads the input file and stores the data line by line in CoreData, without the knowledge regarding what each line is.

**LineProcessor**

Converts the lines stored in CoreData and constructs titles and words to ignore. These are stored in separate arraylists in CoreData for CircularShifter to access.

**CircularShifter**

Constructs all possible circular shifts from each title stored in CoreData, not starting from words to ignore. It then stores the result back in CoreData.

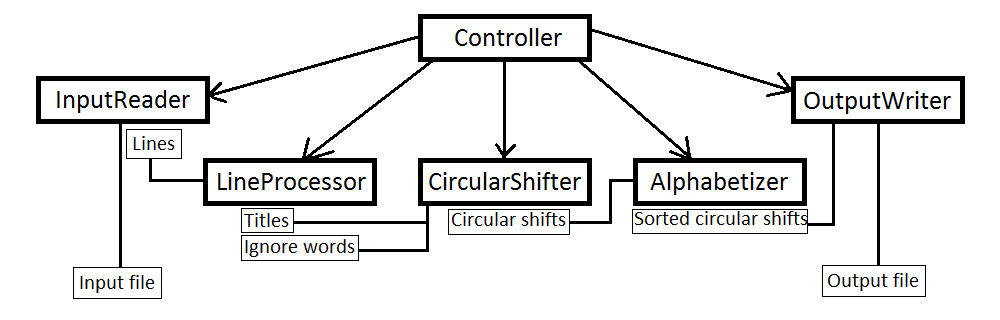
**Alphabetizer**

Sorts the circular shifts in CoreData in the alphabetical order.

**OutputWriter**

Writes the sorted circular shifts in CoreData to the output file specified by user and opens the output file.

**2.2 Solution 2: Abstract Data Types**

****

Solution 2 implemented by Hyung Jon is a simplified version of the Solution 2: Abstract Data approach in Reading 2. It corresponds to a pipes and filters architecture. It consists of the following components:

**Controller**

Master control in charge of invoking the involved components in the correct order with correct parameters.

**InputReader**

Reads the input file and stores the data line by line, without the knowledge regarding what each line is, and offers methods for LineProcessor to access individual lines.

**LineProcessor**

Takes all lines stored in InputReader and converts them to a form usable by CircularShifter by constructing two arraylists of strings: for titles and words to ignore.

Rather than storing all words in all lines as 2D arraylist, LineProcessor contains the method splitToArrayList() that converts a title at given index to an arraylist of words, invoked by CircularShifter only when constructing circular shifts of that title.

**CircularShifter**

Constructs and stores all circular shifts, not starting with words to ignore and with first word in upper case, from each title obtained from LineProcessor. It contains methods for Alphabetizer to access all circular shifts.

**Alphabetizer**

Sorts and stores the circular shifts generated by CircularShifter in the alphabetical order.

**OutputWriter**

Writes all sorted circular shifts sorted by Alphabetizer to the output file specified by user and opens the output file.

**3. Comparison**

**Comprehensibility**

Solution 1 is the naturally intuitive approach to generating circular shifts from titles. Therefore, it is reasonably easy to comprehend the process of the entire system, although this advantage may not remain true if the system is expanded greatly and the possible scenarios increase. Solution 2 is highly modularized, each component in charge of a specific purpose. Therefore each component requires only the knowledge relevant to that purpose. For example, LineProcessor is the only module that knows where to find words to ignore in the input. Therefore, in Solution 2 it is easier to comprehend the behavior of each component individually unlike Solution 1 which is comprehensible only as a whole.

**Modifiability**

Because the system is modularized in Solution 2, modifying the system requires modifying only the relevant components. For example, changing the data storage format to storing titles as strings rather than arraylist of words and generating circular shifts by splitting the titles at specified index and swapping the substrings will affect almost all components in Solution 1. Conversely, in Solution 2, only LineProcessor and CircularShifter need to be modified. Therefore it is much easier to make modifications, especially small-scale changes restricted to single components, in Solution 2.

**Extendibility**

In Solution 1, all components interact with the CoreData rather than with each other. Therefore it is relatively easy to add a new function that accesses the data without having to modify other modules. Conversely, this is relatively difficult in Solution 2 due to each component retrieving data directly from the previous component. For example, to add a component X between CircularShifter and Alphabetizer, both components must be modified so that CircularShifter converts the data into a form usable by X and Alphabetizer accepts data processed by X.

This is partially overcome by implementing the interfaces in each component to return data as widely understandable formats such as arraylist of words, rather than something that can be understood only in a particular component. For example, adding a component that modifies the list of titles before constructing circular shifts is reasonably easy since arraylist of strings is a widely understandable format. The only exception to this is the raw lines read from the input file, since only LineProcessor knows that the first line contains words to ignore and how they should be stored.