Assignment 1: KWIC

Repository URL:	https://github.com/TIC3001/tic3001-ay2122s2-13/releases/tag/Assignment1		
Team number:	13		
Date:	7 Mar 2022		
Name:	Roshan Kumar	Name:	Noel Lim Xian
Student Number:	A0094621H	Student Number:	A0211270Y

1. Introduction

Roshan Kumar – *Pipe & Filter (Section 3.1)* **|** Noel Lim - *Abstract Data Types and Objects (Section 3.2)*

During the requirements gathering phase, it was understood that the program must take in an input file, parse it into a format that's usable for the purpose of the exercise, and then run certain functions that include circular shifts and lexicographically sort it before writing the result out as an output file. Therefore, these requirements were broken down into certain boxes with an input / output filter provisioned within them.

2. Requirements Analysis

2.1 Functional Requirements

- As a user, I want to point to an input file so that the program can take in the input's file data and make sense of it.
- As a user, I want the system to be able to do circular shifting of each line item so that it can produce a list of possible sentence structures for each line item.
- As a user, I want the system to be able to collect the list of possible sentence structures from each original line item and combine them into a single list so that I can sort them later.
- As a user, I want the system to be able to perform sorting in alphabetical order first using the first word of a sentence, and if there are similar first words, compare the second word so that it produces an accurate list of sorted sentences as per alphabetical order.
- As a user, I want to write the result of the sorted list as an output file so that I can compare it to the expected result to verify correctness of the program.

2.2 Non-Functional Requirements

- Upon start of execution of program, total time taken for output file to be produced should be less than 2 secs.
- Program should be able to take a flexible list of input sentences (E.g., 1 – 100 sentences) and should work normally.
- Program should be able to accept special characters as input data and be able to sort it accordingly.
- The source file and its corresponding result file should be easily identifiable. This is achieved by placing each source into separate folder. After processing the source, the application will store the result in the same folder as the source file with the filename as source file name appended with "-output".

Example - Test Folder Stucture

```
`-- sample_test_cases
            |-- set1
            |   |-- source.txt
            |   `-- source-output.txt
```

3. Designs

3.1 Pipe & Filter

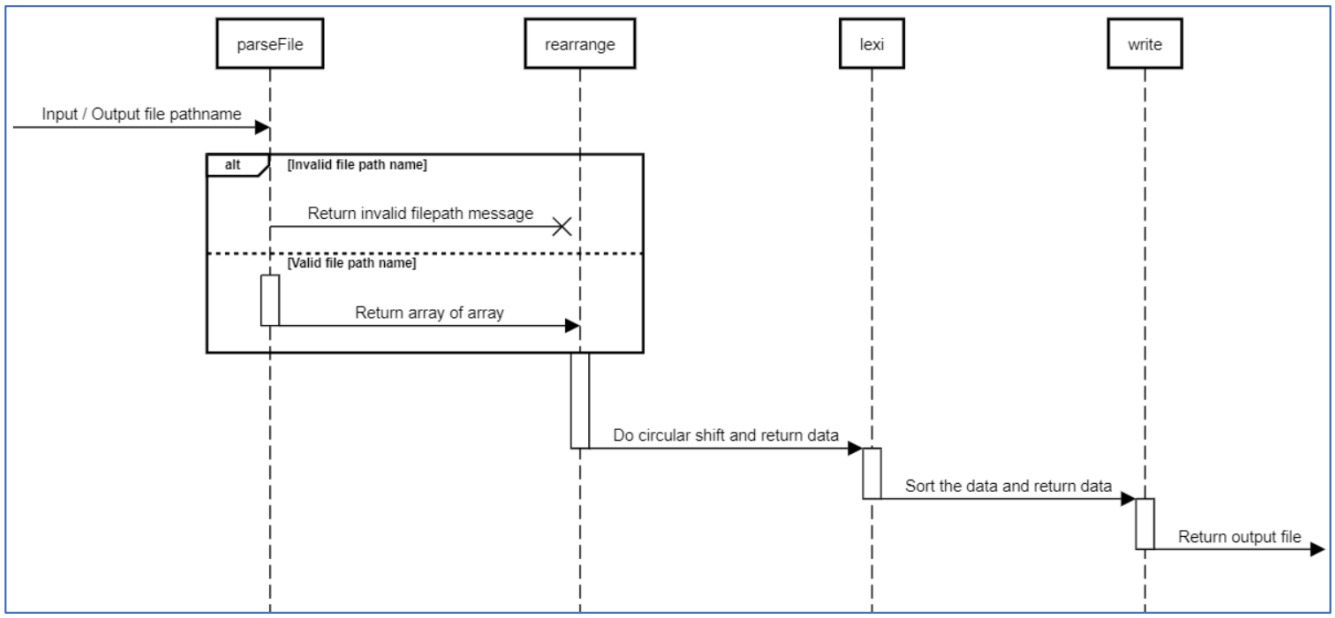
3.1.1 Architectural Design

This design takes in an input file and parses it, and then sequentially uses a function to output some data which is then used as an input for another function which then produces another output. This process is repeated till it reaches the last function, where the final output is an output file containing some processes data.

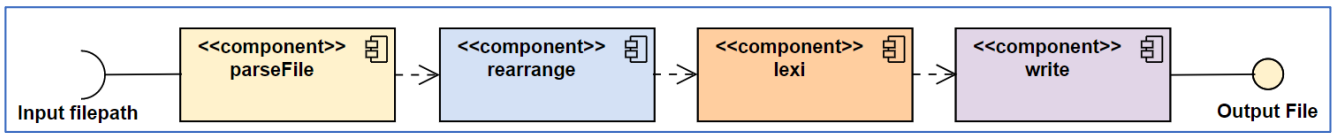
Once input data is provided (Input / Output file path) and program is run, the following occurs:

[parseFile] Takes in input file data and processes the data as an array of array
[rearrange] Takes in data and for each array, it produces a list of arrays using circular shift
[lexi] Takes in data and performs alphabetical sorting
[write] Takes in data and writes data into an output file in a particular format

Sequence Diagram



Component Diagram



3.2 Abstract Data Types and Objects

3.2.1 Definitions

Word

A word is an ordered set of characters.

Line

A line is an ordered set of space-separated words.

Lines

Lines are set of lines delimited by a line break as shown in Example 1.

Derivatives of Lines

Source: Lines as input to generate concordance.

Concordance: Generated line(s) from source. For every line in source, the words will be rotated by shifting the first word to the end of the line. The rotation will repeat to create a set of distinct lines. The sequence of combinations progresses naturally, ordered by line position in source followed by number of shifts operated on the line. Example 2 below shows Concordance of Example 1.

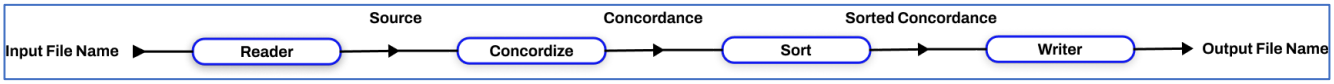
Sorted Concordance: Concordance sorted lexicographically (case-insensitive). Example 3 shows Example 2 after sorting.

Example 1	Example 2	Example 3

`10 Cloudy`
`9 Harvey`
	`10 Cloudy`
	`Cloudy 10`
	`9 Harvey`
	`Harvey 9`
		`9 Harvey`
		`10 Cloudy`
		`Cloudy 10`
		`Harvey 9`

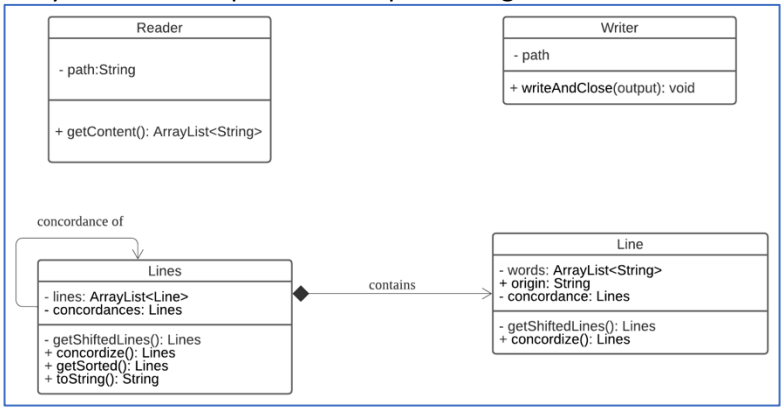
3.1.2 Workflow

1) Reader accepts a file name and reads the file content as source text.
2) Concordance is populated from the source.
3) Concordance is sorted.
4) Writer buffers concordance into an output file.

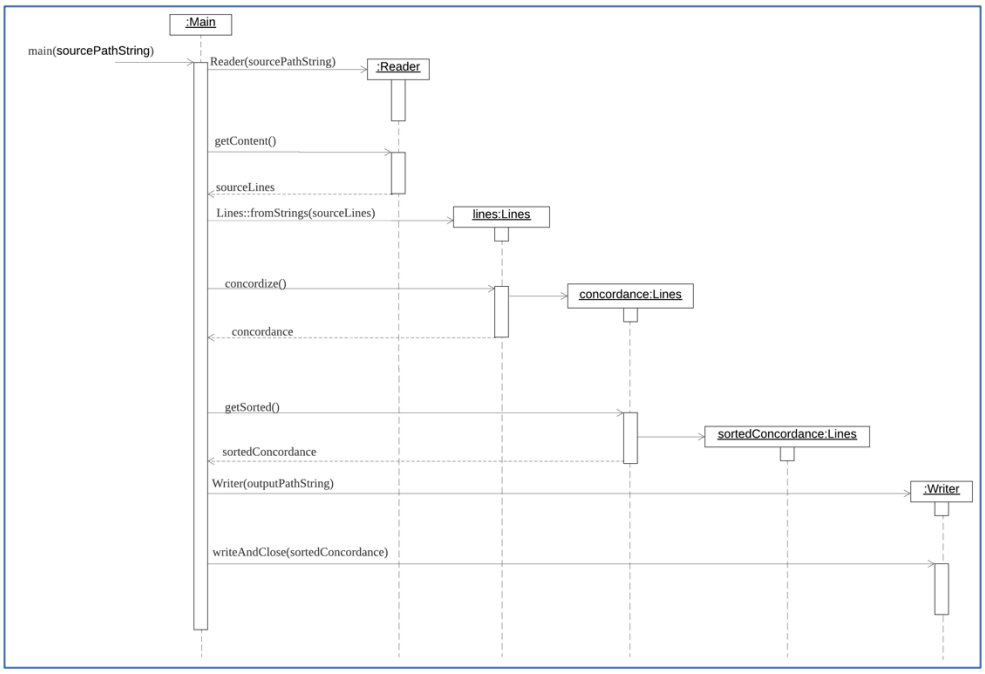


Class Diagram

Reader: Reads from file and parse content as Java object.
Writer: Buffers result into output file.
Lines: Representation of an array of Line and its related behaviors.
Line: Representation of a Line and its related behaviors. A line is stores as both a contiguous string and array of words. This allows flexibility for data manipulation and processing of lines to fulfill functional requirements.



Sequence Diagram



4. Limitation & Benefits of Selected Designs

4.1 Pipe & Filter

Benefits

1) Maintains the intuitive flow of processing
2) Allows for loose and flexible coupling of filters
3) New functions can easily be inserted into the pipe at appropriate points
4) Allows for ease of modification as each filter is logically independent of other filters

Limitations

1) Many independent filters will reduce performance due to excessive computation
2) Inefficient use of space as each filter must copy all its data to its output ports.
3) Not possible to modify the design to support an interactive design

4.2 Abstract Data Types and Objects

Benefits

1) Rich data representation allows indexing, construct composite data and access abstract data properties.
2) Enables development of custom and specialized operations with efficiency (delete particular line / modify word by reference, line deduplication) by adding methods to class components.
3) Centralized control. The main block as shown in the sequence diagram orchestrates sequence of operation.
4) Clear method interfaces separate implementation concerns from caller.
5) Reusability of class components – Each sequence of action can leverage on the common methods and data representation in the Lines and Line components.

Limitations

1) Tight coupling with underlying data classes.
2) Design opacity – Code tracing and readability may be harder as operations are chained from multiple service layers.

4.3 Comparison between Designs

In summary, we find that the main benefit of pipe and filter is data flows unidirectionally and clear demarcation of responsibility of independent pipe/filter component with operations, object and data states confined to its environment. However, mem-write performance is expensive as each pipe/filter generates a new set of data for the next pipe/filter. The design caters for functionalities which requires no computational side effects.

For ADT, class instances once initiated can persist until the end of the program lifetime. This is advantageous if we would want to work on intermediate data across multiple events. The downside for ADT is with multiple actors, it is difficult to keep track when a object state is mutated. The design is good for submodules with strong interdependencies as ADT helps to group contextual data and behaviors into components and enforces communication via interfaces.

5. Contributions

Roshan Kumar – Contributed towards the complete coding of the pipe & filter implementation, and the underlying benefits / limitations of pipe & filter, as well as how it is clearly distinguishable from the object-oriented architectural design.

Noel – Contributed towards the complete coding of the ADT implementation, and the underlying benefits / limitations of ADT, as well as how it is clearly distinguishable from pipe & filter implementation.