Assignment 2.1
KWIC Implementation

Pawan Bhandari

May 18, 2025

1 Introduction

This report assesses the data flow architectural style for the implementation of KWIC (Key Word in Context) and compares it to the shared memory implementation style presented in the paper "An Introduction to Software Architecture" [1].
KWIC index system takes text as an input and circularly shifts all lines (sentences) by removing the first word and appending it to the end of the line. Resulting lines are alphabetically sorted and output so that each word can be searched with the contextual information of the line containing the word [1].

2 KWIC Implementation: Data-flow (pipes and filters)

Pipes and filter style of implementation uses a set of filters which can be chained such that the input data is processed incrementally to produce an output. Each filter is independent of other and do not share any state/memory with other filter. This means that the input and transformed data must be transmitted throughout every filter (usually as pipes or via filter chaining) to produce the output. This codebase contains the data flow style implementation of KWIC indexing in Java. Text input is parsed as a list of sentences (lines) which is then passed to a CircularShifter. The output of the circular shifter is fed into the Alphabetizer and the result is printed to the console. The code snippets for each filter and the main method are presented below.

```
    public class InputParser{
    public List<String> parseInput(String input) {
        return Arrays.asList(input.split("\\."));
    }
}
```

```
    public class CircularShifter implements Filter<List<String>>{
    @Override
    public List<String> execute(List<String> inputLines) {
        List<String> shiftedLines = new ArrayList<String>();
        for (String inputLine : inputLines) {
            String[] words = inputLine.split(" ");
            for(int i = 0; i < words.length; i++) {
                String firstPart = String.join(" ",
                    Arrays.copyOfRange(words, i, words.length));
                String secondPart = String.join(" ",
                    Arrays.copyOfRange(words, 0, i));
                shiftedLines.add(String.join(" ", firstPart, secondPart).strip());
            }
        }
        return shiftedLines;
    }
}

    public class Alphabetizer implements Filter<List<String>>{
    @Override
    public List<String> execute(List<String> input) {
        input.sort(String.CASE_INSENSITIVE_ORDER);
        return input;
    }
}

    public class Main {
    public static void main(String[] args) {
        InputParser inputParser = new InputParser();
        CircularShifter circularShifter = new CircularShifter();
        Alphabetizer alphabetizer = new Alphabetizer();

        String inputText = "The Key Word in Context index system accepts an
        ordered set of lines. Each line is an ordered set of words,
        and each word is an ordered set of characters.";

        // Input
        List<String> input = inputParser.parseInput(inputText);
        // Shift
        List<String> shiftedInput = circularShifter.execute(input);

        // Alphabetize
        List<String> alphabetized = alphabetizer.execute(shiftedInput);

        // Output
        printOutput(alphabetized);
    }

    public static void printOutput(List<String> output) {
    System.out.println("************************KWIC************************");
        for (String s : output) {
            System.out.println(s);
        }
    }
}
```

3 Assessment of Data-Flow implementation and comparison with shared memory implementation style

Data flow implementation style is intuitive and easy to follow as the filters/processors are chained in the order: InputParser, CircularShifter, Alphabetizer, and Output. Each of these components are logically independent and can be updated as long as its input and output data formats remain unchanged. This also provides the flexibility of introducing additional filters to process the data that is being passed. This also allows for reusing the filters as long as the structure of the data in and out of the filter is same. Furthermore, this approach is also suitable for concurrent execution by implementing each filter as a separate task. The main drawback of this style become evident when there is a requirement for interactive applications and data needs to be output incrementally (for displaying). Since each filter is logically independent of other and requires a complete input in pre-specified format, the pipe and filter approach processes the data in batch which limits the ability to process data incrementally.
Data Flow style is also memory intensive as the data needs to be transmitted throughout the filters. The shared memory style of implementation uses less memory as compared to the data flow style and can benefit from increased performance. Both, Shared memory & data-flow style allow for changes in processing logic with ease since new components/filters can be introduced which can access the same shared data in memory (in case of shared memory), and new filters can be chained (in case of data-flow) because every filter is logically independent of each other.

References

[1] David Garlan and Mary Shaw. “An Introduction to Software Architecture”. In: *Advances in Software Engineering and Knowledge Engineering* I (Jan. 1994).