Assignment 1: KWIC-KWAC-KWOC

Code Repository URL: https://github.com/Whisperingroad/Software-Engineering-Principles-and-Patterns

Name	**Chew Yi Xiu**	**Sebastian Wong Zhi Qian**
Matriculation Number	A0101742W	A0101856J

Contents

1. Introduction

In this report, we will be describing two different architectural designs that our team has decided to implement in order to solve the Key Word In Context Problem. The Key Word in Context Problem proposed by Parnas is a well-known teaching device used to introduce and highlight the relative advantages and disadvantages of different software architectural designs. For this project, we have decided to focus on the Abstract Data Type Design (ADT) and the Implicit Invocation Architecture Design. The rationale behind our selection is because the ADT architecture design has long been accepted to be one of the most basic and effective approach to managing complexity in large scale software systems. On the other hand, the Implicit Invocation Architecture Design is an interesting and alternative integration technique that we have not been exposed to. This report will detail some of the strengths and weaknesses of our selected architectural designs as well as the modifications we have made to improve the system.

2. Design

Abstract Data Types

This original Abstract Data Types architecture proposed by Parnas involves dividing the system into 5 main components, namely, the MasterController, Input, Circular Shift, Alphabetic Sort, and Outputs. In my implementation, I have altered the architecture slightly by including an additional Storage component. The rationale behind this addition is to decompose the program into modules with distinct features so that their functionalities do not overlap. For instance, the Input module should only be responsible for reading the user input and not storing the data. This is in compliance with the Separation of Concerns (SoC) design principle as well as the Single Responsibility Principle.

In this ADTs architecture, the input data is processed in five basic steps. The MasterController controls the 6 components mentioned above and sequences them in turn. Firstly, the Input module will read in the list of titles and words to ignore from two separate text files. The Storage module will then keep the data read by the Input module. Each individual title will be retrieved from the Storage module by the Circular Shift module and is shifted accordingly. This essentially creates a list of circularly shifted titles which is passed to the Alphabetic Sort module which arranges the titles in ascending alphabetical order. Finally, the ordered titles will be passed to the Output module and displayed to the user.

Implicit Invocation

In the Implicit Invocation architecture, there are 5 main modules in the system, namely MasterControl, Input, Shift, Alphabetizer and Output. Processes in this architecture are data driven and the interactions between the different modules are based on a active data model.

In this implementation, instead of using a shared repository, I have altered the architecture such that modules have their own data storage. The rationale for this modification is to prevent all of the modules from being affected by changes in the data representation format. In addition, add and delete functions are supported. Users can add titles by specifying the filename(e.g. add titles.txt) or add a individual title (e.g. add Fast and Furious). Users can also delete a title by specifying the original title (delete Fast and Furious). In this case, all shifted permutations of the original title will be deleted. If the user specifies a permutated title (e.g. Furious Fast and), only the permutated title will be deleted.

MasterControl class acts a facade between the UI class and the other logical components in the system. The UI class can only access the logical components through MasterControl, which hides the implementation of the other 4 modules.

When a user chooses to input either a text file or a single title, the Input module will parse the inputs line by line. Assuming that the user chooses to add a text file containing a list of titles, this module will begin by reading the first title. This action will in turn trigger the Shift module, which will shift the title accordingly with respect to the list of ignore words. Upon the completion of this shift action, the shifted title will be passed to the Alphabetizer, triggering the insertion of the shifted title into a list which will be sorted in an ascending alphabetical order. Upon the completion of the alphabetical sort, the sorted results will be passed to the Output module.

To summarize this architecture, the insert or delete functions in all modules are invoked whenever the user enters an add or delete command. These add or delete procedures are called from one module to another sequentially. For

instance, when a user wants to insert titles to the KWIC system, the MasterControl module will invoke the insert procedure in the Input module, which will invoke the insert procedure in the Shift module, which invokes the insert procedure in the Alphabetizer module. Therefore, when a user command is given, all procedures that are tied to the command will be executed. In addition, this architecture parses the input given by the user line by line, doing computations and modifying data simultaneously, making this architecture a active data model.

3. Limitation & Benefits of Selected Design

Strengths of Abstract Data Type

The strengths in the ADT architecture lies in its flexibility and ability to support design changes. It is possible to change both the data representation format and the processing algorithm in an individual module without having to change other components. For instance, if the programmer decides to change the algorithm used to shift the titles in a circular manner, this change can be accommodated simply by altering the Circular Shift module. Both the Storage and Alphabetic Sort modules will remain unaffected.

In addition, this architecture also supports the reusability of existing classes and objects as individual modules make few assumptions about the modules that they are interacting with. This in turn leads to looser coupling.

Weaknesses of Abstract Data Type

Despite the advantages discussed above, there are a few weaknesses in this architecture. First and foremost, the most noticeable disadvantage is that it is more difficult and troublesome to enhance this solution with different functionalities. According to Garlan, Kaiser, and Notkin, it is challenging to implement logically independent requirements without interlacing logically independent implementations. This can be seen from the enhancement requirement in the KWIC to remove lines starting with words in the ignore list. In principle, the decision to include or exclude a line should not affect the implementation of the Circular Shift module, however, it is still implemented in

the Circular shift component for the sake of efficiency. This results in a compromise in the simplicity of the Circular shift module.

Secondly, as the data is not shared across different modules, this solution requires more space and is slightly less efficient as access through interfaces may be slightly slower.

Lastly, in order for an object to interact with another in this architecture, it must first know the identity of the object. This is in contrast to the Pipes and Filters architecture, in which individual filters do not need to know about the existence of other filters in the system. This implies that if the identity of an object is to change, changes will have to be made to every other module that explicitly invokes the object. Although this disadvantage is not evident in this particular solution, it will be an issue if the program is larger.

Strengths of Implicit Invocation

One of the main advantage of the Implicit Invocation architecture is that it allows for functional improvements to be added to the KWIC system easily. This can be achieved by implementing additional modules in the system and registering them to be invoked on data changing events. By doing so, functional enhancements can be integrated to the system without affecting other existing modules significantly.

Secondly, changes in the algorithm in one module is unlikely to affect other modules in the system as components are designed to be depend only on the existence of certain data changing events on another module. Therefore, changes in the algorithm does not matter as long as the inputs and outputs of the altered module remains unchanged. This also implies that modules can be reused as long as certain data changing events are present.

Weaknesses of Implicit Invocation

The primary disadvantage of this architecture stems from the fact that it is difficult to control the order of processing as modules are implicitly invoked. Modules cannot be explicitly invoked as the order of processing is not known by each modules. For instance, the print function cannot be executed directly from the Output module after completing a insertion or deletion. This is because the

Output module does not know if the Input module has read and processed all of the user input since inputs are processed line by line and the Output module, being implicitly invoked by the Alphabetizer module, does not know about the existence of the Input module. As a result, the Input module has to have a print event to trigger the other modules which will then invoke the print function in Output implicitly.

As each module has its own data storage, this architecture requires more memory space as compared to a shared data architecture. This can become a serious problem if the data input is very large as larger storage would be needed which takes up a significant portion of memory.

A change in data representation will result in some modifications to the module Output but it is not as significant as compared to a shared data repository. This is due to the fact that all modules are not accessing a shared storage but rather they have their own data storage to rely on.

Conclusion and Insights

Although both architectural designs both have their relative advantages and disadvantages, we think that the Abstract Data architecture is better in supporting changes in data representations as other modules in the system are unlikely to be affected. In addition, as the modules in Abstract Data Type are very well defined and modularized, we think that it can be reused easily. On the other hand, the Implicit Invocation Architecture is much superior as compared to Abstract Data Type Architecture in terms of adding improvements as it offers programmers the ability to add functional enhancements to an existing system without compromising on the simplicity and performance of the original modules. In conclusion, although we favour the Abstract Data Type Architecture due to its intuitive decomposition and effectiveness in managing complexity, the decision to choose either design will still depend on the problem in hand.