Contents

[1 Introduction 3](#_Toc88460674)

[2 KWIC Project 3](#_Toc88460675)

[3 Evaluating the Architectural Styles 4](#_Toc88460676)

[4 Total Points and Due Date 4](#_Toc88460677)

[5 Identification of Problem 5](#_Toc88460678)

[6 Make or Break Issues Driving Design Decisions 5](#_Toc88460679)

[7 Significant Features of the Architectures 6](#_Toc88460680)

[Main program and subroutines 6](#_Toc88460681)

[Object-oriented style 8](#_Toc88460682)

[Pipe-and-filter 9](#_Toc88460683)

[8 Compare Alternatives-Strengths and Weaknesses 10](#_Toc88460684)

[Main program and subroutines 10](#_Toc88460685)

[Object-oriented style 10](#_Toc88460686)

[Pipe-and-filter 10](#_Toc88460687)

[Conclusions 11](#_Toc88460688)

[Performance Comparison: 12](#_Toc88460689)

[Summary 13](#_Toc88460690)

# Introduction

This project will give you an opportunity to examine and describe the possible architectural styles (patterns) for a software system. You will develop at least three defensible designs, and analyze the relative merits of each. Note that all designs should be reasonable; do not propose a lousy design for the sake of making the others look good. Style alternatives are as follows:

**Call-and-Return** Traditional functional decomposition with a controller main program that calls certain sub-modules. Sub-modules share data.

**Call-and-Return** The modern object-oriented (or ADT-based) approach that allows information hiding (to hide design decisions and data representations when possible).

**Data Flow** The pipes and filters approach where each component performs an incremental transformation of the data. The transformations are combined so that outputs of certain transformation serve as inputs to other. The entire program is then the composition of these computations/transformations. Constraints:

1. Filters must be independent entities that do not share state with other filters.
2. Filters do not know the identity of upstream or downstream filters.

Examine the problem statement for the KWIC system (briefly described below). Think carefully about the problem and identify problem the actually being solved. Express the problem in diagrams when possible. Make sure that you have identified the make or break issue that will drive your design choices.

Based on what you know about the architectural level design, propose three architectural designs that meet the requirements of the KWIC project. Discuss the significant features of the architectures with respect to the categories described below. Compare the alternatives and discuss relative strengths and weakness of each architecture.

# KWIC Project

The Key Work In Context (KWIC) system takes a set of text lines as input, produces all circular shifts of these lines, and then alphabetizes the results. A KWIC systems is primarily used to create an index that is quickly searchable because every key word can be looked up alphabetically even if it does not appear at the beginning of the original index phrase. This problem was originally introduced by David Parnas in 1972 to emphasize information hiding. The following is an example of the input and output of a KWIC system:

Input: sequence of lines:

*An Introduction to Software Architecture*

*Key Word in Context*

Output: circularly shifted, alphabetized lines (case is ignored)

*An Introduction to Software Architecture*

*Architecture An Introduction to Software*

*Context Key Word in*

*in Context Key Word*

*Introduction to Software Architecture an*

*Key Word in Context*

*Software Architecture an Introduction to*

*to Software Architecture an Introduction*

*Word in Context Key*

# 3 Evaluating the Architectural Styles

You may use the following categories for evaluating the above architectural styles:

* What implications are there if “change in algorithm” was requested?
* What implications are there if “change in data representations” was requested?
* What implications are there if “change in functionality” was requested?
* Performance: which architecture performs better with respect to time? And space?
* Reusability: which architecture allows higher degree of reusable components?

The algorithm for each task or function or procedure need not be very sophisticated. For example, for sorting, a selection sort is good enough. In particular, you want your sorting routine to be only one module. (In other words, a selection sort with local swap is OK.) Use C, C++, or Java.

# 4 Total Points and Due Date

Total Points: 100 (70 for the design and algorithms; 30 for presentation)

Tentative Due Date: November 22, 2021

# 5 Identification of Problem

This project will compare three architectural styles on a Key Word in Context (KWIC) effort. As described in section 2 above, the KWIC system takes text lines of input, processes the line, first by outputting it in its original format, then shifting the last word to the beginning of the text line and outputting the adjusted text line. This process is repeated until all words have been shifted to the beginning position. The entire set of text lines will be alphabetized providing an alphabetized list of all words in the first position of the line, making it easy to identify all words and phrases in the text file.

This project will use the following three architectural styles:

* Main program and subroutines
* Object-oriented style
* Pipe-and-filter.

It will compare these styles across six categories:

* Change in algorithm
* Change in data representations
* Change in functionality
* Performance in time
* Performance in space
* Reusability.

# 6 Make or Break Issues Driving Design Decisions

This is a simple application with no complex data structures or needs. Its make or break points come in its use of network resources, time and space.

**Main program and subroutines** computations are implicitly invoked as the data is altered, based on the subroutines invoked. By separating functionalities into modules(subroutines), the architecture separates concerns into smaller amounts of complexity, thereby managing the complexity more effectively but as the system changes or adds new demands, the functional structure becomes clumsier. Also, due to high coupling between modules, changes are difficult to accommodate as changes in one function can have unknown effects on other functions.

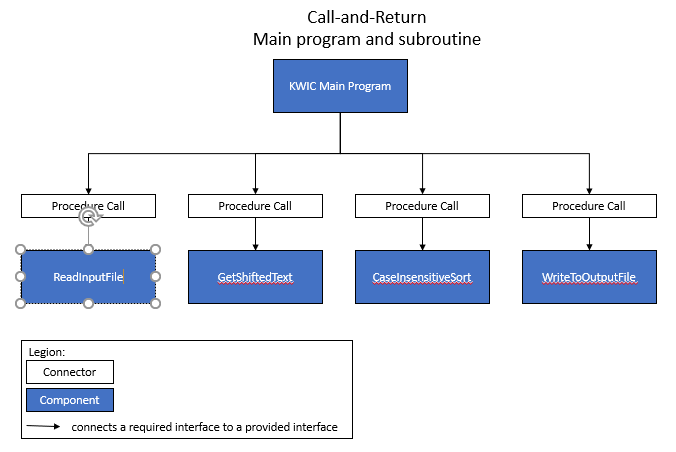
**Object-oriented** style has an interface for each object in the system. It's a design that divides an application's or system's tasks into separate, reusable, and self-contained objects. Object-oriented design is a prominent method that views a software system as a collection of items known as objects. Interaction occurs by either passing parameters or by sending messages. The object uses information hiding to ensure the integrity of its data and functions and only interacts with the interface

**Pipe-and-filter** isolates each process from its co-processes. All filters and processes are independent entities. They do not share state or status with other entities. Output data from one entity is input data to the next entity. The output of each filter must become the input of the next filter. Thus, the data format must line up for each of the adjacent filters.

# 7 Significant Features of the Architectures

## Main program and subroutines

The architectural style of main program and subroutines is a traditional language-influenced style that is the result of using older, traditional programming languages, such as Basic, COBOL, C, C++, Java and Pascal.



**Components** include the main program which controls the four functions: get sentence, shift sentence, sort sentences and display sentences.

**Connectors** exist as procedure calls for each subroutine.

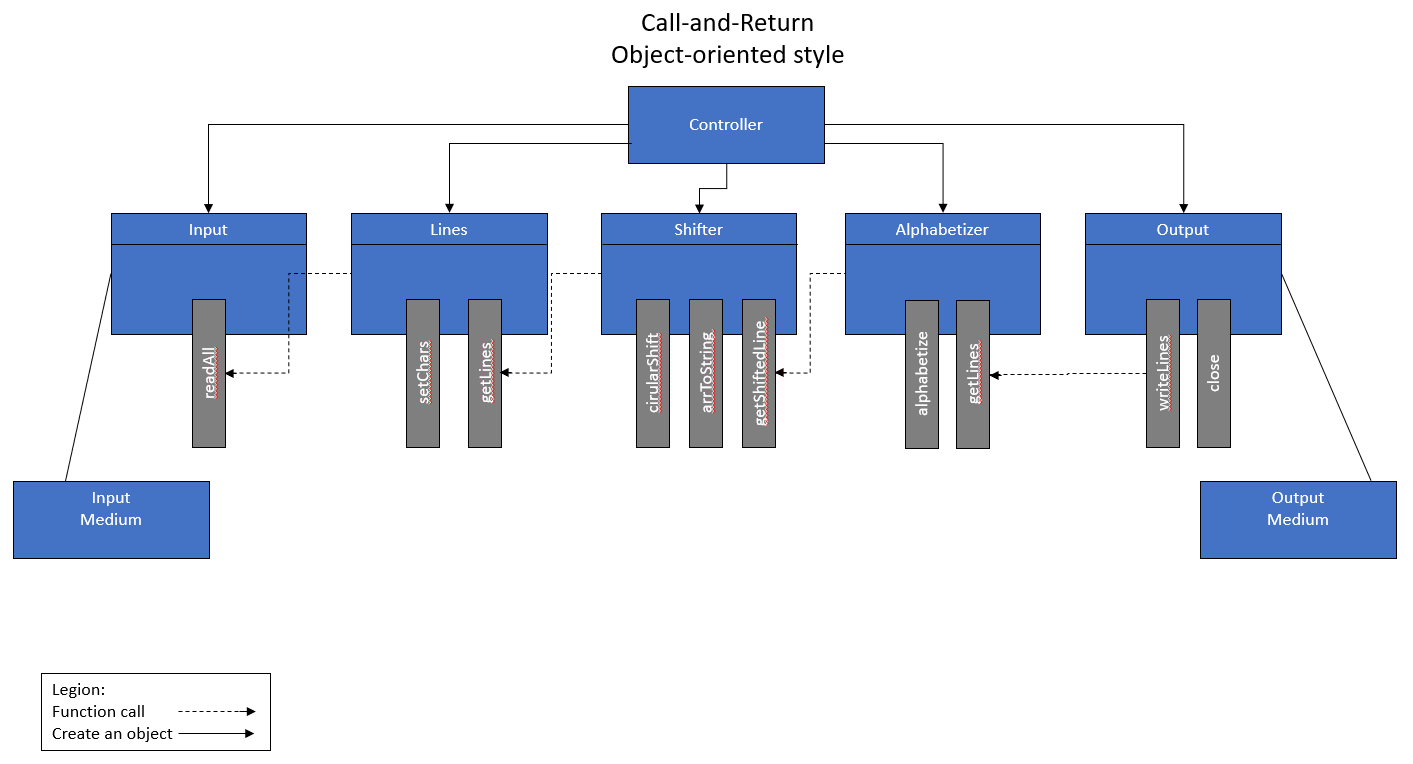
**Data elements** will use shared storage, passing values in and out of the subroutines.

**Topology** is a static hierarchical organization of components, represented in a fully structured directed graph.

**Typical use** is that of small programs for training purposes.

## Object-oriented style

The architectural style of object-oriented is a style in which the state of an object is strongly encapsulated with the functions that interact with the state of the objects. Typical programming languages would include C++ and Java.



**Components** include the all objects of the process.

**Connectors** are the method invocations.

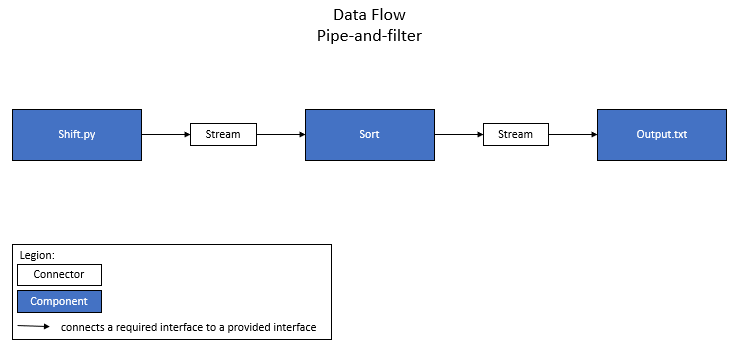
**Data elements** will be provided via parameters into methods.

**Topology** can come in many forms. In this project it will use data sharing.

**Typical use** is when a solution needs a tight correlation between the real-world entities and the coded modules.

## Pipe-and-filter

The architectural style of pipe-and-filter allows separate programs to executed potentially concurrently while data is being passed as a stream from the initial program to next program, both executing at the same time. This style is often used in operating systems and Unix shells. It uses mutually independent filtering processes to incrementally alter incoming data into output. This style is often used to couple two and more filtering process together and can run as a concurrent process.



**Components** use one or more filtering processes to manipulate input streams of data into output data streams.

**Connectors** are explicit processes used to route streams of data.

**Data elements** will be provided via parameters into methods.

**Topology** consists of pipelines.

**Typical use** of the pipe-and-filter style is pervasive in operating systems applications. It is a simple process and frequently used for quick and simple manipulation of data streams.

# 8 Compare Alternatives-Strengths and Weaknesses

## Main program and subroutines

**Strengths:**

* The process is intuitive and easy to understand
* Provides an efficient use of time
* Provides an efficient use of space
* Uses shared data across modules
* Easy to modify.

**Weaknesses:**

* Challenge to scale to large applications
* Not much attention to the data structures
* Lack of information hiding, data changes affect all modules
* Not much reuse
* Tight coupling

## Object-oriented style

**Strengths:**

* Information hiding, data can be changed inside a module without affecting all modules
* Reuse
* Loose coupling

**Weaknesses:**

* To provide access to remote object by distributed systems can require significant middleware
* Inefficient for high-performance modules
* For complex functions, solutions and/or modifications may be challenging
* Somewhat less efficient use of time
* Somewhat less efficient use of space

## Pipe-and-filter

**Strengths:**

* Processing flow is easy to understand
* Processing flow is easy to modify by additional filters
* Reuse

**Weaknesses:**

* When data structures are complex processes maybe inefficient and cumbersome
* Interactive processing is challenging
* Often less efficient use of time
* Often less efficient use of space
* Processing is limited to sequential and batch flow of data

## Conclusions

***Main program and subroutines*** are easy to maintain and efficient with their time and space.

*Change in algorithm:*

If a change in algorithm is proposed, then the architectural style will get impacted as there will be a change in the overall system processing mechanism.

*Change in data representation:*

Change in data representation will also impact overall style because the change in data format may lead to the creation of new subroutines or change in the existing ones.

*Change in functionality:*

If functionality gets changed then components' overall structure will not get affected much. As new functionality can easily replace the existing functionality of the program.

***Object-oriented*** provides an ease of changing data and structures but are not necessarily efficient in use of resources.

*Change in algorithm and data representation:*

When algorithm or data representation design changes are considered, the changes can be accomplished in the individual modules, without affecting other modules. The design uses a high level of cohesion, making algorithm and data changes easier than in other architectures.

Moreover, reuse is supported because modules make fewer assumptions about the others modules with which they interact.

*Change in functionality:*

On the other hand, the solution is not particularly well suited to enhancements. The main problem is that to add new functions to the system, the implementer must either modify the existing modules -- compromising their simplicity and integrity -- or add new modules that lead to performance penalties.

***Pipe-and-filters*** are simple and easy to maintain but are limited when it comes to complex application needs.

*Change in algorithm:* It would be fairly simple and easy to change the algorithm for this pipe-and-filter program. Each component performs one task and connects to the adjoining components regardless of what those components do.

*Change in data representation:* It would be simple to adapt for different data representation. Either the sort function would be changed for different input, or an extra module and filter would be added to the end of the program.

*Change in functionality:* It would be simple to add or change the functionality of the program. Each module in this program performs one task. If that task needs to be changed or a new task added, only that module must be changed or a new module created. The surrounding modules are unaffected by these changes. However, it would be difficult to change the functionality to include user input. This will be discussed more in later parts of this project.

### Performance Comparison:

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance Evaluations** | **Main and subroutines** | **OO** | **Pipe-and-filters** |
| Performance: |  |  |  |
| Size of object code | 595 KB | 473KB | 5KB |
| Execution Time1 | 50.12ms | 157ms | 7.52ms |
| Execution Time 2 | 61.27ms | 234ms | 7.09ms |
| Execution Time 3 | 55.14ms | 160ms | 7.42ms |
| Average Exec Time | 55.51ms | 184ms | 7.34ms |

## Summary

The best architecture for this project should be based on the architectural quality attributes required by the stakeholders. If performance is the top attribute, then pipe-and-filters has provided the best run-times. If modifiability is the top attribute, then OO would be the top choice followed by main-and-subroutine. If reuse is the most important quality, then the choice would be between OO and pipe-and-filters, however OO has the best reuse option. If it is the size of object module, then PF is the smallest. It is seldom that just one quality attribute is used to determine an architecture. All top-quality attributes must be taken into consideration and balanced to provide the stakeholders with the best possible solution. Below is a comparison of quality attributes to be taking into consideration:

