Project Summary

Through this project, we will be evaluating the 4 KWIC Index architectures (Shared Data, Abstract Data Type, Implicit Invocation, and Pipe and Filter) and try to decide which architecture will be the best choice to use for developing a KWIC index generation tool for an online course.

The created KWIC index generation tool will have the following requirements:

- It takes a group of HTML files as its input that represent the online notes from one lecture where the titles of each page in the group will be indexed with couple assumptions:
    - Each lecture will have a maximum of 100 pages.
    - If a page has no title, the tool will run for that specific page and generate a title for it so that it can be used in the outer/general title generation process.
- The tool will be running once a week with a new lecture's worth of HTML pages where it will add the new index entries to the existing index created from previous runs assuming that there is only one lecture occurring per week with the goal of developing this tool to run for more than one lecture per week.
- The output of the indexing tool will be HTML webpage that displays all the index entries from the previous weeks plus the new index entry from the current week in a list form sorted ascendingly by lectures' dates.
- The output HTML webpage will override any existing webpage created from previous runs/generations.
- The indexing tool will initially be executed in a Windows 10 PC platform; however, the future goal is to be able to run this tool in other platforms like Linux and macOS.

Evaluation Criteria

In my opinion, nowadays, most technology tools' infrastructure should support scalability, reuse, and change in all forms like algorithm, functionality, etc. because of how fast technology keeps developing which makes customers frequently ask for new requirements to make use of that technology development. Accordingly, I'm going to evaluate the 4 KWIC Index architectures by considering several factors concerning the initial use of the system but also how an architecture would adapt to future changes.

Firstly, the architecture should allow the change in algorithm and function for cases like:

- Switching between indexing just the page title and indexing all words on the page in case of page title doesn't exist.
- Having to run the tool for only one lecture versus multiple lectures per week

Secondly, the architecture should also be efficient in both performance and time aspects where the process does not take long time to generate the output HTML page and does not consume a lot of hardware resources like CPU, RAM, and data storage inefficiently.

In addition, I believe that the architecture should have a good maintainability quality where the tool can easily be repaired when any part of the process fails due to new changes.

Lastly, I think the architecture should support portability so it would allow the tool to run not only on Windows 10, but also on other platforms like Linux or macOS which will save time and mental overhead for anyone involved in moving new versions of the tool across different environments.

That is, we are going to evaluate the 4 architectures based on the following criteria: change in algorithm and function, performance, maintainability, and portability.

---

4 Architectures Evaluation

Now that we have our set of criteria ready, we are going to discuss the strengths and weaknesses of each architecture with respect to each of the previously mentioned criterion.

- *Shared Data:* this architecture is probably the best architecture in terms of performance because the data is communicated between the system components through shared storage where all computations also share that same storage which represents both time and hardware efficiency. Besides, I believe that having a shared storage and connected components would make portability easier because all you need to do is to focus on handling only the data storage with its components from one platform to another instead of worrying about moving each isolated component. However, this shared storage logic will badly affect the ability to easily make any changes in the algorithm of that shared storage as it will affect all the components using that storage so you will have to update each component to reflect such change. In addition, I believe maintainability will also be an issue with this architecture because if any failure occurred to that shared storage, it would affect all components using it which makes the maintainer applies the fixes to all components instead of just one piece that caused the issue.
- *Abstract Data Type:* this architecture would be a good choice performance-wise if no change in function will be required in the future. For example, if we wanted our tool to handle languages that read/write from right to left like Arabic, we would have to modify existing modules which affects their simplicity, integrity, and performance. However, this architecture works great if we wanted to make algorithm changes like enhancing the circular shift logic, since individual modules are protected/isolated from others. Also, having those modules separated from each other makes their maintainability much easier because a fix or failure will only affect one module without affecting the

others with which they interact. On the downside, the portability might be an issue with this architecture as you will have to handle each individual component separately on each platform instead of just one chunk of components like in Shared Data architecture.

- *Implicit Invocation:* I believe this architecture is the enhanced version of the Shared Data one especially in terms of change of function and algorithm because data is accessed abstractly instead of exposing the storage formats to the computing modules and additional modules can also be attached to the system which adds some sort of insulation to the system components. This insulation and abstraction accordingly improve the maintainability of the system because of the limited access each component has to other components so the maintainer will only have to worry about fixing the faulted component without affecting other components. However, those enhancements would slightly affect the performance because of the more space its invocations would use, which I believe is not a big sacrifice in our case because the frequency our system will run is not that much (i.e. once a week), and the amount of data stored is not that big compared to other bigger systems like government systems. Another sacrifice that we might have to give in this architecture is portability because of how complicated this architecture style is, which makes it slightly harder to move it from one system to another.
- *Pipe and Filter:* I think this architecture is a perfect choice in terms of change of functions and algorithm since the system's filters are isolated and logically independent of each other which accordingly supports maintainability as well. Thanks to that isolation, the maintainer only focuses on maintaining one filter at a time without affecting other filters. However, this architecture is even worse than the Implicit Invocation architecture when it comes to performance space-wise because of the redundant data copy each filter will do to its output ports. In addition, it would be impossible to add interactive design feature to the system to add, update, or delete an index line for example, since the system does not have any persistent shared storage which is a blocker in our case as I believe that our tool should somehow be interactive with the user (e.g. lecturer) in case of new modifications needed to be made to a specific lecture content or title. Lastly, I believe that portability is another drawback as well, because of the isolation that each filter has which makes it harder to build each component by itself with its own data on a different platform instead of just doing it once as a big chunk, in addition to the large space it requires to build on different platforms.

	Shared Data	***Abstract Data Type***	***Implicit Invocation***	***Pipe and Filter***
Change in Algorithm	*No*	*No*	*Yes*	*Yes*
Change in Function	*Yes*	*No*	*Yes*	*Yes (except for the interactive design feature)*
Performance	*Yes*	*Yes (if no future changes required)*	*Yes (just in our case, but not in general)*	*No*
Maintainability	*No*	*Yes*	*Yes*	*Yes*
Portability	*Yes*	*No*	*No*	*No*

Table 1: Comparison of KWIC Architectures

Best Architecture

After comparing all 4 KWIC architectures and looking at Table 1 results, we can say that the Implicit Invocation is our best choice to use in the online course situation for matching almost everything in our criteria except for the portability part. I am also a true believer of Heraclitus of Ephesus's saying "the only constant in life is change" especially when that change is related to technology, so I based a lot of my decision points on the architecture that gives the system the maximum support for change in both algorithm and function aspects which makes Implicit Invocation a winner over other types that does not support both kind of changes like Shared Data, which only supports change in function. In addition, Implicit Invocation's maintainability is also a plus that reduces the mental overhead by just maintaining one isolated piece at a time without worrying about affecting other system pieces. I was also debating between choosing Implicit Invocation and Pipe and Filter, but the fact that Pipe and Filter does not support interactive design and has a bad performance efficiency compared to Implicit Invocation, made the latter a winner.

Conclusion

In conclusion, through this project, we evaluated the 4 KWIC Index architectures and decided that Implicit Invocation architecture is the best choice to use for developing a KWIC index generation tool for an online course, since its properties are the best suited for our project requirements at the meantime. Although Implicit invocation does not have the best performance criterion like Shared Data and Abstract Data Type, I still think its performance drawback is acceptable in our case especially if we put the amount of stored data and system run frequency into consideration, which can also be fixed by installing modern and high

efficiency hardware like fast RAM, CPU, and Solid-State Drives with big storage. Portability is also another drawback of Implicit Invocation, but I personally think that it is a minor criterion if we compare it to the change of algorithm and function criterion especially that the initial requirement is to have the system running only on Windows 10. In the worst-case scenario, we will just have to spend more time on moving the system's individual modules from one platform to another than we would usually spend on other systems that use architectures like Shared Data.

However, our Implicit Invocation choice would change if we were certain that the system will never change in function, then I would choose the Abstract Data Type architecture as I believe it has the best performance out of all 4 architectures and it also supports a good maintainability quality. Also, if the initial requirement was to have the tool running on multiple systems like Linux or macOS beside Windows 10 without having to worry about change in function, I will change my choice to be Shared Data ,unless we can make the Implicit Invocation design more simple that we can move it from one platform to another as easy as Shared Data, by maybe encapsulating the whole system as a one piece that consists of multiple abstract interfaces that we could move as a one chunk of software from one platform to another.
