SE464: Software Architecture and Design
Key Word in Context (KWIC) Case Study

Background:

- First proposed by David Parnas to demonstrate encapsulation as a design principle.
- The state of programming in 1972:
  - Well, it didn't really exist like it does today.
  - Dominant programming paradigm was procedural programming, as opposed to object oriented programming.
    - Flow charts were a major design technique.
  - Simula67 (invented by Dahl and Nygaard) was around, and Smalltalk (invented by Alan Kay) had just been released by Xerox in 1972.
  - The cost of computer time and hardware greatly exceeded that of labour, which is the reverse of today.

The Problem:

Given an ordered set of sentences, print all circular shifts of each, starting with the alphabetically first one. Circularly shifted sentences are sentences where, instead of starting with the first word, we start with an arbitrary word, then continue along the sentence until all the words are printed, wrapping back to the start of the sentence if we reach the end. The alphabetically first one begins with the alphabetically first word.

<u>Example:</u>

"A dark and stormy night"	becomes	"A dark and stormy night" "dark and stormy night A" "and stormy night A dark" "stormy night A dark and" "night A dark and stormy"
"Welcome to Waterloo" "We are Warriors"	becomes	"to Waterloo Welcome" "are Warriors We" "Waterloo Welcome to" "Warriors We are" "Welcome to Waterloo" "We are Warriors"

Practice Question:

What output will be produced for:

"Fluffy baby geese"

or

"Spring is a good term"
"Although few are on campus"

Design Alternatives:

Brainstormed in class...

Idea:

1. Use a circularly linked list of tokens, where each word is a token, and a different list is maintained for each sentence.

- could sort the words then generate the list to get the first word to use in the circular shift
- could generate the list before sorting, giving the list a method for finding the first word to use.

Notes:

Could use the visitor pattern to implement different functionality in the token list.

2. Could use a character array for the sentence (which doesn't require any additional memory, since it is inherently supported) and track the indexes where each word starts.

Would run faster, since performance is currently determined primarily by IO operations.

Used by David Parnas...

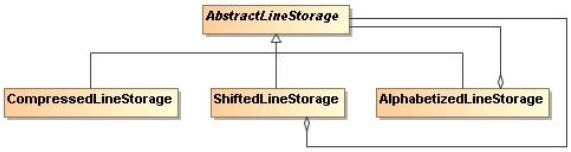
Idea:

1. Use a flow-chart based algorithm.

Notes:

Uses a shared memory store, meaning that little memory. Between the input and shifter modules, the strings can be compressed, further saving memory.

2. Encapsulate the process using an abstract data type.



We used the composite pattern to implement this.

<u>Also compared are:</u>

Idea:	**Notes:**
1. Pipe and filter • similar to the flow chart design in terms of the algorithm, but instead of having a shared memory store, each module reads in a character stream from the last	A uniform interface is used between modules, that is, the character stream. The control flow is distributed, allowing each component to run provided it has collected the necessary data from the last.
2. Implicit Invocation • based off of the Observer / Listener pattern, where accessing memory stores triggers events	The components are similar to the ADT design in that they encapsulate each process, hiding the data representation as well as the process function. However, this leads to loose coupling in the code.

<u>Practice Questions:</u>

What design patterns are used in each of the designs? How does this reflect the use of design patterns as tools for communication in software design? Explain.

What is the Liskov substitution principle? What meaning does it have in object oriented programming? Does the composite pattern reflect this? Explain.

Criteria for Analysis:

- How well does the flow chart based decision compare to the ADT design?
    1. Can the modules be developed independently?
        - **Flow Chart**: Yes and no. The complicated interface makes it more difficult for each module to be developed independently. However, if this interface were well defined from the beginning, then the modules could be independant.
        - **ADT**: Yes and no. Each module extended an abstract base class defining the interface, which means that each module is dependent on it. However, once this class is complete, then each module can be independently implemented.
    2. Can it handle a change in IO format?
        - **Flow Chart**: Requires changing one module, the input module, to handle the new IO format (minimal changes)
        - **ADT**: Requires a new ListStorage designed for the new IO format (minimal changes)
    3. How easy is it to change whether or not the text is stored on RAM or disk?
        - **Flow Chart**: Requires changing each module, since they all need to access the data, and therefore know whether or not it is in RAM or on disk (extensive changes)
        - **ADT**: Requires adding another LineStorage class to handle uncompressing the data for the other modules (minimal changes)
    4. How easy is it to handle compressed string data?
        - **Flow Chart**: Requires changing each module, since the compressed data is shared, and each module requires a way to access (decompress) it (extensive changes)
        - **ADT**: Requires adding another LineStorage class to handle uncompressing the data for the other modules (minimal changes)
    5. How easy is it to change from copying data to just indexing it, and vice versa?
        - **Flow Chart**: Requires changing each module, since data and therefore it's representation is shared (extensive changes)
        - **ADT**: Requires adding another LineStorage class to handle the new data representation (minimal changes)
    6. How easy is it to change how the data is alphabetized?
        - **Flow Chart**: Requires changing one module, the alphabetizer (minimal changes)
        - **ADT**: Requires changing the AlphabetizedLineStorage, or adding another (minimal changes)

- 5 criteria to judge each design on:
    1. How easy is it to change the overall algorithm?
    2. How easy is it to change how the data is represented?
    3. How easy is it to change or expand the program functionality?
    4. How well can it perform?
    5. Can the code be easily re-used?

	Flow chart	ADT	Pipe and Filter	Implicit Invocation
Algorithm	**X**	**X**	√	√
Representation	**X**	√	**X**	**X**
Functionality	√	**X**	√	√
Performance	√	√	**X**	**X**
Re-use	**X**	√	**X**	√

- Pipe and filter designs, because of the copying of data from one filter to another, may experience performance deficits compared to the Flow chart or ADT design, where the data is not copied repeatedly.
    - it makes up for it though by having "plug-in" components, making it super easy to change the algorithm, and to re-use the pieces.

<u>Practice Question:</u>

Pipe and filter designs are generally more maintainable. Why?

Conclusion:

- Different designs offer different tradeoffs that must be considered when determining fitness for future.
    - There really isn't a single "correct" design
    - The context in which a design is made will affect both its fitness of purpose and its fitness for future

- The *Expression Problem* is the fundamental tradeoff between the ability to easily change functionality or data representation in any design.
    - This problem cannot be solved using a particular language or style
    - This tradeoff will determine what future decisions can and cannot be made

<u>Practice Question:</u>

For the designs brainstormed in class, what context were they made in? (i.e. What is the main programming paradigm? What functionality was immediately considered? What future functionality was expected?)

References:

David L. Parnas. "On the criteria to be used in decomposing systems into modules". Communications of the ACM. Volume 15, Issue 12. pp 10535-1058. 1972.

David Garlan and Mary Shaw. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*. 1993.

Richard N. Taylor, Nenad Medvidovi_c, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. 2009.