Architecture Case Study: Key Word in Context (KWIC)

- **Aims:**
  - To demonstrate key features of four architectural styles.
  - To identify relative strengths and weaknesses of these four architectural styles.
- First proposed by **David Parnas** as an example to demonstrate information hiding - key idea behind OO. The problem:

  "The KWIC system 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. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."
- Widely used in Computer Science:
  - Unix man page permutated index
  - Keyword in context indexes for libraries

KWIC Example

Input:

Pattern-Oriented Software Architecture
Software Architecture
Introducing Design Patterns

Output (assuming Pattern-Oriented treated as one word):

Architecture Software
Architecture Pattern-Oriented Software
Design Patterns Introducing
Introducing Design Patterns
Patterns Introducing Design
Pattern-Oriented Software Architecture
Software Architecture
Software Architecture Pattern-Oriented

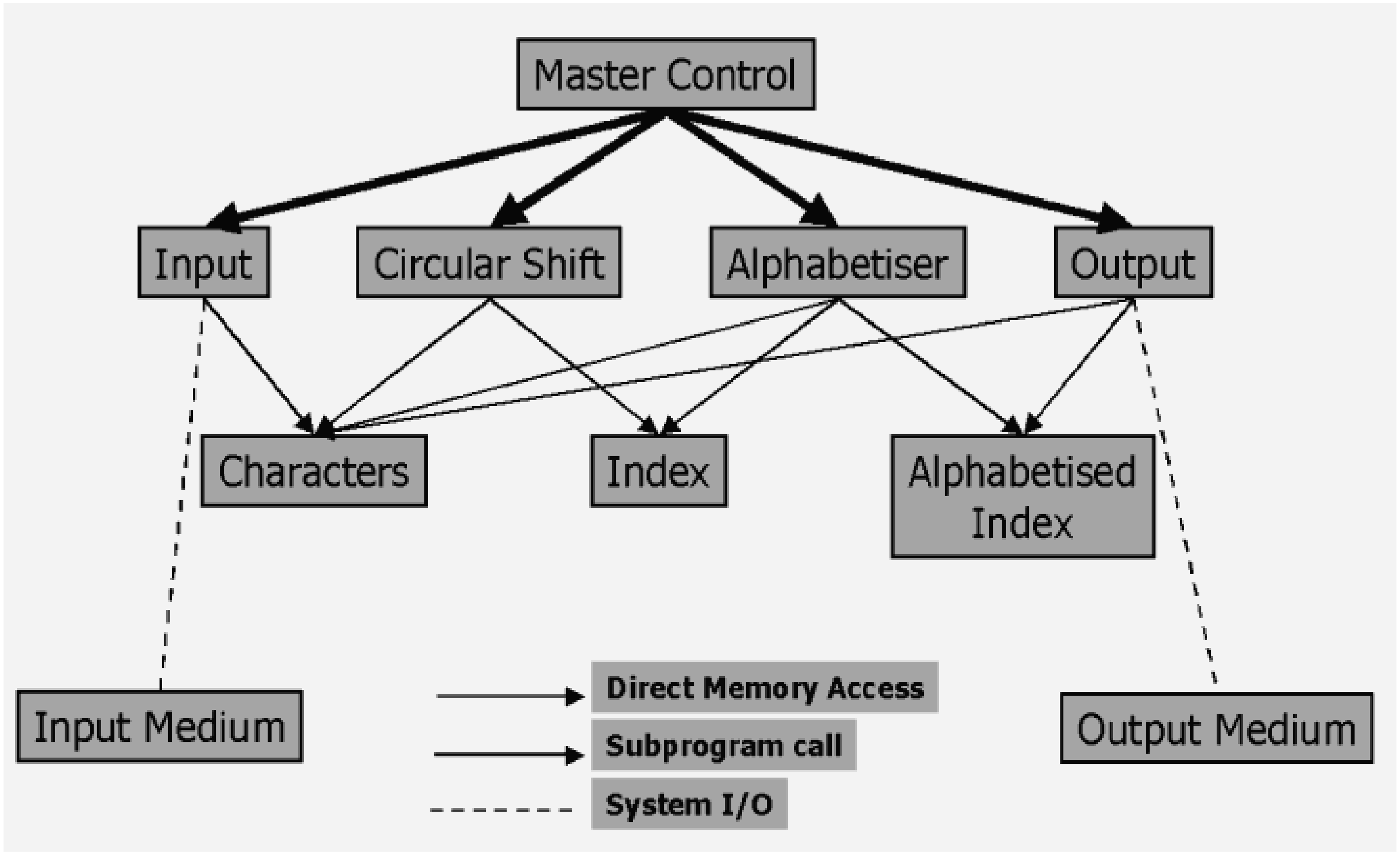
- Can now quickly search for titles that contain phrases such as “Software Architecture” or “Design Pattern” ...

Comparison Criteria

- **Change in overall processing algorithm**: line shifting can be performed as line read in, on all lines after they are read, or on demand when sorting requires a new set of shifted lines.
- **Change in data representation**: circular shifts can be stored explicitly or implicitly as indices into the original lines. Different data structures can be used.
- **Enhancements:** eliminate shifts that start with noise words ("a", "the"), allow deletion of lines, make system interactive
- **Performance**: space and time.
- **Reuse** : to what extent may components be reused?

Four well known, published and implemented solutions based on different architectural styles.

Main Program/Subroutine with Shared Data



Main Program/Subroutine with Shared Data

- Four basic functions: Input, Shift, Alphabetise, Output.
- Main Program controls these **Components** and sequences them in turn.
- Data is **communicated** through shared storage:

  Characters, Index, Alphabetised Index.
- Module **Input** reads the input lines and stores in **Characters** data structure.
- **Circular Shift** directly accesses **Characters** to build **Index** data structure. Each entry in **Index** identifies the address of circular shift in **Characters**.
- **Alphabetiser** directly accesses **Characters** and **Index** to build **Alphabetised Index**. This contains an <u>ordered</u> set of indices into **Characters**.
- **Output** directly accesses **Alphabetised Index** and **Characters** to output the ordered, shifted titles.

Main Program/Subroutine with Shared Data

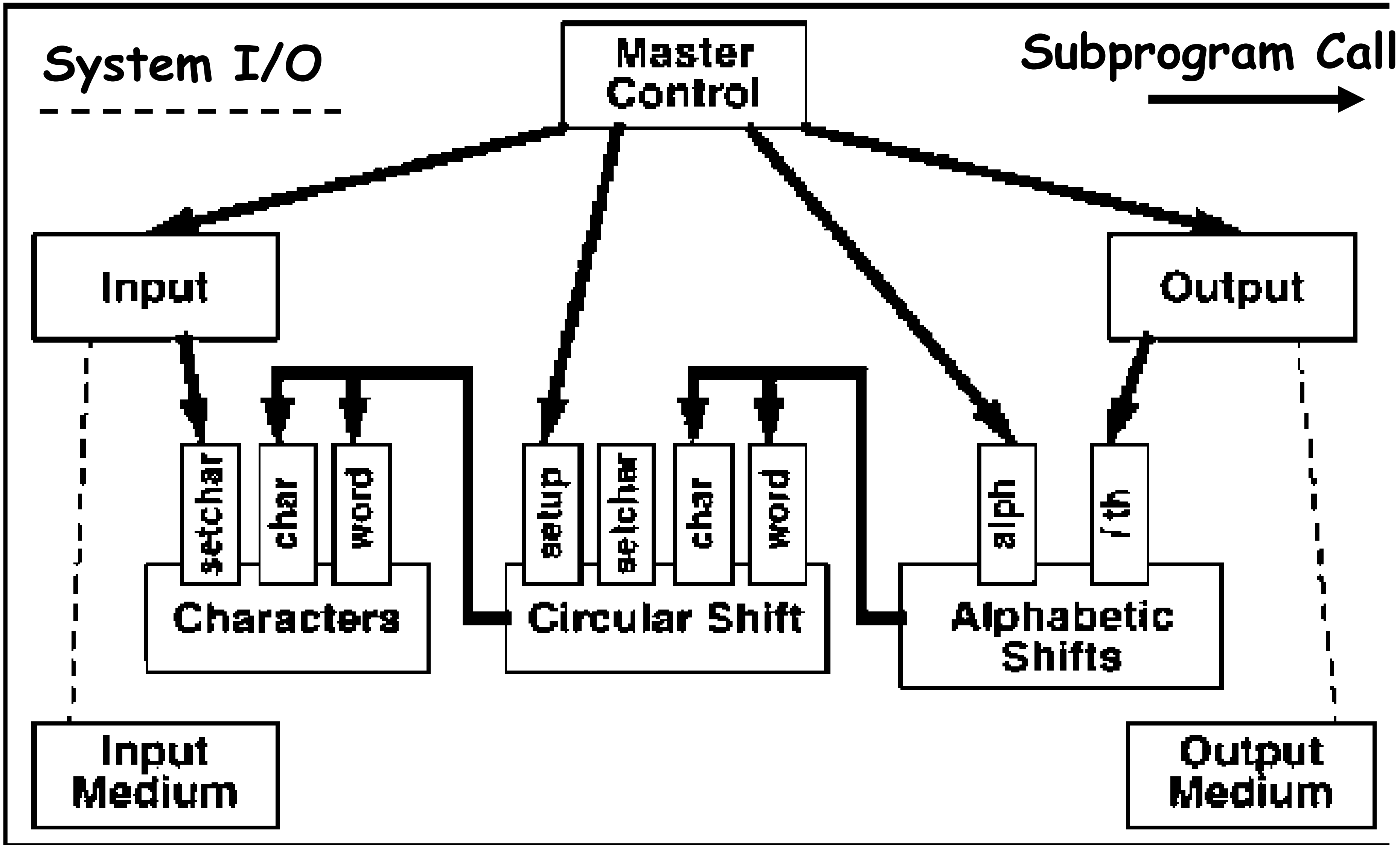
Strengths:

- Efficient use of space (and time) - computations share same data
- Intuitive appeal - natural solution?
- Enhancements based on shared data easily accommodated e.g. removing shifts starting with noise words.

Weaknesses:

- Change of data representation affects all modules - **all modules take advantage of explicit data representation - no information hiding**.
- Not particularly supportive of reuse - explicit references to data structures and other functions. **Tight coupling**.
- Changes to overall processing algorithm – depends on nature of change.

Abstract Data Type (OO)



Abstract Data Type (OO)

- Similar set of modules to Shared Data Architecture.
- Control algorithm is similar.
- **Key Difference:**

  Data **not** directly shared - **data accessed only through module interfaces.**
- Circular Shifts and Alphabetic Shifts typically hide shifted/sorted copies of the original lines (although Parnas' 1972 solution avoided this).

Abstract Data Type (OO)

Strengths:

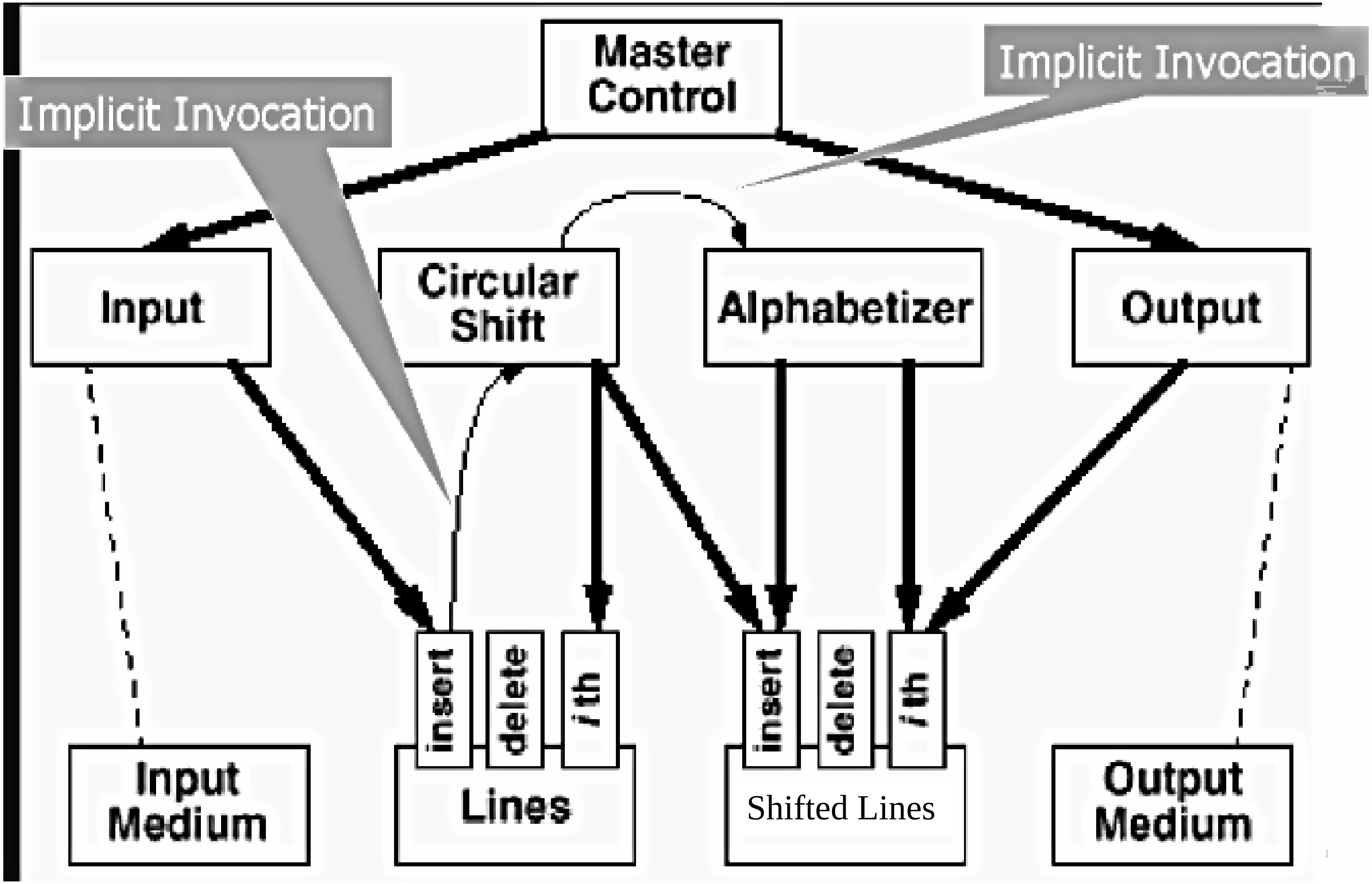
- **Data representations can be changed** inside individual modules without affecting others.
- Reuse better supported because modules make less assumptions about others - looser coupling.

- **Weaknesses:**
  - Research has suggested that this solution is more difficult to enhance with different functionality e.g.

    How would you remove shifts starting with noise words? Feasible,but potentially messy.

    Garlan/Shaw argue that these *functional* modifications can be messy in a data abstraction architecture.
  - Performance? Perhaps more space, and access through interfaces may be slightly slower.

Implicit Invocation



Implicit Invocation

- Implicit invocation - Observer-like notification of change.
- Components similar to Abstract Data Type. Abstract interface hides data representation.

But:

- Components invoked implicitly when data modified - active data model.
- Adding a new line to **Lines** causes an event to be sent to **Circular Shift** module which puts a circular shift of the line in separate, abstract, shared-data store.
- The **Alphabetizer** is triggered by the completion of shifter activities to sort the lines in the **Shifted Lines** buffer.

  Can be based on shared or separate buffers. The sort can be incremental (e.g. by insertion) or on the complete buffer.
- **Alphabetizer** could trigger **Output**.

Implicit Invocation

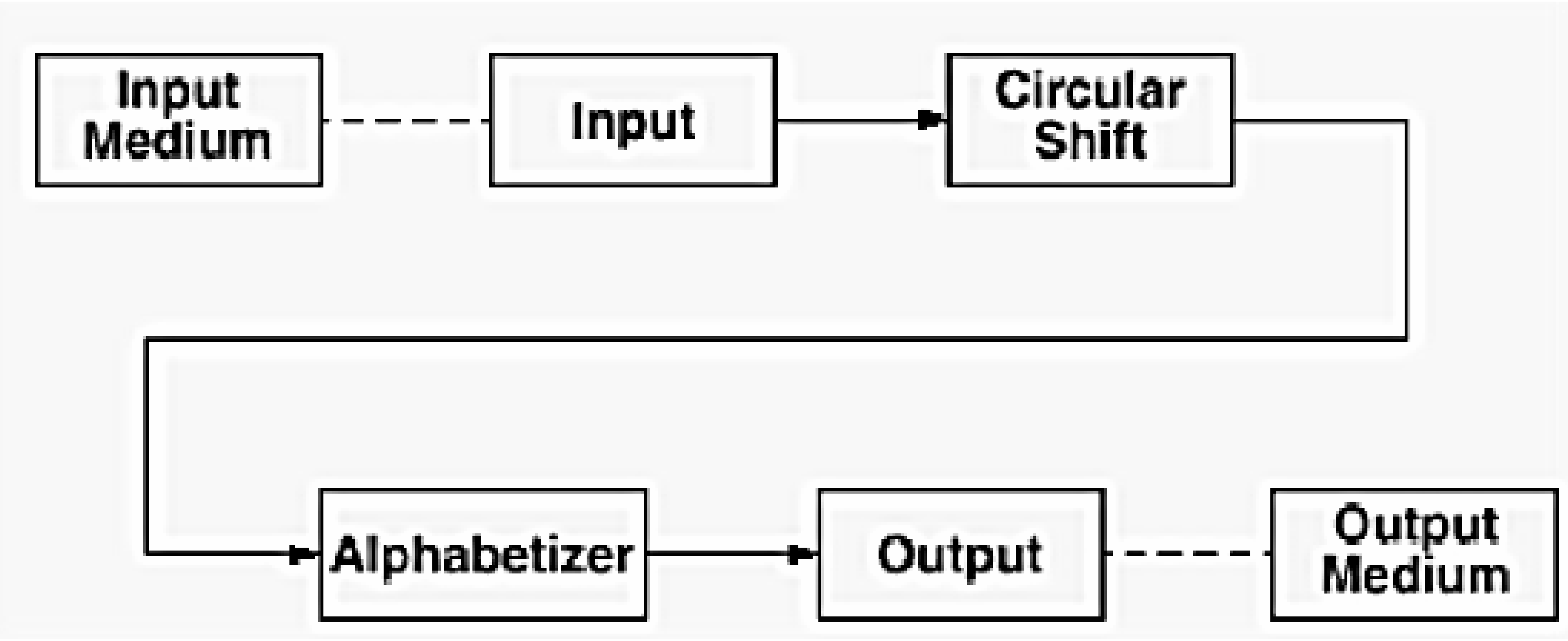
Strengths:

- Can alter overall processing algorithm by registering on different events - triggering after each line entered or when all lines entered.
- May enhance by adding independent functions e.g. Omit that deletes 'noisy' shifts after a Shifted Line insert.
- Data still represented abstractly - supports change in data representation.
- Reusable - implicitly invoked modules only rely on triggering events - loose coupling. Need central management.

Weaknesses:

- **Difficult to control (and understand) processing order. Cycles are a potential problem**
- Data driven nature can mean that more space is required e.g. Line, Shifted Line and Sorted Line buffers.

Pipe and Filter



Pipe and Filter

- 4 filter components: Input, Shift, Alphabetize, Output.
- Each filter (incrementally) process data and passes to next filter.
- Control distributed - each filter can run when it has necessary data.
- Data sharing strictly limited to that transmitted on pipes.

Pipe and Filter

Strengths:

- Intuitive flow of processing.
- Supports enhancements by addition of filters e.g. an omit 'noisy shift' filter, or by modifying independent filters.
- Supports reuse, filters operate in isolation.

Weaknesses:

- Extremely awkward to make interactive e.g. how would you delete user-selected lines (needs persistent data storage).
- Data transmitted in 'lowest common denominator' form.
- Inefficient use of space.
- Overall processing algorithm limited to sequential flow / batch style.

Conclusions

- KWIC Case Study demonstrates generally accepted strengths/weaknesses of Architectural Styles.

  Relative strengths vary from problem to problem and also characteristics: batch or interactive, update or query intensive, ...
- **Main Program** tends to be space efficient and may offer potential to enhance functionality. Generally applicable.
- **Abstract Data Type** (OO) supports change of data representation. Generally applicable.
- **Implicit Invocation** attempts to achieve both data and functional abstraction - loose coupling. **However**, this can be at the cost of a loss of control and understanding.
- **Pipe and Filters** offers simplicity and maintainability but is of limited applicability - 'batch' style.
- **There appear to be errors in the Summary table of Garlan / Shaw - Figure 10 - contradicts what they have written.**

Comparison

Be careful with Garlan and Shaw table in Figure 10 (duplicated in their book and others):

	Shared Data	Abstract Data Type	Implicit Invocation	Pipe and Filter
Change in Algorithm	-	-	+	+ ?
Change in Data Rep.	-	+	- ?	-
Additional Functionality	+	-	+	+
Performance	+	+ ?	-	-
Reuse	-	+	- ?	+