Pipe and Filter Architectural Style

Group Number: 5
Group Members: Fan Zhao 20571694
Yu Gan 20563500
Yuxiao Yu 20594369

1. Have its own vocabulary for its components and connectors? (define)

The Pipe and Filter is an architectural pattern for stream processing. It consists of one or more components called **filters**. These filters will transform or filter data and then pass it on via connectors called **pipes**. These filters, which merely consume and produce data, can be seen as functions like sorting and counting. All of these filters can work at the same time. Also, every pipe connected to a filter has its own role in the function of the filter. When data is sent from the producer (**pump**), it goes through the pipes and filters, and arrives the destination (**sink**). The pump can be a static text file or a keyboard input. The sink can be a file, a database or a computer screen.

2. Impose specific topological constraints? (diagram)

Figure 1 shows a basic structure of Pipe and Filter architecture style. In this example, there are five filters and eight pipes. Each filter will get input from one or more pipes and pass it via pipes. The combination of several filters and pipes can be regarded as a “big” filter.

Figure 2 is an specific example using Pipe and Filter architecture style. This example demonstrates a simple process of making sandwiches. To begin with, the first 4 filters can work simultaneously for preparation. Once they are done, the 5th filter can get the output and combine them together. Next, a following filter will add sauce to it and pass it to customer through a pipe.

In implementation, Pipe and Filter architecture can easily increase in size by adding more pipes and filters to it. However, when data and architecture size become very large, its overall performance may be slow and buffer overflow could

occur.

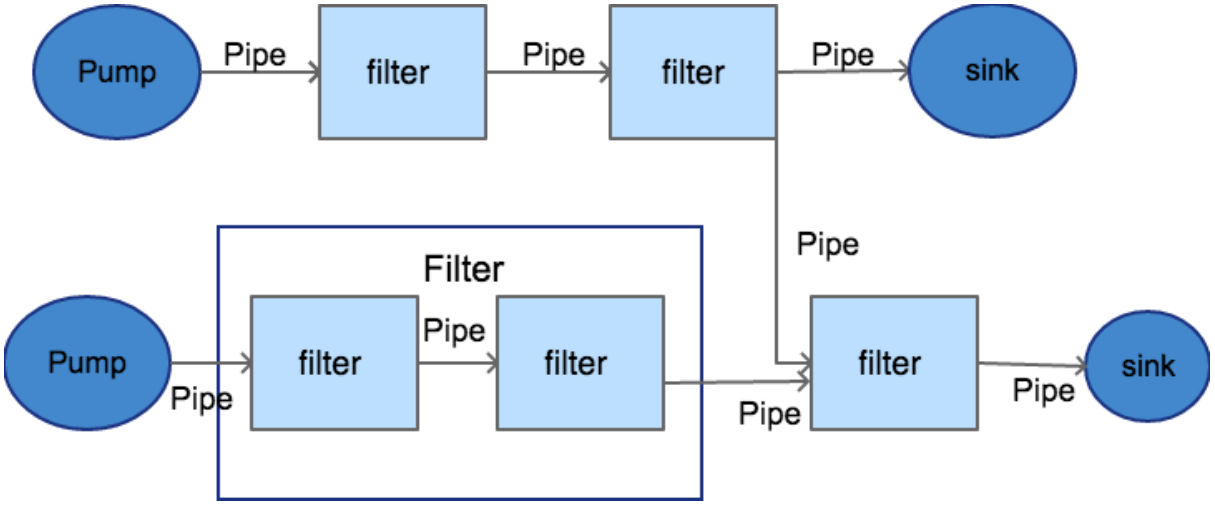


Figure 1

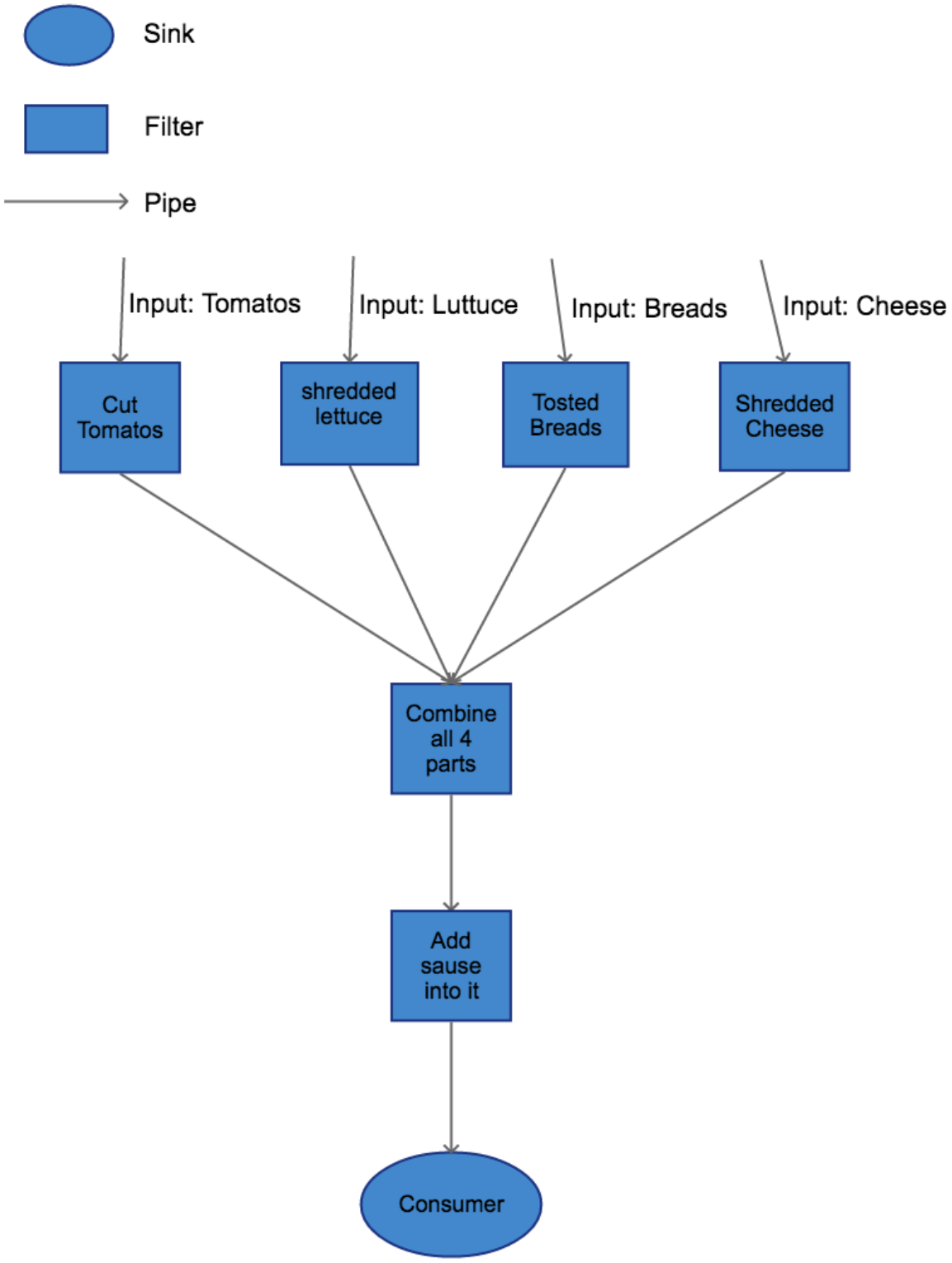

Sink
Filter
Pipe
Input: Tomatos
Input: Luttuce
Input: Breads
Input: Cheese
Cut Tomatos
shredded lettuce
Tosted Breads
Shredded Cheese
Combine all 4 parts
Add sause into it
Consumer

Figure 2

3. Most applicable to specific kinds of problems?

The Pipe and Filter architectural style is suitable for applications that require a defined series of independent computations to be performed on data. Since filters do not share state with other filters, they don't need to know the identity of their upstream or downstream filters. In other words, filters can work independently.

Examples of Pipe and Filter architectural style can be found in Unix Shell Scripts and Compilers. In Unix programs, the output of one program can be linked to the input of another program, i.e. connecting Unix processes via pipes. In compilers. the consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation, which also implements Pipe and Filter architecture style.

4. Engender specific kinds of change resilience?

Systems using Pipe and Filter architectural style can be easily maintained and enhanced with respect to extension of an existing architecture. New filters can be added to existing systems since the filters are separate from each other and connected by explicit connectors. In addition. Therefore, the systems with Pipe and Filter architectural style can engender change resilience.

5. Have any specific negative behaviours?

Since a filter can have arbitrary number of input and output pipes, if some pipe only allow a single data type to pass through, filters need to do parsing internally. This behaviour could slow the filters down. Moreover, if a filter needs to receive data as a whole and then transform it, for example an array, its data buffer could overflow. Also, pipe and filters are similar to black boxes. We don't know how they are implemented. For example, we give an input to a pipe, and the pipe passes this input to the filter. Once the filter is done with the input, it produces an output, and we don't know what the filter has done internally. Last but not least, if an error is passed in from a pipe to a filter, it will also be transmitted to the next filter.

6. Support/inhibit specific NFPs?

Supported NFPs:

- Efficiency: it is possible to have all filters working in parallel
- Scalability: pipes and filters can be added or removed from existing system based on the complexity of the system
- Reusability: pipes and filters can be reused based on the requirement. It is also possible to modify the role of pipes and function of filters to satisfy different design

Inhibited NFPs:

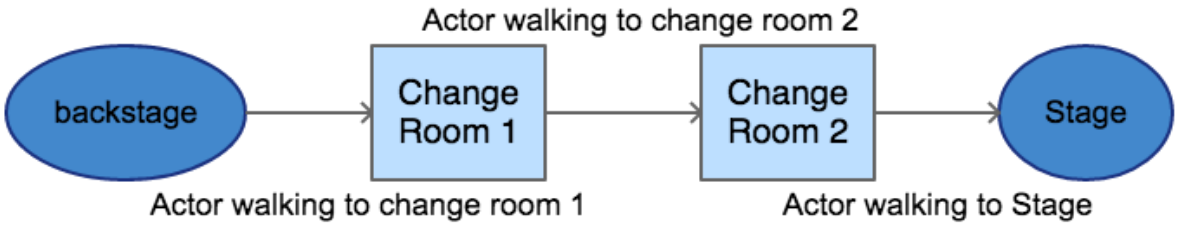
- Dependability (Reliability): reduced reliability due to "weakest link". One failure of one pipeline could result a broken system

- Time performance: 1. when a filter is waiting for the input data, it may deadlock; 2. when data structure is complicated, the processing time of filter can be long

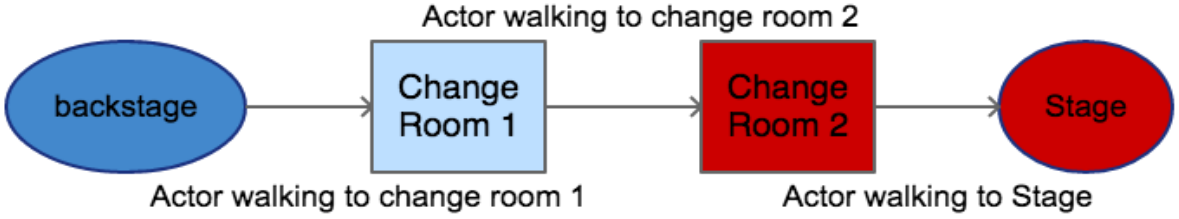
Possible Case Analysis

Change Room 1: Take off the coat and wear the scarf
Change Room 2: Wear a new coat

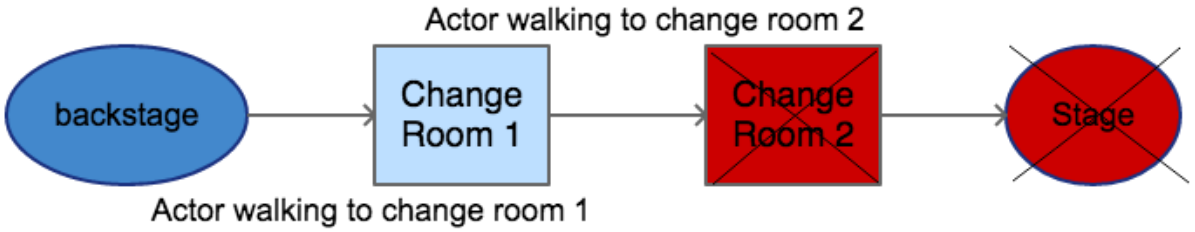
Case 1:
An actor wearing a coat starts from backstage. He goes to Change Room 1, and then takes off the old coat and wear a scarf as required. Then he goes to Change Room 2, wears a new coat. After all these are done, he goes to Stage and start the show. Everything goes smooth in this case.



Case 2:
An actor wearing a coat starts from backstage. He goes to Change Room 1. However, he fails to take off his coat (maybe it's too cold), so he goes to Change Room 2 with his coat. As he is already having a coat, he cannot wear the new coat. Finally, he goes to Stage and starts the show with a bad outlook :(



Case 3:
An actor wearing a coat starts from backstage. He goes to Change Room 1, and then takes off the old coat and wear a scarf as required. Then he goes to Change Room 2 and tries to wear the new coat. However, the new coat is too small for him. He could neither wear a new coat or go to Stage without a coat. The system crashes.


Actor walking to change room 2
backstage
Change Room 1
Change Room 2
Stage
Actor walking to change room 1