Pattern: Pipes and Filters

Consider as in BATCH SEQUENTIAL the case where a complex task can be sub-divided into a number of smaller tasks, which can be defined as a series of independent computations. Additionally the application processes streams of data, i.e. it transforms input data streams into output data streams. This functionality should not be realized by one monolithic component because this component would be overly complex, and it would hinder modifiability and reusability. Furthermore, different clients require different variations of the computations, for instance, the results should be presented in different ways or different kinds of input data should be provided. To reach this goal, it must be possible to flexibly compose individual sub-tasks according to the client's demands.

In a PIPES AND FILTERS architecture a complex task is divided into several sequential subtasks. Each of these sub-tasks is implemented by a separate, independent component, a filter, which handles only this task. Filters have a number of inputs and a number of outputs and they are connected flexibly using pipes but they are never aware of the identity of adjacent filters. Each pipe realizes a stream of data between two components. Each filter consumes and delivers data incrementally, which maximizes the throughput of each individual filter, since filters can potentially work in parallel. Pipes act as data buffers between adjacent filters. The use of PIPES AND FILTERS is advisable when little contextual information needs to be maintained between the filter components and filters retain no state between invocations. PIPES AND FILTERS can be flexibly composed. However, sharing data between these components is expensive or inflexible. There are performance overheads for transferring data in pipes and data transformations, and error handling is rather difficult.

An example of PIPES AND FILTERS is shown in Figure 5. Forks/joins as well as feedback loops are allowed in this pattern, but there is also a variant referred to as a *pipeline*, that forbids both, i.e. has a strict linear topology.

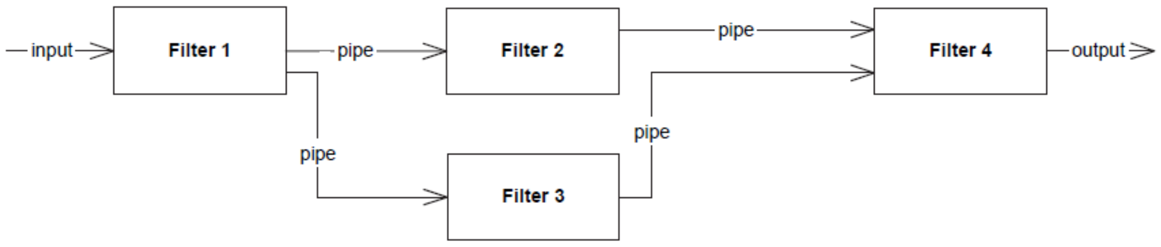

Figure 5: Pipes and filters example

In contrast to BATCH SEQUENTIAL, where there is no explicit abstraction for connectors, the PIPES AND FILTERS pattern considers the pipe connector to be of paramount importance for the transfer of data streams. The key word in PIPES AND FILTERS is flexibility in connecting filters through pipes in order to assemble custom configurations that solve specific problems. Also in PIPES AND FILTERS there is a constant flow of data streams between the filters, while in BATCH SEQUENTIAL, the processing steps are discrete in the sense that each step finishes before the next step may commence.

The pure form of the PIPES AND FILTERS pattern entails that only two adjacent filters can share data through their pipe, but not non-adjacent filters. Therefore pure PIPES AND FILTERS is an alternative to LAYERS and SHARED REPOSITORIES, only if data sharing between nonadjacent processing tasks is not needed. On the other hand, more relaxed forms of the PIPES AND FILTERS pattern can be combined with data-centered architectures like SHARED REPOSITORY, ACTIVE REPOSITORY, or BLACKBOARD to allow for data-sharing between filters. PIPES AND FILTERS can also be used for communication between LAYERS, if data flows through layers are needed.