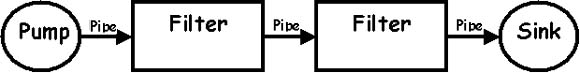
Explain pipes & filters with example

A very simple, yet powerful architecture, that is also very robust. It consists of any number of components (filters) that transform or filter data, before passing it on via connectors (pipes) to other components. The filters are all working *at the same time*. The architecture is often used as a simple sequence, but it may also be used for very complex structures.



The **filter** transforms or *filters* the data it receives via the pipes with which it is connected. A filter can have any number of input pipes and any number of output pipes.

The **pipe** is the connector that passes data from one filter to the next. It is a directional stream of data, that is usually implemented by a data buffer to store all data, until the next filter has time to process it.

The **pump** or producer is the data source. It can be a static text file, or a keyboard input device, continously creating new data.

The **sink** or consumer is the data target. It can be a another file, a database, or a computer screen.

**Examples**

* Unix programs. The output of one program can be linked to the input of another program.
* Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.

**Where does it come from?**

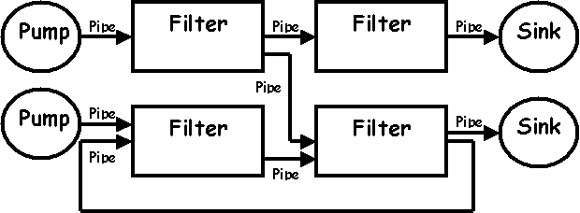
The popularity of the architecture is mainly due to the Unix operating system. It has become popular because Ken Thomson (who created Unix, together with Dennis Ritchie) decided to limit the architecture to a linear pipeline. Using the architecture at all was an idea of Doug McIlroy, their manager at Bell Labs at the time (1972). Both filters (coroutines) and pipes (streams) were not new, but it is not clear to me who designed the architecture of linking the coroutines by streams. As far as I can see, the design was made by Doug McIlroy.

**When should you use it?**

This architecture is great if you have a lot of transformations to perform and you need to be very flexible in using them, yet you want them to be robust.

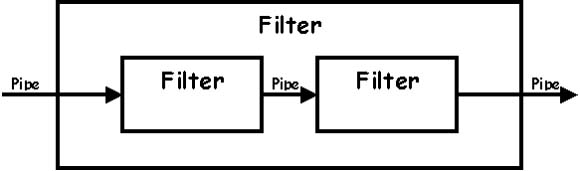
**How does it work?**

The application links together all inputs and outputs of the filters by pipes, then spawns separate threads for each filter to run in.  
Here's an idea of the relationships that can be created between the different filter processes, through pipes.



All filters are processes that run (virtually) at the same time. That means, they can run as different threads, coroutines, or be located on different machines entirely. Every pipe connected to a filter has its own role in the function of the filter. So if you connect a pipe, you also need to specify the role it plays in the filter process. The filters should be made so robust that pipes can be added and removed at runtime. Every time the filter performs a step, it reads from its input pipes, performs its function on this data, and places the result on all output pipes. If there is insufficient data in the input pipes, the filter simply waits.

The architecture also allows for a recursive technique, whereby a filter itself consists of a pipe-filter sequence:



**Problems**

* If a filter needs to wait until it has received all data (e.g. a sort filter), its data buffer may overflow, or it may deadlock.
* If the pipes only allow for a single data type (a character or byte) the filters will need to do some parsing. This complicates things and slows them down. If you create different pipes for different datatypes, you cannot link any pipe to any filter.

**Common implementation techniques**

* Filters are commonly implemented by separate threads. These may be either hardware or software threads/coroutines.