Pipes and Filters Redux

Robert Hanmer
Alcatel-Lucent
2000 Lucent Lane, 6L-521
Naperville, IL 60563

robert.hanmer@alcatel-lucent.com

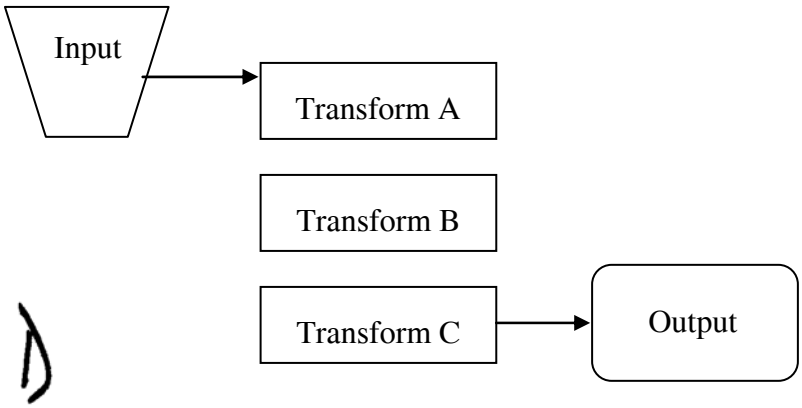
The *Pipes and Filters* pattern was included in Pattern Oriented Software Architecture—A System of Patterns in 1996. It is a well-known pattern that is very, very familiar to anyone that uses or has used the Unix (or Linux) command line. This PLoP submission is a retelling of this classic pattern in a new style. The purpose of this style is to present the information in an easy to use, non-threatening way to new audiences that wouldn't pick up either a 15 year old book or a computer book in general.

Piping Your Data Through Filters

In This Pattern

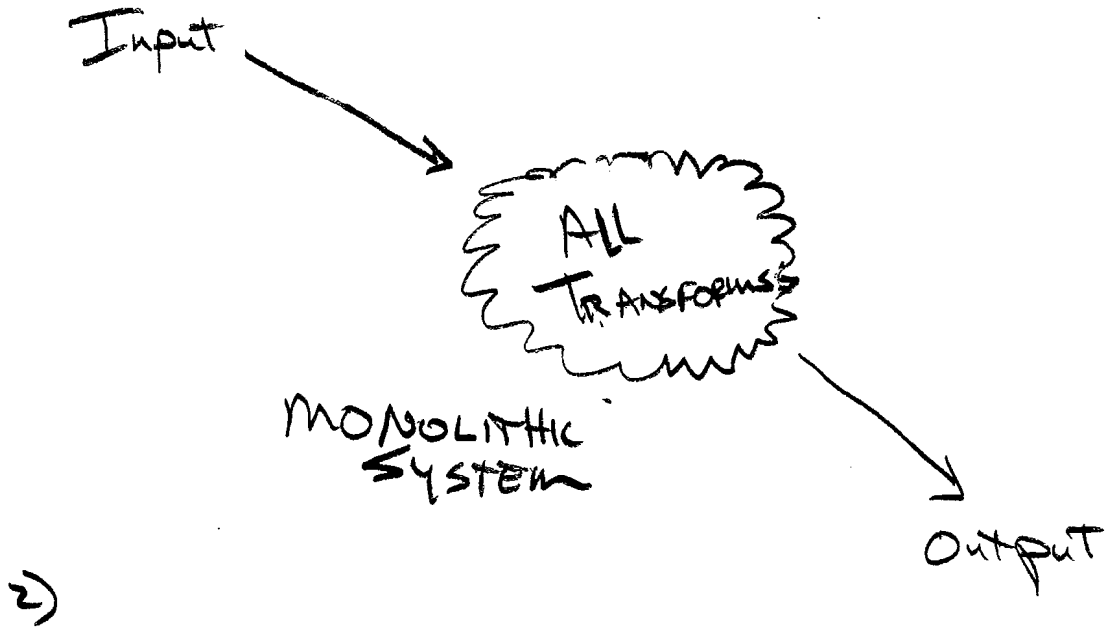
* Streaming your data through a series of filters
* The benefits and liabilities of a Pipes and Filters architecture
* Creating and using Pipes and Filers

Your boss has asked you to lead the development of a new image analysis system. The system will take input from the new image capture system that your company is creating and produce a stream of analyzed data. As you start looking into what's needed you realize the image processing will take a series of stages to transform the data from the raw input into the desired output form as shown in figure 1.



1)

Your first thought is to build one big system to process all of the input. This will be bigger than your group can build with the required schedule so you think that it is an opportunity to get to know those guys in the other group better, maybe also getting more experience managing big projects across the organization.

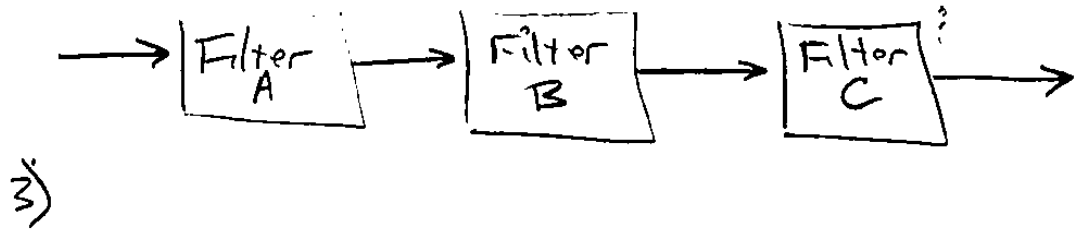


2)

But after thinking about it more and after initial discussions with the other guys you realize that the big solution approach won't work and that you are better off working separately on each of your analysis stages rather than producing a combined, monolithic system.. This comes mostly from the realization that it is easier to build and test small analysis parts rather than big ones.

So instead of one big system that combines all of the analysis steps you think about a set of separate analysis filters that you can combine one after another to perform the analysis. This maps nicely onto the requirements view that shows the different stages of analysis. Each of these analysis steps will be created as a filter on the input stream. It will filter the

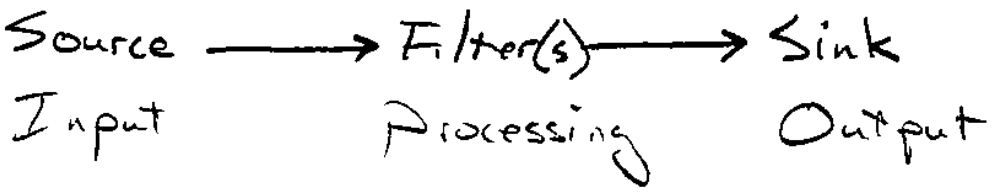
input by performing its required analysis and then produce output that will be used as input to the next stage as shown in figure 3



3)

This separation into filters makes the development easier too. You can focus on filterA and filterC and they other guys can work on filterB which is their expertise. An added benefit is that you don't need to talk to them much because your parts are independent. This can lead to filters that won't work together though, so you realize the need to coordinate with them on the input formats.

Both of your teams know how to design things to clear interfaces, so you decide that's the way to go. You will each work on your filters, A and C for you, and B for the other group and the three parts will be able to communicate via their clear interface specification. Each of your filters can be connected with bits of software piping to create a pipeline for the data. The filters will take in what comes over the input pipe from the input source, filter it and put the output into the output pipe headed towards the sink. This is shown in figure 4.

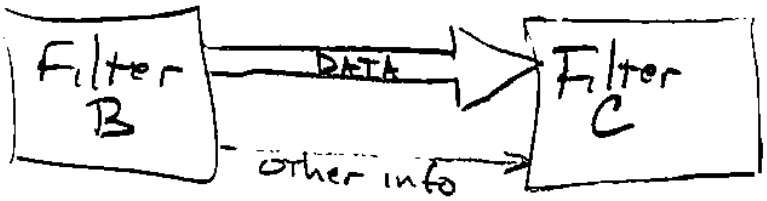


4)

After some work towards the development of your filters has been done your boss points out that a new analysis step is needed and that you need to perform a different filtering than what you planned and everything needs to change. You think about it for a while and then you realize that it simple, you just replace filter C with a new filter and voila, you have the new functionality and it was very easy to change. After getting this done you also see that what you've created is a general collection of parts that can be combined in many, many ways.

As you were building the system you and the other group had to wrestle with several big questions. The first realization was that even though you couldn't build a monolithic solution you saw that the amount of information that you two needed to share was small. Each filter only needs a little bit of information to perform the processing it needed to, and that information is well defined and can be expressed in the interface specification.

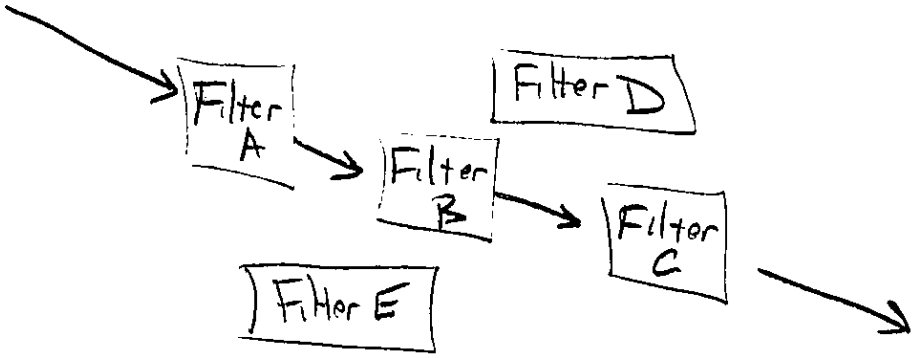
There isn't much need for control information between the filters. The stream of data will contain all of the necessary information.



5)

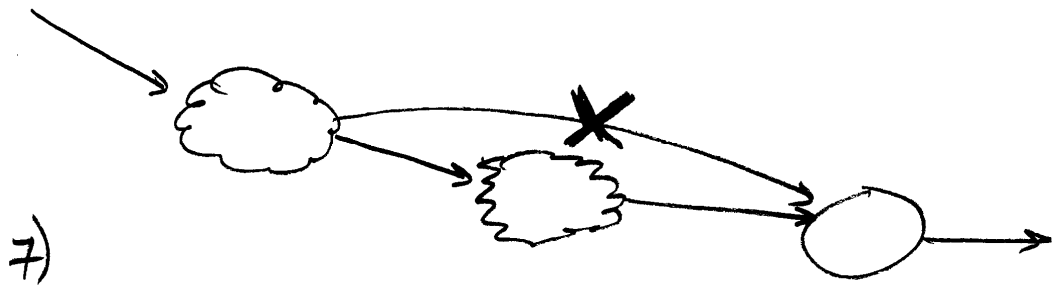
☑ *The input that is needed is clear and well-defined, as is the output produced.*

Another realization later that you had was that you can do different things to the stream of data by combining another, different filter to the sequence. This is what happened when your boss brought the new step. This other filter only needs to know what is intended as its input. It doesn't need to know what happened at previous steps in the chain, as long as it gets the kind of input it expects in the format that it expects. The set of filters that you can build is quite large, as figure 6 shows.



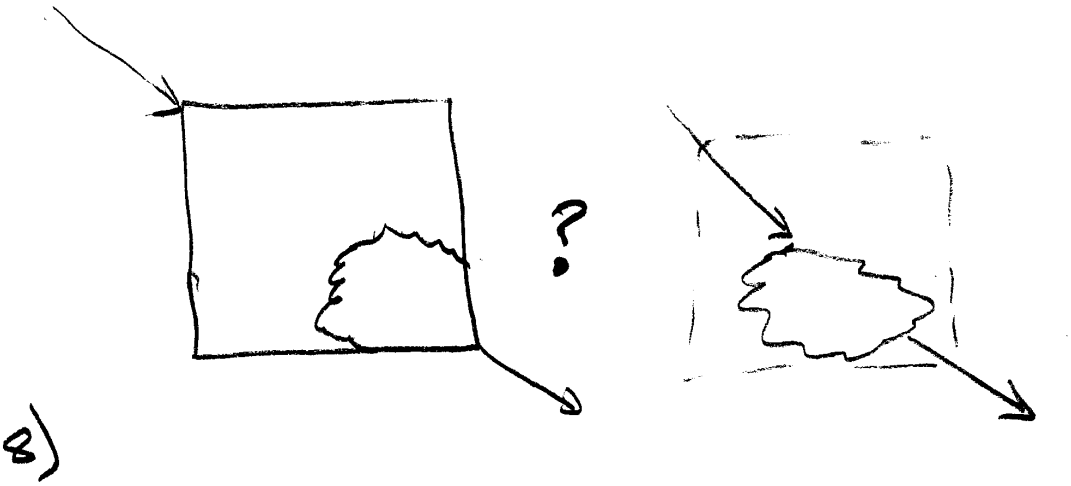
6)

Steps in the processing chain that aren't adjacent do not share information. All information flows directly from one filter to another in the pipeline. No information goes out of band from one filter to another, skipping the intermediate filter elements.



☑ *Information flows only between directly connected pieces of the pipeline.*

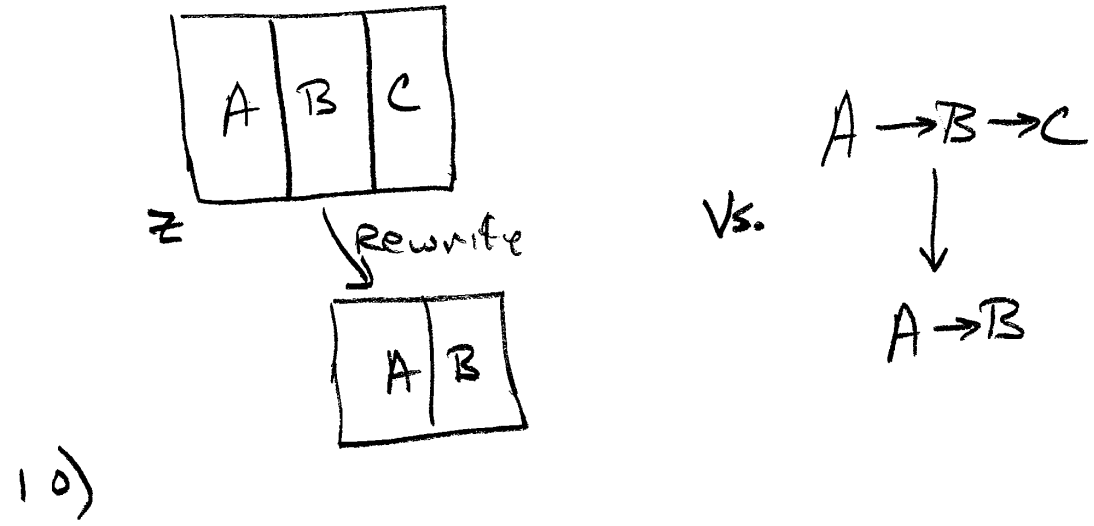
When you were thinking initially about combining the parts of the system into one big monolith you realized that among the other reasons to not combine it was the difficulty that you would have reusing the bits in new contexts. The new requirements that came after development started would have made a large amount of rework.



> ***Sidebar*: Software Tools**
>
> Software tools are small tools that you have in your toolbox that you can pull out whenever you have a new problem that is a repeat of the ones that you've solved before. The filters of our pipeline are perfect examples of tools. The Unix® and Linux® operating systems' shell programs are built on the principle of small commands that you can string together in many, many combinations to do new and interesting things.
>
> > cat <MyFile | grep MyText | wc –l
>
> 9)

When your boss came along and told you that you needed to be able to solve a new and different problem you didn't cringe because you realized that you had the basic building blocks in the filters that you had already built. Take the filters already built, add another of filter or two and you can easily solve a new set of problems. This is one of the benefits of this pipeline of components.

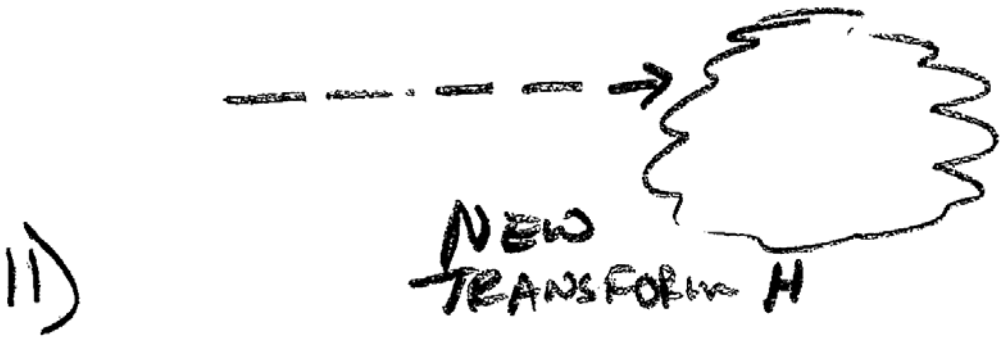
If you had made the bits really big you could still reuse them in different compositions. But if you had built a single system that would do A, B, and C's functionality in one component Z when you only needed two filters, A and B, you'd have to start over and rebuild a system with only the two filters, or build a new system that combined A and B to solve this new problem. If you had A that piped into B that piped into C you can easily just eliminate the C step and have your solution as seen in figure 10..



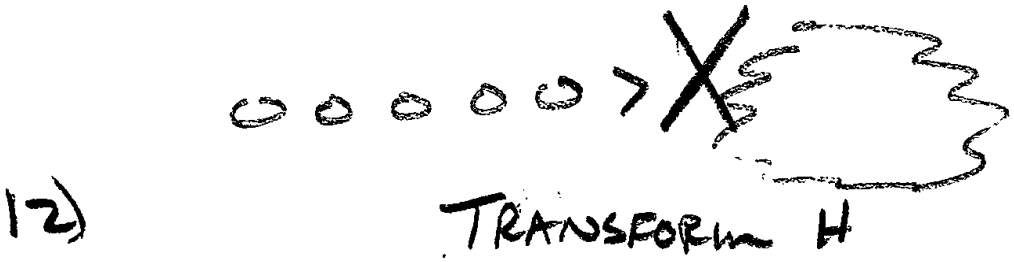
10)

☑ *Small steps are easier to reuse in different contexts than large steps.*

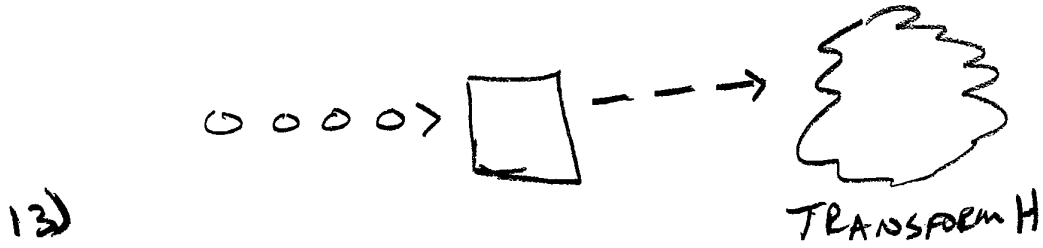
The other fellow's filter was designed to process input of a certain kind. It took input that was formatted in the way of the original specifications.



If the input should ever change, for example if the components were to be used for processing the telemetry from a new image data source, the input format might be a little different.



But with a new, additional filter to the pipeline the input can be translated from the new way into the way that will work with the existing components



The pipeline makes it easy to add new steps to acknowledge that not all data looks alike and that even though you might want to do the same thing, that is run the same analysis, you might not be able unless you allow new building blocks.

☑ *Different data sources exist for your pipeline.*

A related realization is that you might not want the same output all the time. You can add processing steps to the other end of the pipeline to produce different output.

☑ *Present or store results multiple ways.*

During testing your development teams needed to test your part of the system. In order to do this you put sample data in a file that mimicked the output from the data source. That was easy and simple to do. The writers of filter C put their test input in a file to

simplify their testing too. This meant that they didn't have to wait for your filters to generate it, as a result they were doing their development in parallel.

Along the way you gave some thought to efficiency. The way that the two groups were working in parallel showed that exchanging information via files was possible. But you realized the size of the performance penalty that the system would incur if every filter read its input from a file and wrote its output to a file.

About this time your manager comes in and asks if you're done yet. You aren't sure how to answer him. He asks why you don't deliver the filters that read from and write to files. You explain that although you could do this it the system wouldn't meet its performance requirements. Writing a file involves the file system. To build processing steps that talk from one to another is much more efficient. No disk space, no disk caching and/or delays. Building a pipeline to pass data from source to sink through a series of filters is a better solution.

☑ *Explicit storage of intermediate results in wasting of resources.*

Combining some of your previous thoughts you see that you can easily add or remove parts to the pipeline to change its input, change the output, or add and delete elements to produce subtly different results.

☑ *Future enhancements are easy by exchanging processing steps.*

As you're thinking about the problem space you start realize that this solution won't fit in all situations. Systems that are highly interactive (at other than a command line level) or that are event driven aren't good matches because they can't easily be broken down and structured into different pipeline stages. They also tend to proceed in the same way over and over so the capability of recombination and rearrangement of the stages isn't useful or helpful.

Your Pipes and Filters Architecture

Pipes and Filters structure the processing of a stream of data. Each processing step is implemented as a filter with information flowing between the filters through a pipe. Filters can be recombined to provide a family of solutions.

Exploring the impacts

As with any design activity there are consequences to the design choices that you've made. Lets look at the consequences of defining a pipes and filters architecture.

You saw the two-fold flexibility of the system in the example above. The system's behaviour could change by exchanging filters for new and different filters. Rearranging the filters into a different order could also change the system's behaviour.

The filters can be reused in other situations. Since the inputs and outputs are well defined and standard, the filters created for one application can be used for different applications, combined with different other filters.

The use of files as the pipeline is possible, and you saw the use of files in the debugging of the system. You can use or avoid files as desired, using them when they make sense.

By allowing the different filter stages to talk directly together through the defined interface, intermediate files can be avoided. Writing information into a file and then reading it out in another stage of processing is inefficient. It takes up disk storage and time to create the file, write the results, close the file, then the filter that is going to use the information must open the file, read it and then close it.

Filters and pipes help rapidly prototype solutions because of the flexibility previously noted. It's easy to build a prototype by combining existing filters to see if the resulting system can be used.

It is hard to share state information in an environment of pipes and filters. If there is state information that needs to be exchanged between the different filters it must be done out of band – outside the normal flow of data from one filter to another, in synchronization with the flow of data in the pipe. This makes it awkward to use a pipes and filters architecture if extra information about the data being passed is needed. The alternative isn't pretty – designing an interface that combines state information and data into one stream. It needs to be combined on one side and separated at the other side. If global data should be shared then it is likely that a pipes and filter architecture shouldn't be used.

Pipes and filters are conducive to parallel processing. If the filters can take their inputs in small chunks they can be started in parallel and run independently allowing the work to be done in parallel.

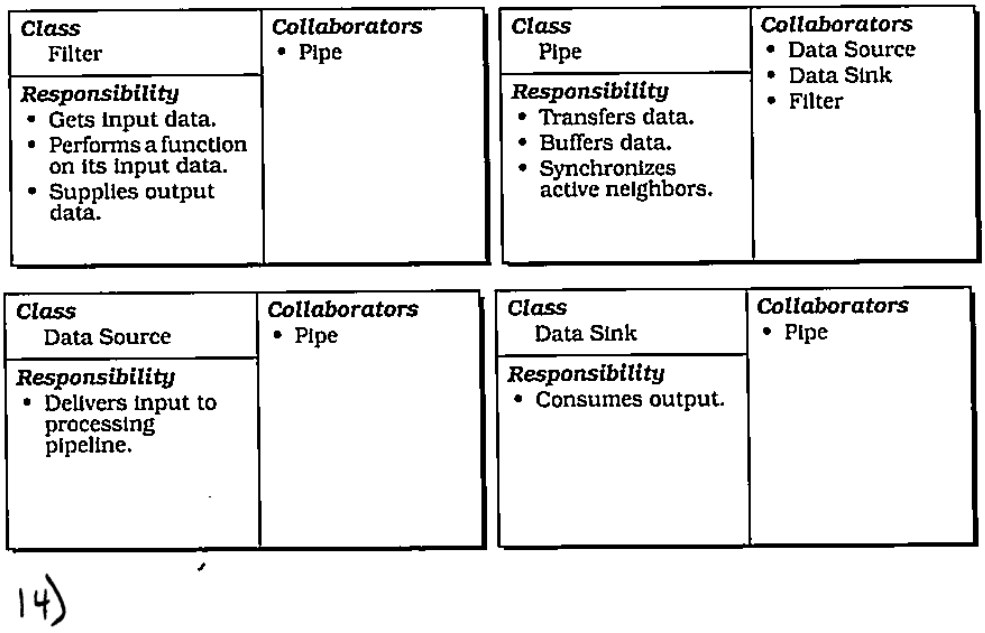
The gains from parallel processing of pipelines can be an illusion though. This is for several reasons such as the cost of transferring the information from one filter to another might be greater than the perceived processing gain. On many systems the actual processing will still require context switching and the management of multiple threads or processes. Another reason is that some filters will need to consume their entire input before doing any processing. This prevents multiple filters from stepping through the data in parallel. And lastly filter synchronization might require resources that will eat into the parallelism gain.

An ideal pipeline passes data to its filters in a form that they can immediately process and where neither the pipeline mechanisms nor the filters need to transform the data which results in overhead. A command line pipeline, like provided in Unix, requires that the information between the stages be translated into ASCII to be managed by the pipeline. Rather than passing only ASCII data around other mechanisms can be used, such as shared memory or even direct calls handing the data off directly.

Error handling is difficult with pipes and filters because the information flow is not conducive to reporting the error information. What actions should the system do when an error is reported by one of the filter stages? If a good strategy for handling errors can't be found for your application then another pattern such as Layers might be more appropriate.

Implementing Pipes and Filters

Four different classes will be present when you implement Pipes and Filters. These are shown in the following figure along with their responsibilities.



14)

There are six steps to implementing a Pipes and Filters architecture. These will be discussed now. The example from the earlier sections will be used to illustrate the steps.

1. Divide task into a sequence of filters which are the Filter classes in figure 14. Look at your problem and identify the different tasks to be performed. They can't have any overlap, and they must provide complete output from one filter that is complete input to another. In other words each filter input is exactly the output of the previous filter in the pipeline. This is the step to think about other combinations or other filters that you might want to work together.

In the example, you've found that there are the A, B and C filter that should be implemented separately.

The overall requirements of the system, will define the nature of the Data Source and Data Sink classes.

2. Define the format for information that the pipe objects of figure 14 will pass between the different filters. You want as uniform a format for this information as possible because it will allow the greatest reuse. Most filters in Unix and Linux pass ASCII text that can be thought of as being organized into lines. This isn't a required format though. Depending on your interconnection method you can use other formats, which you might want to do for efficiency reasons. It's inefficient to convert data back and forth to ASCII for the purposes of the pipe if the same internal representation is going to be used by the different filters. Whatever format is chosen be sure to identify the end of input so the filters know when to stop processing.

For the information to pass between the stages of our pipeline you'll use a binary representation. You've worked with the other group to define the data format during the development of the API for each filter.

3. Decide how to implement each pipe connection. The simplest way to exchange the information along the pipeline is for each filter to "call" the next one, pushing the data for it to process towards it. If you are building upon a Unix or Linux base operating system then you can use the built-in pipeline semantics "|" to build up the pipeline and connect your filters. Another way is to build a framework around your pipeline that will manage the elements. This is the best way if you are building a set of interchangeable filters and efficiency is a primary consideration.

Three variants of Pipes and Filters show up as you are considering the pipeline connection question. They are the Push, Pull and Push/Pull variants as listed in table 1. In the Push variant each stage of the pipeline *pushes* its output to the next filter which waits passively for its input to arrive. In the Pull variant the final recipient *pulls* the data through the pipeline by requesting the information from the previous filters that keep pulling the data from the source. In the Hybrid variant the pushing and pulling goes both ways.

Variant	Best for
Push	Initiated by the Source having some data to be processed. When the amount of data is low enough that the filters can process whatever arrive.
Pull	Initiated by the Sink calling for some data. When the amount of data needs to be managed, so the filter asks for more data when it is ready.
Hybrid	Filter components might have varying needs within the same system. Some might pull data from their source, and others push data out to the next filter. Pipelines that combine some filters that Push and others that Pull data are common.

Table 1 The 3 Variants of Pipes and Filters

Select the variant that best supports the application that you are building.

In the design of your filter you use a Push variant because the Data Source is constantly generating the data that needs to flow through the system. You define an interface between filters, that is between filter and pipe classes that packages the data from the source or previous filters and includes necessary framing and reference information. Data will be transported in a binary representation. Any new filters must accept and produce this same data format.

4. Design and implement the filters. The next step is to design and implement the filters. If the filters are going to be "passive" and their input is delivered to them then you can use functions to implement them. If the variant that you're using is pulling the data along then a procedure can be used. A passive filter element is one that waits for its input to arrive or waits for its output to be asked for. This is different than an active filter that fetches its own input and pushes its output out and to the next stage of the pipeline. To implement an active pipeline filter you can use either threads or processes.

When designing the filters you should think about efficiency. The overall pipeline of information is thought of as a processing whole, but each part might be a separate process. This means that as your data passes through the pipeline context switches will occur. The effort required to copy data between address spaces is another performance impact to consider. Creating small filters will be flexible but will increase the overhead.

As you design your filters think about how you can control and customize them. You might want to reuse them in slightly different ways, so how can you change their behaviour? Unix and Linux filters take command-line arguments. Creating a global environment that contains the control information is another way to handle this. This might mean a procedural or programmatic framework to manage the pipes and filters, invoking them or customizing their actions. You should choose a method that is compatible with the operating system and operating environement that you are using. Since filters are built to perform only one transformation their implementation can be streamlined and efficient.

In our example, at the startup of the image stream processing the filters are started and given pointers to each other so that they know where they will receive data from and where they will provide it to. The source of the data and the sink of data will only be told where to send or receive data from.

5. Design a way to handle errors. Errors from within a pipeline are hard to handle in general. The individual filters will have different error criteria, and different rules for how they should handle bad input or internal errors. Because there's no shared state there is no easy way for one filter stage to report to its adjacent filters that it has an error, it should have a way to tolerate the errors and continue processing the input stream.

When an error occurs what should a filter do? Unix/Linux has "stderr" which is a standard output stream for error information. Your filter can report its error into stderr. Be aware that all the other filters are doing the same thing so the record might get jumbled and hard to follow. As far as what to do with the input with an error, a good approach in a filter environment like this is to absorb the erroneous input – read through to the end of line but don't do any processing on it and then resume with the next line of input. In some circumstances you will want an error in a filter to abort the whole processing, but that case will be rare.

Error handling is hard to design and implement because the filter might have no control over the input which keeps coming.

With the filters you are building, design them so that whey they encounter an error they send errors to stderr and skip forward to the next chunk of input. To help the downstream filters realize that an error occurred, the flag indicating "error" will be injected into the output stream in place of the output that would have appeared for the erroneous input. The filter elements will recognize this flag and ignore the chunk of input.

6. Set up the processing pipeline. Now that the filters are built and the overall mechanism is defined, create a way to invoke your pipeline. This can be as simple as a script to invoke a shell command line, like you'll use for our example pipeline. If the system only handles a single task you can write a program to coordinate the filters and more the data through the pipe.

You can build a framework that allows you to “drag and drop” filter components onto a pipeline to build up your processing pipeline graphically. Using a script in your environment’s scripting language is another way to define the flow of the pipeline.

For the image processing system you choose to build a graphical framework that allows you to specify (drag and drop) filters onto the image stream. This framework creates the necessary references between filter and pipe objects to process its input stream from source to sink.

Acknowledgements

Thank you to the POSA authors: Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal.

Thanks to PLoP 2011 Shepherd Rebecca Wirfs-Brock.