HEWLETT PACKARD

Composite Filter Pattern

Sherif M. Yacoub
Publishing Systems and Solutions Laboratory
HP Laboratories Palo Alto
HPL-2001-131
May 30th , 2001*

E-mail: sherif_yacoub@hp.com

design patterns, 00 design, filters, document understanding

The design of many software systems often involves the manipulation and processing of digital media or digital content. For instance, the ability to deliver e-services through internet-based delivery channels requires that printed material such as books or articles be converted into forms suitable for electronic distribution. This type of processing often includes preprocessing of the digital media, transformation from one format to another, extraction of metadata information, segmentation of content, and in many cases verification and validation of the resulting data. These types of systems among others could be thought of as the integration, composition, and cascade of processing modules or units. We call these processing modules filters. Each filter manipulates input data and delivers output data to other filters after executing a specific data processing function.

The design of such filtering mechanism is often implemented in systems that manipulate large volume of digital media such as images or streams of data. Filtering systems should be designed in a way that enables the integration of different types of filters whether simple, cascade, or composites. A design that is not limited to specific mechanisms for data filtering is itself usable in multiple applications. We call this design the *Composite Filter* pattern, which integrates the design of several well-known design patterns: *Strategy* pattern [Gamma+95], *Filter* pattern [Grand98], and *Composite* pattern [Gamma+95]. Whereas the integration of several patterns does not necessarily result in a new pattern, we find the design developed by integrating these patterns useful in the implementation of filtering subsystems. This integration provides a robust versatile design solution, which is often used in applications that require various types of data filtering. The *Composite Filter* pattern provides the designer with flexible ways for configuring and integrating filters.

* Internal Accession Date Only Approved for External Publication
To be presented at and published in EuroPLoP 2001, 4-8 July 2001, Irsee, Germany
© Copyright Hewlett-Packard Company 2001

Composite Filter Pattern

Sherif M. Yacoub
Hewlett-Packard Labs
1501 Page Mill Rd., MS 1L-15
Palo Alto, CA 94304
sherif_yacoub@hp.com

INTRODUCTION

The design of many software systems often involves the manipulation and processing of digital media or digital content[1]. For instance, the ability to deliver e-services through internet-based delivery channels requires that printed material such as books or articles be converted into forms suitable for electronic distribution. This type of processing often includes preprocessing of the digital media, transformation from one format to another, extraction of metadata information, segmentation of content, and in many cases verification and validation of the resulting data. These types of systems among others could be thought of as the integration, composition, and cascade of processing modules or units. We call these processing modules filters. Each filter manipulates input data and delivers output data to other filters after executing a specific data processing function.

The design of such filtering mechanism is often implemented in systems that manipulate large volume of digital media such as images or streams of data. Filtering systems should be designed in a way that enables the integration of different types of filters whether simple, cascade, or composites. A design that is not limited to specific mechanisms for data filtering is itself usable in multiple applications. We call this design the *Composite Filter* pattern, which integrates the design of several well-known design patterns: *Strategy* pattern [Gamma+95], *Filter* pattern [Grand98], and *Composite* pattern [Gamma+95]. Whereas the integration of several patterns does not necessarily result in a new pattern, we find the design developed by integrating these patterns useful in the implementation of filtering subsystems. This integration provides a robust versatile design solution, which is often used in applications that require various types of data filtering. The *Composite Filter* pattern provides the designer with flexible ways for configuring and integrating filters.

---

[1] We use the terms digital media, digital content, and data simultaneously.

THE COMPOSITE FILTER PATTERN

Context

You are designing a system that processes digital media, streams of data, or digital content.

Problem

A part of the overall structure of a digital media processing system is a filtering subsystem that transforms/manipulates streams of digital media. The functionality of that subsystem is achieved by integrating several processing modules (filters) together. The way you integrate and connect these filters together controls the format of output (processed) digital media. There are several ways of combining these filters. A flexible design structure is required for modeling the complex combination of these filters. The problem is how do you design your filtering subsystem to support the complex hierarchical combination of filters?

Forces

There are several ways that you can use to combine filters. The composition of filters should be robust enough to allow addition of new filters and replacement of existing ones. It should also be flexible enough to support different ways of combining filters together. For example, assume that your system is processing the digital content of a document page. You are using an OCR filter to extract text from that page. The design of the system should enable you to add a new OCR filter and integrate it in the workflow without having to change other client objects in the system. Another example is the integration of a filter in a pipeline of filters, for instance adding an OCR engine in sequence with another filter that checks the format of the document page. The design should also enable you to use complex filters, which are hierarchical wholes of several parts, for example a complex OCR filter that is composed of several OCR engines and an arbitration mechanism to vote among results.

Such a design should allow you to: plug in and take out filters without affecting other parts of the system (simplify maintenance process); select which filters to use dynamically (simplify customization); and configure the workflow of filtering activities. You should be able to easily arrange the integration and cascading of filter with minimal impact on the system design.

To satisfy these system requirements, consider the following solutions and tradeoffs:

- An obvious choice is to design the filtering subsystem by plugging these filters together in a pipeline using the *Pipes and Filters* pattern [Buschmann+96]. *Pipes and Filters* provides a system design as a cascade of processing steps implemented as filters connected by pipes. This design gives support for pipes and filters as first class elements and gives explicit support for

sequential calls of filters *but* it does not support the integration and composition of those filters in hierarchical fashion.

- The *Filter* pattern [Grand98] can be used to implement sequential calls of filters. It gives explicit support for sources and sinks and simplifies the sequential combination of filters. *However*, this design does not support hierarchical combination of filters, filters that are composition of other filters.

Solution

Use a *Composite* pattern [Gamma+95] to design the filtering system and provide explicit support for filter pipelines.

To implement the design of the *Composite Filter,* you will need to design the filtering subsystem such that the filtering mechanism is encapsulated behind an interface and use the design structure of the *Composite* pattern to provide support for complex hierarchical and pipelined filters. The *Composite* pattern provides support for combining filters in a hierarchical fashion. *Composite* will allow you to model complex multilevel tree structure of filters but the explicit representation of pipelines is not visible in its design. Therefore, you will add first-class support for the pipeline roles in the design structure of *Composite*.

Structure

To implement a structure for the *Composite Filter* pattern solution, you will use a combination of several design patterns: the *Strategy* pattern, the *Composite* pattern, the *Filters and Pipes*, and the *Filter* patterns. Each of these patterns provides one particular design aspect; the integration of these design solution provides the design structure of the *Composite Filter* pattern. Lets first discuss how each solves one particular design aspect of the problem and then find out how to combine them into the *Composite Filter* design[2]

The *Strategy* pattern [Gamma+95] provides a unique interface to the filtering subsystem. Using this pattern, the filtering strategy is flexible and hidden from any calls and invocations form any other application component. This design provides encapsulation of a family of filters and making them interchangeable. The *Strategy* pattern provides the solution structure for this problem as illustrated in Figure 1. The `Strategy` class is the interface for all possible filtering strategies. The `Context` class plays the role of the client (the application entity using the filtering subsystem). This pattern by itself does not provide a solution of how the filters are composed, cascaded, nor the order by which the filters are called.

[2] If you are familiar with the design of the *Strategy*, *Composite*, and *Filter* patterns you can jump to Figure 4 directly.

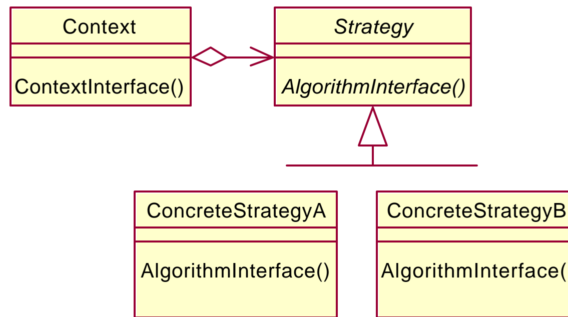

Figure 1 Structure of the Strategy pattern

The *Composite* pattern [Gamam+95] provides a solution structure for filters that could be composed of other simple filters. The design of the *Composite* pattern represents part-whole hierarchies of filters. It also provides a mechanism to attach and detach filters. However, it does not provide control over the order in which various simple filters (components) are called from within a composite filter. The following figure illustrated the solution structure for the *Composite* filter. The `Component` class plays the role of the interface for filters in the composition and declares interfaces for accessing and managing child filters. The `Leaf` class can be used to represent simple filters. The `Composite` class defines behavior for components having children. The `Client` class plays the role of the client (the application entity using the filtering subsystem).

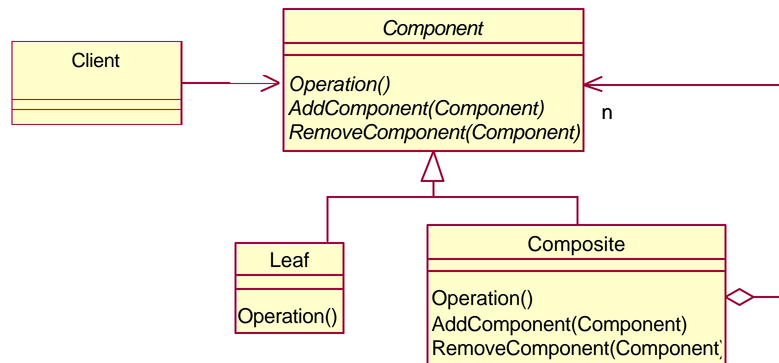

Figure 2 Structure for the Composite pattern

When the order by which the filters process the data is important, you can use the *Filter and Pipe* [Buschmann+96] and the *Filter* [Grand98] patterns. The two patterns have similar designs in which they cascade operation of filters and provide data sources and data sinks. The *Pipes and Filter* pattern is more architectural in the sense that it treats pipes and filters as first-class design constructs. The *Filter* pattern assumes that pipes are established as method calls to the next filters. For simplicity, lets consider the *Filter* pattern only in the following discussion. In these particular filters the order in which the filters are

cascaded define the filtering solution. For instance take the following design structure for the *Filter* pattern (Figure 3). The `AbstractFilter` class is the interface for the *Filter* pattern. The `SimpleFilter` class is the type of filters that does not do cascading i.e. it is the sink of the cascade operation. The `AbstractCascadeFilter` is an interface for a cascading filter. During the creation of a `ConcreteCascadeFilter` the next filter to be called is passed as an argument, hence, the client controls the cascading of the filters according to the order required.

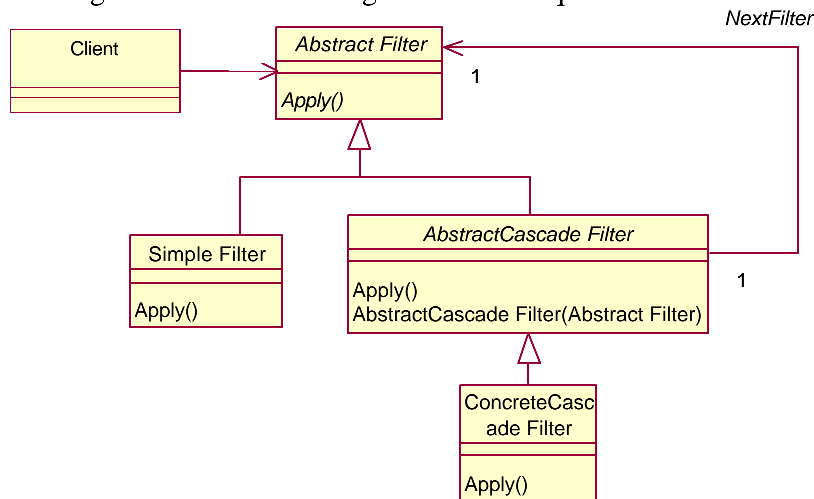

Figure 3 Structure of the Filter pattern

Now, in order to design a filtering subsystem that supports the complex hierarchal and pipelined combination of filters, we will need to combine the above three patterns in one versatile design pattern. The core solution structure is the *Composite* pattern because it provides support for hierarchy of filters. In summary, the *Strategy* design provides the encapsulation aspect, the *Filter* pattern provides a design in which order of filter calls is important, and finally the *Composite* pattern provides a design of filters, which are composition of other filters. The combination of these three patterns together in one design (as illustrated in Figure 4) can be achieved as follows:[3]

- The `Context` class of the *Strategy* pattern, the `Client` class of the *Filter* pattern, and the `Client` class of the *Composite* pattern all represent the same thing; other parts of the system that call the filtering subsystem, lets choose the name `TheClient` as the name of the class representing these classes in the *Composite Filter* pattern as shown in Figure 4.

[3] Legend: A word in italic, for example *Strategy,* is used to represent a pattern name. A word in courier font, for example `Context`, is used to represent the name of pattern class. A word in courier and with round brackets, for example `Apply()`, is used to represent the name of a method.

- The interface for the filtering subsystem can be represented by one class integrating the following three interfaces: Strategy class in the *Strategy* pattern, the Component class in the *Composite* pattern, and the AbstractFilter class in the *Filter* pattern. We call this class AbstractFilter in the *Composite Filter* design structure (Figure 4). This interface contains a method to apply the filter, which corresponds to the AlgorithmInterface() method in the Strategy class of the *Strategy* pattern, the operation() method in the Component class of the *Composite* pattern, and the Apply() method in the AbstracFilter class of the *Filter* pattern. We will use the method named Apply( ) to represent these methods. It also contains the methods to add and remove components defined in the Component class of the *Composite* pattern.
- Simple filters are the ConcreteStrategy classes of the *Strategy* pattern, the Leaf class of the *Composite* pattern, and the SimpleFilter class of the *Filter* pattern. We call this the SimpleFilter in the final design.
- The AbstractCascadeFilter and the concreteCascadeFilter classes are used identically as used in the *Filter* pattern.
- The Composite class is used identically as used in the *Composite* pattern.

As a result of this integration, the design of the *Composite Filter* pattern is illustrated in Figure 4. To use this pattern in your design you can directly use the design in Figure 4, the pervious diagrams are used for illustration and for explaining how the design of the *Composite Filter* pattern evolved from other patterns.

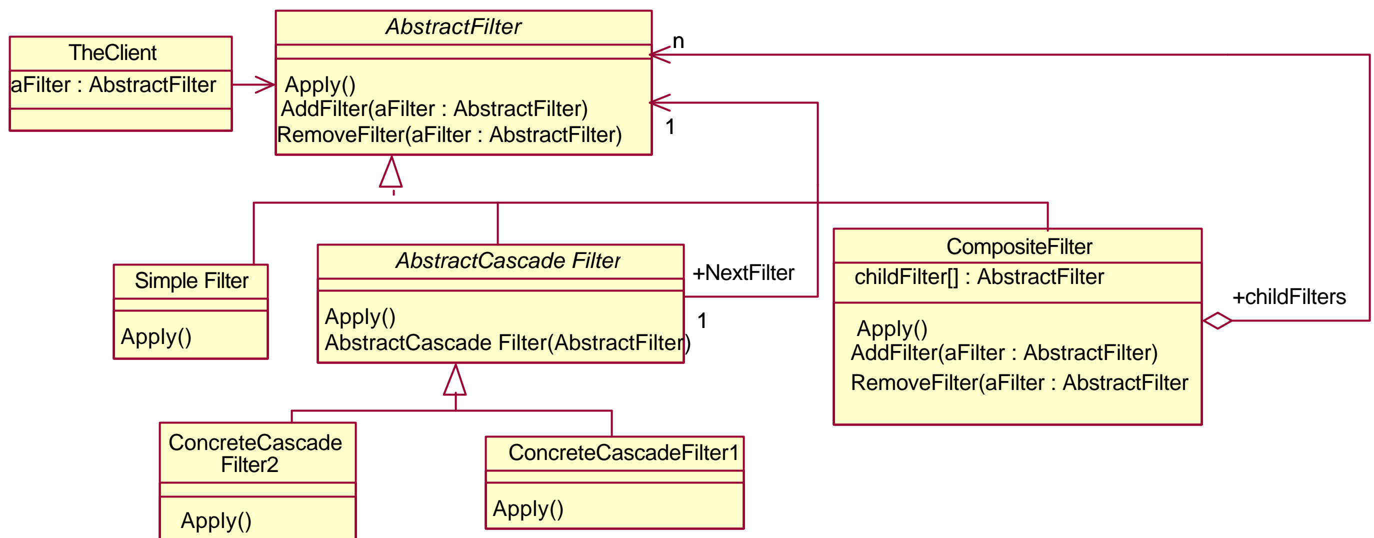

Figure 4 The design of Composite Filter pattern

Solution Behavior

Having the design of the *Composite Filter* pattern in place makes it easy to plug and configure various types of filters. The behavior of the filtering subsystem will depend on the type of filters hooked into the filtering subsystem. The following discussion gives examples of such behaviors.

- Assume you are using a *Composite* filter in you filtering subsystem. The following shows an example of the interaction between the various components of the filter.

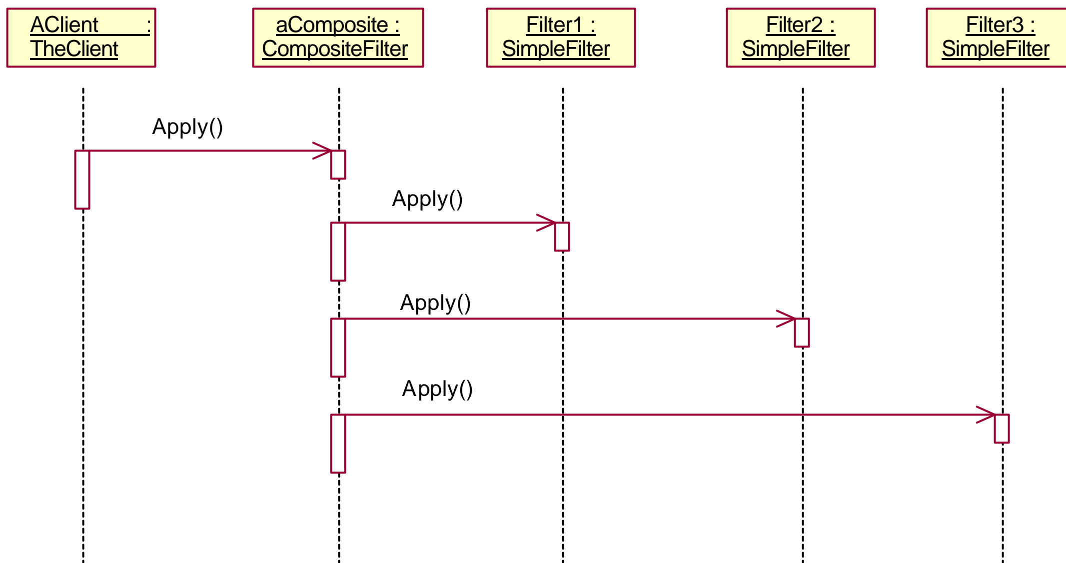

Figure 5 Behavior of the *Composite Filter* pattern consisting of a composite filter

- Assume that you are using a set of cascaded filters. The following shows the behavior of a cascade filer system.

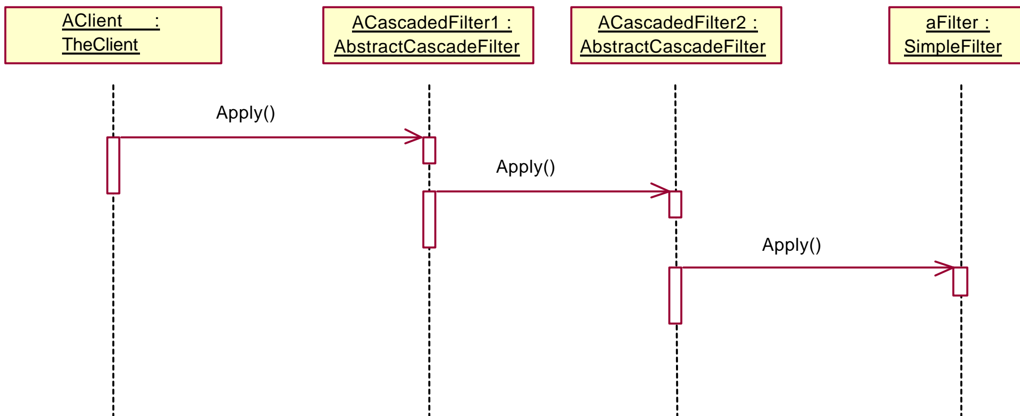

Figure 6 Behavior of a *Composite Filter* pattern consisting of a set cascaded filters

- Assume that the filtering subsystem is a cascade of filters and one of these filters is a composition of simple filters. The following figure illustrated the interaction between various components in such a filter.

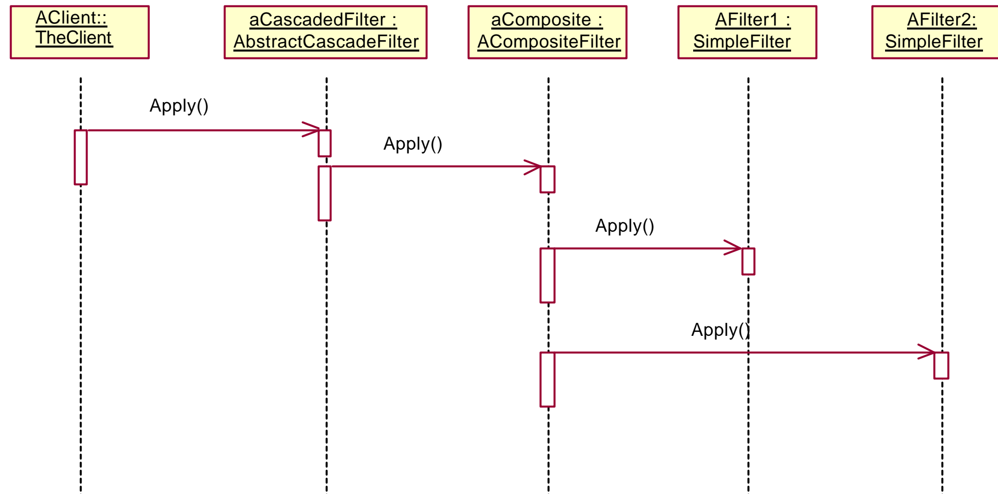

Figure 7 Behavior of a *Composite Filter* consisting of composite and cascade filters.

You can drive as many sequence diagrams as you might want because with the integration of the three patterns *Strategy*, *Composite*, and *Filter,* the design becomes flexible enough to easily accommodate many filter combinations.

If you are implementing filters that handle diversity of data formats then it might be difficult to unify the interface and make filters comply to a common filter interface. One solution that is often used in systems that handle volumes of data is to keep the data stored in a blackboard system (could simply be directory structure) and use pointers to the location of the data as parameters in the filter calls.

Consequences

- The design provides a common interface to the various filtering mechanisms. By virtue of using the *Strategy* pattern as part of the *Composite Filter* design pattern, the filtering subsystem provides a common interface to any filter design or any combination of filters that you will use in your system. This design facilitates the integration of the filtering subsystem in the application design. It also minimizes the impact of choosing a different filtering technique or changing any filter. Such changes have no effect on the rest of the application.
- The design supports various filtering mechanisms. By virtue of using the *Composite* pattern, the design of the *Composite Filter* pattern support types of filters that are composition of simple filters.

By virtue of using the *Filter* pattern, the design of the *Composite Filter* pattern supports types of filters that are sequence of application of simple filters. Hence, the design of the *Composite Filter* pattern can accommodate many filtering mechanisms.

- The design is easily configurable. Choosing between various filtering techniques is easy and has no change-impact on the application since the filtering subsystem provides a common interface. Using dynamic binding, the rest of the application design uses this interface and is not tightly related to the type of filter attached or detached.

Liabilities

By using *Composite Filter* you loose explicit support for pipes in your design. Pipes are data transfer units from one filter to another. *Composite Filter* does not provide support for pipes and assumes that you are able to transfer data between filters as parameters in method invocations, or filters store data on a shared location and pass their references to other filters.

Another possible limitation in the *Composite Filter* is the issue of scale; i.e., using diversity of filter types in your system. All filters have to comply to a common interface. By increasing the number and types of filters used, there is a possibility that you will not be able to force the filters to comply with a unique interface. For instance, some filters may require input of type text while others may require input of type image. Using references to locations where intermediate data is stored can be used as a viable solution.

Known Uses

Filter designs are used in many applications that manipulate digital media, streams of data, and digital content. The design of the *Composite Filter* pattern was abstracted from a prototype for a system that manipulates digital content produced from scanned materials (books and journals). The filters include processing of scanned images, text and image segmentation, transformation from one format to another, quality control, etc. The *Composite Filter* pattern is also useful in the design of compilers in which parsing of text and many other filtering operations are required. The pattern is also useful in image processing applications in which many image-processing algorithms (considered as filters) are applied to raw image data.

Related Patterns

The *Pipes and Filters* architecture pattern [Buschmann+96] provides a structure for systems that process a stream of data in sequential steps. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters in the *Pipes and Filters* pattern allows the designer to build family of related systems. The *Pipes and Filters* pattern has one advantage; it models pipes as first class design elements. However, it does not provide solution for composing filters in

hierarchical fashion. The *Filter* pattern [Grand98] is similar to the *Pipes and Filter* pattern; it has support for sources, sinks, and cascaded filters. However, it does not provide support for pipes as first class design constructs. If you are looking into organizing layers of composite filters, the *Cascade* [Foster+97] pattern can be used. *Cascade* helps making the layers of *Composite* explicit and more visible in the design.

Variations

- You might want to use pipes as first class design constructs in the *Composite Filter* pattern by adding a class type that handles data delivery from on filter to the other. This is useful architecturally if the delivery channel is not as simple as method invocation of the next filter. For instance, you might want to transfer data in a pipe from one location (machine) to another or add error correction control over the transmitted data. However, for cases where pipes are simple method invocation, the use of pipes adds additional classes to the design that will have trivial responsibility of forwarding messages.
- You might want to control the order in which simple filters are called from within a composite filter inside the composite class itself rather than using a separate class for cascading. In this case, you need not use the classes of the *Filter* pattern. The disadvantage of such design is that the essence of cascading is lost and will not be clear at the design level unless you provided it in the documentation.
- You might also consider using variations of the *Composite* pattern itself. For instance if you want the child filters to know about their parent you will need to add a relationship from the `AbstractFilter` class to the `Composite` class as described by Mark [Grand98]

ACKNOWLEDGEMENT

I would like to thank Michael Kircher, who was my EuroPLoP shepherd, for his valuable comments and feedback, which helped me identify the exact forces behind this pattern and sharpen various sections of the patterns including the problem definition and the solution sections.

REFERENCES

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley 1995.

[Buschmann+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture - A Pattern System", Addison-Wesley, 1996.

[Grand98] Mark Grand, “Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML” John Wiley & Sons, 1998.

[Foster+97] Ted Foster and Liping Zhao, “Cascade”, 4th Annual Conference on the Pattern Languages of Programs, PLoP’97.