Rushikesh K. Joshi

Department of Computer Science and Engineering

Indian Institute of Technology

Bombay, India

rkj@cse.iitb.ac.in

Filter Configurations for Transparent Interactions in Distributed Object Systems

In this article, I identify filter configurations based on filter objects for OO distributed systems. The configurations are based on the notion of first class filter objects that transparently intercept messages in a client-server object architecture. Transparency primitives for filtering are described and various filter configurations are illustrated using the transparency primitives. The paper identifies six filter configurations, namely Replacer, Router, Repeater, Value Transformer, Message Transformer, and Logger. Each filter configuration is demonstrated through an example implementation template. I also discuss the consequences of filter configurations for design patterns and evolutionary systems.

Filtering is a very useful abstraction in distributed systems. Filters are increasingly being recognized as important architectural abstractions. An architectural description of filters appears in the work of Shaw and Garlan.[1] Filter implementations have started appearing in commercial systems. For example, CORBA[2] implementations such as Orbix[3] provide specific filtering abilities. The COM specification[4] also defines filtering capabilities for COM. Programming language models for filtering include the Composition Filter model[5] and the Filter Object model.[6] Composition Filters describe filter specifications embedded with object descriptions, whereas, Filter Objects are based on an interclass relationship called filter relationship. An IDL-centric design for distributed filter objects based on this interclass filter relationship can be found in "Filter Objects for Distributed Systems."[7]

Some example applications of filtering systems are security filters, mail filters that filter electronic mails, routers that route requests for load balancing, request loggers that maintain logs of external accesses, and online caches. A requirement of filters in distributed systems is transparency. With transparent filters, clients are not aware of the existence of intermediate filter objects.

In this article, I identify various filter configurations for distributed system structuring. The work is based on the notion of transparent Filter Objects described in "Message Filters for Object-Oriented Systems."[6] Filtering abilities may be viewed as meta-patterns as defined by Pree.[8] Filtering configurations based on the filtering abilities have interesting consequences in the implementation of design patterns. The conventional implementations of design patterns are based on the direct delivery message-passing model. Filter constructs open up a new way of implementing design patterns based on transparent filter objects.

I begin with a brief description of the first class filter object model based on filter relationship between classes. The primitive filtering abstractions in the filter object model consist of a filter relationship between two classes and consequently, a filter relationship between their instances, specifications of filter member functions for their corresponding server member functions and various capabilities of filter objects. These abstractions lead to various filter configurations. Six different filter configurations called Replacer, Router, Repeater, Value Transformer, Message Transformer, and Logger are described in this work. Each description carries an example and an implementation template. A notation is also introduced for representing interaction scenarios in presence of filter objects.

The paper is organized as follows. In the first section, we describe the transparent filter object model. In subsequent sections, the filter configurations are described. Finally, we discuss the consequences of the filter configurations to design pattern implementations and the problem of software evolution.

TRANSPARENT FILTER OBJECTS

Figure 1 differentiates the basic filter object model from the direct delivery model of message passing. In the direct delivery model, a message `Obj.service (x)` originating at ob-

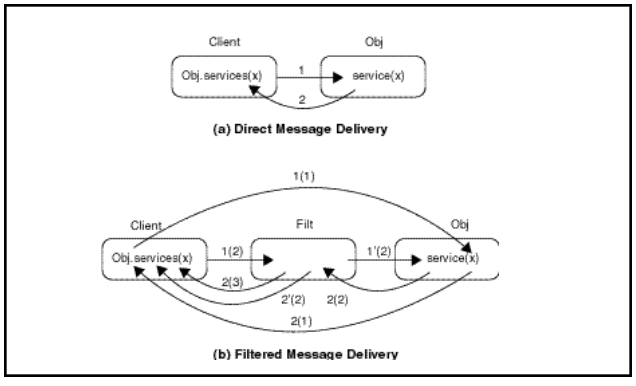

Figure 1. The Filter Object Model.

ject `Client` is directly delivered to object `Obj`. The client's view of message delivery is the same as that of the system's view. In the filtered delivery model, a message is filtered on-the-fly by an intermediate filter object `Filt`. Messages are filtered in upward direction, whereas, return results are filtered in downward direction. The figure shows various possible actions that can be taken by the filter object `Filt`. In this model, the client's view is different from the system's view. The client views the message path as a direct message path to the intended server object, whereas, the system's view of the message path includes an intermediate filter object. The filter object may capture both upward and downward messages leading to a message path 1(2)-1'(2)-2(2)-2'(2). Alternatively, it may bounce a result to the client without passing the message on to the intended server, resulting in a message path 1(2)-2(3). Either or both filtering actions may be deactivated or unspecified. For example, message path 1(1)-2(2)-2'(2) is traversed in absence of an upward filtering action, whereas, message path 1(2)-1'(2)-2(1) indicates an absence of a downward filtering action. Message path 1(1)-2(1) is traversed in presence of a filter object that is not activated in either direction.

Transparent filter objects can be created in an object system through a filter relationship between two classes. For example, at the class level, if class F filters class S (represented as F|S), an instance of class F, say f, can be plugged to s, an instance of class S to filter messages sent to s. The filter object f is a transparent object. The clients invoke methods directly on object s. Filter Object f can intercept on-the-fly a member function invocation on s and can take intermediate actions. The filter class F specifies filter member functions

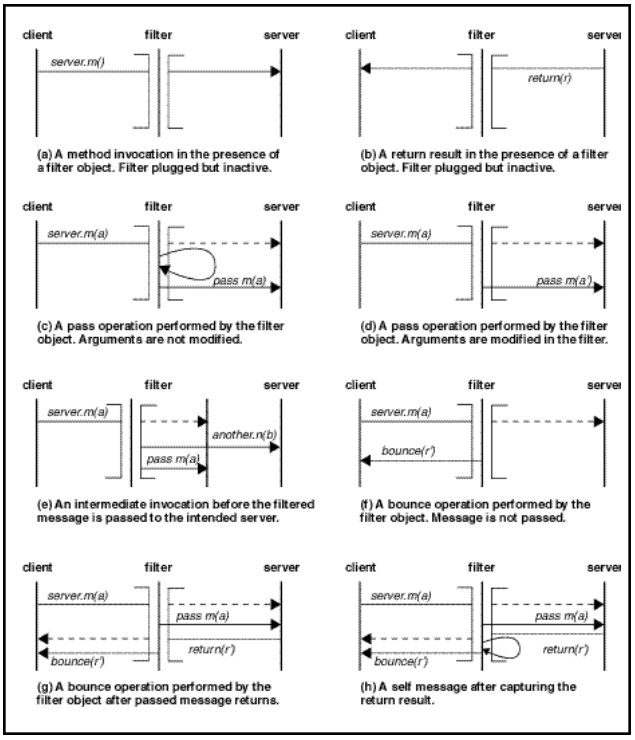

Figure 2. Filtering Scenarios.

that act as filters for their corresponding member functions in the server class S. The filter member functions are specified in a distinct interface called filter interface in the filter class. The filter class can also export conventional interfaces such as public, private, and protected. The filter member functions specified in a filter interface are not accessible as direct invocations by any member function including self-members. Refer to "Message Filters for Object-Oriented Systems,"[6] and "Filter Objects for Distributed Systems"[7] for a detailed discussion on the filter object constructs and their semantics.

Figure 2 shows scenarios elaborating primitive interactions in the presence of a filter object. These interactions form the basis for obtaining various filter configurations that are described in subsequent sections. An extended notation has been developed for drawing interaction diagrams. In Figure 2, the transparency of a filter object is depicted through square braces surrounding the vertical line, representing the filter object.

Scenario (a) shows a method invocation on a server object in presence of a filter object with filter performing no action. Similarly, scenario (b) shows a return result from Server in presence of a filter object, which performs no action. In scenario (c), the filter captures the ongoing message, performs a local computation in the corresponding filter member function, and passes the message on to the destination. The method arguments are unchanged. The portion of the onward message path shown as a dashed segment in this scenario indicates an apparent path, which is not traversed. In the subsequent scenarios, apparent paths (both onward and return paths) that are not traversed have been identified as dashed segments.

In (d), the filter modifies the method arguments from a to a' before passing the method on to the destination. In scenario (e), the filter member function sends an invocation n to object Another before it passes the captured message on to the destination. In scenario (f), the filter member function itself bounces a result to the client without passing the message on to the destination intended by the client. The onward method invocation is thus terminated at the filter object itself. Scenario (g) shows a situation similar to that of (c) except that in a downward direction, an additional filtering action is incorporated. The filter object captures the return result and modifies the return argument from r to r' before it is bounced to the client. Scenario (h) represents a similar situation with a local computation between the result capture and the result bounce actions.

The subsequent sections demonstrate filter configurations based on the properties of filter objects as discussed above. Each configuration is discussed with an example and the filtering abstractions used along with an implementation template. The implementation template has been provided as a pseudo object code that can be translated to a specific programming language extension supporting transparent filter objects. The templates show only the filtering members omitting other implementation details.

REPLACER

A filter member function that works as a `replacer` provides

a replacement code for its corresponding server member function. The filter member in this case must take a bounce action to return a value to the client on behalf of the server. Let `Rpl | S` represent a filter relationship between two classes `Rpl` and `S`, with class `Rpl` acting as a filter class for server class `S`. A replacer upfilter member function for a member function `S::f` may be defined as `UF (S::f) = Rpl::rf = [bounce (self <-- action);]`.

The specification indicates that when a function `f` on an instance of class `S` is invoked, the corresponding upfilter member function `Rpl::rf` transparently filters `S::f`. The code for `Rpl::rf` invokes a method action on self and bounces its return value to the client without passing on the message. Figure 3 provides an interaction diagram for the replacer configuration. The action taken by the replacer implements a bounce action as in scenario (f) and a self-computation preceding a bounce as in scenario (h) in Figure 2.

Action taken on a `cache-hit` situation is an example of the replacer configuration with a cache object modeled as a filter object. The filter object maintains the cached entries as its internal state. Given below is pseudo object code for filter member function `Cache::f` functioning as a replacer in this example.

```
Cache | InfoServer {
filter interface:
       readReplacer()upfilters
       //InfoServer::read(key){
           v = self --> read(key);
           bounce (v);
       }
}
```

A client's view of the message is `InfoServer.read` as shown in the interaction diagram. The dashed segment in this message indicates an apparent path. The filter object explicitly invokes a self message to retrieve the contents of the cached entry. The dotted line representing a return message has originated from within the filter object.

ROUTER

A filter member function that acts as a router redirects requests to other objects. Let `Rtr | S` and `Rtr::f` be a router function for

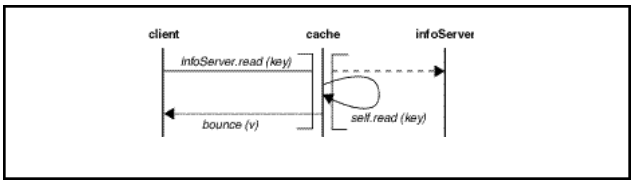

Figure 3. Replacer

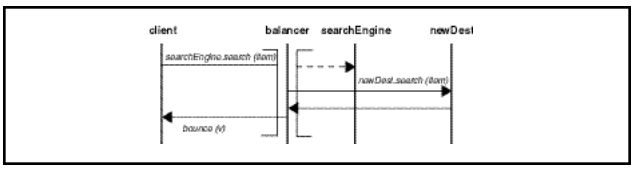

Figure 4. The Router.

member function `S::f`. `Rtr::rf` may be defined as `UF (S::f) = Rtr::rf = [v --> (s2 <-- f); bounce(v);]`. In this case, the router function blocks for a result from server object s_2. However, in a language of implementation that supports non-blocking invocations on objects, it is possible to write non-blocking router functions.

Figure 4 depicts a scenario representing a router configuration. A filter object called `balancer` transparently routes a request `searchEngine.search` to a replicated server object `newDest`. The following pseudo-object-code shows an implementation of the router in a filter class `Balancer`. An invocation of `nextDest()` performs the scheduling. The definition of the member function `nextDest` has been omitted.

```
Balancer | SearchEngine {
filter interface:
       searchRouter () upfilters
       //SearchEngine::search (item) {
             newDest = self --> nextDest();
             v = newDest --> search (item);
             bounce (v);
       }
}
```

REPEATER

A filter object configured in repeater configuration dispatches a member invocation to multiple objects. In this configuration, the filter object maintains a list of subscribed servers to which a message is dispatched. Multicast groups can be created in this fashion. Let a repeater class `Rptr` filter a server class `S`, i.e., `Rptr|S`. In this case, class `Rptr` provides a filter member function `Rptr::rf` such that `Rptr::rf` repeats the invocation of `S::f` on multiple instances of `S` or its equivalent classes. Member function `Rptr::rf` may be defined as `UF (S::f) = Rptr::rf = [s1 <-- f;...sn <-- f; pass;]`.

Let us consider an enrollment scenario in an academic information system. A central enroller object provides an enrollment interface via a member function called `enroll`. Now consider a case where the Civil Engineering department sets up its departmental library and wants all students enrolled as Civil Engineering students to get enrolled in the library. Additionally, there may be another enrollment requirement for students belonging to minor categories.

These new requirements can be handled without modifying the existing code by means of a transparent filter object. The solution is depicted in an interaction diagram in Figure 5. In

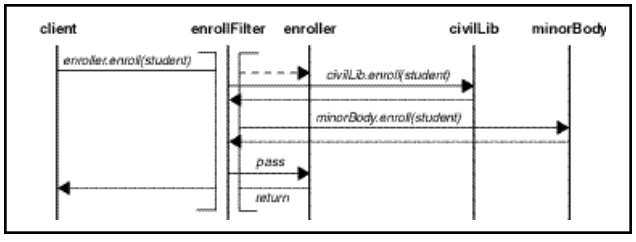

Figure 5. The Repeater.

this figure, the repeater filter object repeats the invocation of method `enroller.enroll` on two other enroll servers. The following pseudo object code describes the corresponding repeater filter method called `multiEnroll()` that dispatches the additional invocations before passing the message on to the intended server.

```
EnrollFilter | Enroller {
filter interface:
     multiEnroll () upfilters
     // Enroller::enroll (student) {
        if (student-->dept()==CIVIL)
        //civilLib --> enroll (student);
        if (student-->status()=MINOR)
        //minorBody --> enroll (student);
        pass;
     }
}
```

In this code, the value returned to the client is the value returned by the intended central enroller. A repeater may alternatively select a return value from various return values received from repeated invocations, including an invocation on the intended server. An example of this variation can be found in the fault-tolerant `n-modular-redundant invocations` in `ShadowObjects`[9] model of replicated services. The `ShadowObjects` model employees a `voting mechanism` to construct a return result from multiple return values received.

VALUE TRANSFORMER

A value transformer transforms the message arguments before the message is delivered to the intended destination. Let `Vtr` be a value transformer filter object for server `S`, i.e., `Vtr | S`. The filter object `Vtr` provides a filter member function `Vtr::vtf` to upfilter the server member function `S::f`. The filter member function can be defined as `Vtr::vtf(v) = [v' --> (self <-- transform(v)); pass (v');]`. The `Vtr::transform` member function is an implementation member defined in `Vtr` to transform the message value. The filter passes the transformed value on to the intended destination. This scenario is similar to scenario presented in Figure 2(d).

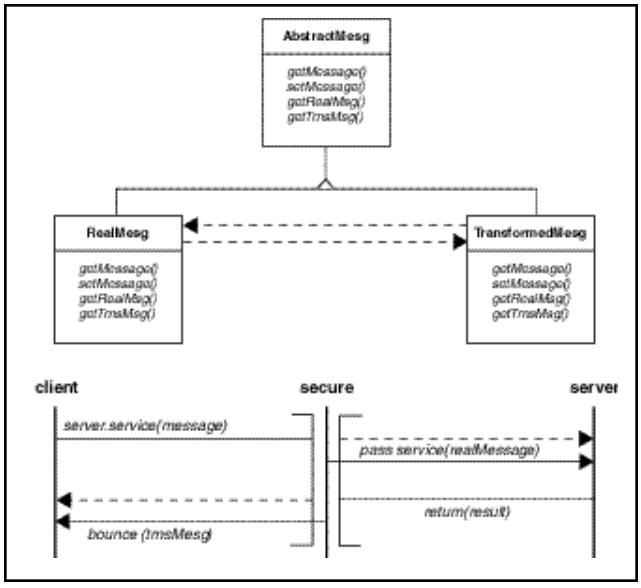

Figure 6. Value Transformer.

Filters that work as value transformers are useful in carrying out on the fly decryption and encryption of message contents. Figure 6 depicts an example of a value transformer configuration. A message content is a polymorphic type as shown in the figure. The filter object that decrypts the incoming message builds an instance of `RealMessage` from the received instance of `TransformedMessage`. Similarly, a return value may be intercepted by the filter object for encryption before bouncing it to the client.

The following code shows a value transformer filter that works as a Decryptor for upward requests and as an Encryptor for downward results. As shown in the figure, the Decryptor function obtains the real message from a decrypted message with a key and similarly obtains an encrypted message from a real message.

```
Secure | Server {
filter interface:
       decryptor () upfilters server::
       //service (message) {
            realMesg = message >getRealMesg(key1);
            pass (realMesg);
      }
       encryptor () downfilters server::
       //service (result) {
            transMesg = result ->getTrnsMesg(key2);
            bounce (transMesg);
      }
}
```

MESSAGE TRANSFORMER

A message transformer filter transforms the type of the message. The new message is either passed on to the intended destination or sent to a new destination. This configuration can be used as a transparent compatibility adapter. For example, older clients that use an older server interface may be diverted to a new interface on a newer server through a transparent message transformer provided that the old messages can be mapped to the new interface. The difference between the `Router` and the `Message Transformer` is that the former routes a message to an equivalent server, whereas, the latter changes the message and diverts it to a possibly different server.

Figure 7 depicts a scenario based on the message transformer configuration. A message transformer filter may be defined as

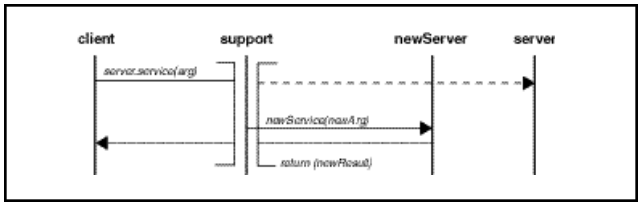

Figure 7. Message Transformer.

`UF (S::f) = Mtr::mtf = [ bounce (s2 <-- g () );]`. In this case, the transformer filter object bounces the value returned by the member function `g()`. The following code shows a message transformer that filters invocations to an old service and invokes a compatible member function on an upgraded server.

```
Support | Server {
filter interface:
      mTransform () upfilters Server::service(arg){
          newArg = self --> newArgument (arg);
          bounce (newServer -->newService(newArg));
      }
}
```

REQUEST LOGGER

Figure 8 shows a filter object that works as a logger. The request logger filter configuration is used for transparent request tracking. The logger sends a `logging message` to a log object. Loggers are common in web-based servers. Let a logger filter class `LF filter` a server class `S`, i.e., `LF | S`. [AQ: Could you please rewrite this previous sentence? It is a little unclear.] Class `LF` provides an upfilter member function `LF::lf` that can be defined as `UF (S::f) = LF::lf`, that invokes a `logging message` on a logger object as in `LF::lf = [logger <-- log(); pass;]`.

A representative code for the logger configuration is shown below. It is assumed that the programming environment supports implicit client identification. The logger code retrieves this identification by means of a helper member function `makeContextDetails`.

```
Logger | Server {
filter interface:
       logRequestX () upfilters server::
       // requestX (message) {
            details = makeContextDetails ();
            log --> record (details);
            pass;
       }
}
```

The logger filter configuration provides an interesting implementation for the Decorator design pattern discussed in *Design Patterns.*[10] This implementation is provided in a "Filter Object-Based Decorator."

A Filter Object-Based Decorator

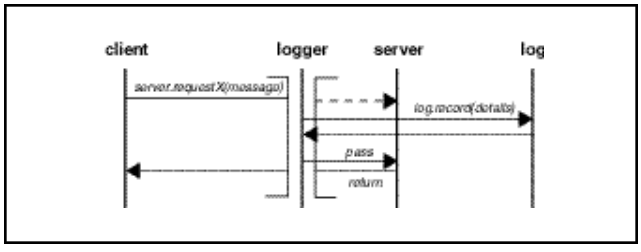

Figure 8. Request Logger.

A filter object-based implementation of the Decorator pattern provides an on the fly solution to the problem of decoration. A decoration is an additional task that can be performed by the transparent filter object without involving the client or the server object. The following code presents an implementation template for a filter object-based decorator.

```
Decorator | Component {
filter interface:
       decorate () upfilters component::draw () {
           self --> decorate ();
           pass;
        }
}
```

As shown in the previous code listing, a logger filter object provides a logging member function that invokes a `decorate` member function on the decorator object. Note that in this example, the decorator object itself is modeled as a filter object. It is possible to switch decorator objects dynamically through dynamic binding of filter objects. The filter object-based solution makes it possible to plug or unplug a decorator object dynamically without involving the client and the server objects. An interesting property of the filter object-based implementation of the decorator is that neither the client nor the server needs to explicitly handle the decorator object through a pointer. The client maintains a pointer to the intended server object, whereas, the server object does not need to be evolved.

FILTER CONFIGURATIONS FOR SOFTWARE EVOLUTION

Filter object based configurations have interesting consequences to providing solutions for evolution.[11] An existing design can be adapted to new requirements using filter objects. Standalone filter objects or a network of filter objects can provide on the fly solutions for software evolution. The evolution methodology is based on the transparency properties of filter objects. The transparency properties ensure that client objects remain unaware of the existence of intermediate filter objects. Filter-based solutions to evolution may be classified in two broad categories of `adaptations` and `total solutions`.

Adaptations are applicable when objects in the system undergo structural changes. As a result of these changes, client objects that depend on the evolved objects may also need to undergo evolution. Adaptations based on a network of filter objects can prevent client code from undergoing evolution. An example of an adaptation through the Message Transformer configuration has been discussed earlier. Total solutions are those that do not involve a modification to client and server object code. A new evolutionary requirement may be satisfied completely by a network of filter objects. An example of a total solution to evolution has been described in "New Programming Paradigms for Distributed and Object Oriented Systems,"[11] with a `readers` and `writers` benchmark problem.

CONCLUSION

Filter configurations are useful in transparently carrying out activities such as routing, logging, and function replacement; caching and message repetition with least involvement of client and server code at the programming level. Filter configurations rely on the transparency properties of filter objects. Six filter configurations were described with the help of examples and implementation templates. Filter configurations can be used to implement design patterns and they are also applicable to the problem of OO software evolution.

References

1. Shaw, M., and D. Garland. *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, Upper Saddle River, NJ,1996.
2. Object Management Group, *Common Object Request Broker Architecture Specifications*, see http://www.omg.org.
3. Iona Technologies Ltd., *Orbix Advanced Programmer's Guide*, 1995. [AQ: Please provide us with more info for this reference. Author, where it was published, url, etc.]
4. Microsoft Corporation, *The COM Core Technology Specification*, 1998. [AQ: Please provide us with more info for this reference. Author, where it was published, url, etc.]
5. Aksit M. et al., *Abstracting Object Interactions Using Composition Filters*, Proceedings of ECOOP'93, pp. 152-184, LNCS-791, Springer Verlag.
6. Joshil, R.K., N. Vivekananda, and D.J. Ram. "Message Filters for Object-Oriented Systems," *Software Practice and Experience*, 27(6): 677-700, June 1997.
7. Srirami, G., and R.K. Joshi. "Filter Objects for Distributed Systems," *Journal of Object Oriented Programming*, 13(9):12-17, Jan. 2001.
8. Pree,W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, MA,1995.
9. Joshi, R.K., O. Ramakrishna, and D.J. Ram. "ShadowObjects: A Model for Service Replication in Distributed Object Systems," Forthcoming in the *Journal of Parallel and Distributed Computing*. [AQ: Do you know approximately when this will be published?]
10. Gamma, E. et al. *Design Patterns*, Addison-Wesley, Reading, MA, 1995.
11. Joshi R.K. "New Programming Paradigms for Distributed and Object Oriented Systems," Ph.D. Thesis, Indian Institute of Technology, Madras, Oct. 1996.