AspectJ Implementation of Dynamically Pluggable Filter Objects in Distributed Environment

Rushikesh K. Joshi and Neeraj Agrawal
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai, India
Email: {rkj,neeraj}@cse.iitb.ac.in

Abstract

Filter objects are dynamically pluggable first class objects which transparently intercept messages sent to server objects to which they are attached. We describe an implementation of Filter Objects for JAVA on Xerox PARC's AspectJ^{TM}. AspectJ supports the transparency properties of Filter Objects. The implementation handles objects in a distributed environment. One distinct feature of the filter object implementation is that filter objects need not be developed at the compilation time of the base system. Filtering capabilities are demonstrated through implementations of a distributed filter configurations. Various filter binding types have been introduced and their related implementation issues are discussed.

Keywords: Filter objects, Aspect Oriented programming, Implementation Model, Filter Bindings, Distributed Environment, Repeater Configuration

1 Introduction

One of the earliest filtering abstraction in a programming language is the Composition Filters model of Aksit et al. for language Sina [8], [10]. Since the development of Composition Filters, various mechanisms have been implemented for supporting filtering in programming languages and environments. Filter Objects of Joshi et al. [1] represent a suit of abstract filter object constructs for object oriented languages based on an interclass filter relationship. Filter objects are dynamically pluggable first class objects. They can be passed and returned as parameters, they have identities and they can be created dynamically. They are modular since they are instance of classes. Filter objects have special capabilities to transparently intercept message sent to server objects related to them. Filter objects, themselves being instances of classes, can also receive direct messages just as other first class objects do. They can organize their independent activities cohesively. Filter constructs have been implemented for C++ [1], JAVA [4] and MICO CORBA [6].

The filter object constructs are primarily composed of an *interclass filter relationship*, *filter member specifications*, and a *filter binding mechanism*. The interclass filter relationship makes it possible to specify filter classes and eventually filter objects. A filter class specifies filtering member functions which filter their respective server member functions when a filter instance is plugged to a server object at run time. Filter binding mechanism facilitates dynamic plugging

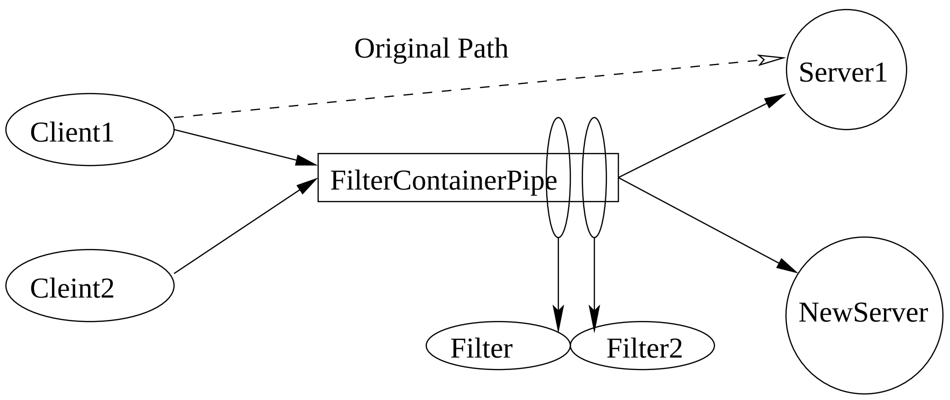

Figure 1: The Filter Model

and unplugging of filter instances (filter objects) to their respective servers. There are other extended filter object constructs which provide additional capabilities to filter objects. For example, filter objects can be layered, they can be made to filter a group of server objects. It may also be possible to switch filter member function implementations at runtime.

This work reports an implementation of Filter Objects on AspectJ^{TM}[5]. We chose AspectJ as a platform for implementation due to semantic similarities between the two suits, AspectJ being a more general paradigm for separation of concerns and Filter Objects being a specific suit of protocols for filtering. One important feature of this approach is the capability to *specify* and *attach* filter objects at runtime. Filter classes need not exist at the time of compilation of the application.

In the first section, we briefly overview the constructs of the filter object model followed by a discussion on our approach to implementation based on AspectJ. A concrete implementation of a filter configuration is presented, subsequently highlighting the underlying development life cycle for filter based applications. Some issues arising in design and implementation of filter objects in a distributed environment including bindings and deployment are discussed.

2 Filter Object Model

Filter Object model can be described in the form of following capabilities:

- *Interclass filter relationship:* A Filter class is like any other class except that it also exports a filtering interface by virtue of the interclass filter relationship. The filtering interface consists of filter members, which are automatically invoked by the execution environment as indicated by filter bindings. A filter object may also export a public interface apart from the filter interface. The public interface is available to objects that know of its identity.

- *Transparency:* Filter objects are transparent to caller (client) or the callee (server). Client and server objects need not know about the existence of the filter objects.

- *Interception of upward and downward messages:* A filter object can filter upward messages traveling towards the intended server with a member function called *upfilter*. An upfilter member may *pass* or *bounce* the message. In the case of a bounce action, the filter itself returns a result to the client on behalf of the intended destination. Also, it can let the message *pass* through to the server. Similarly, a *downfilter* member function can filter a return result on its way to client. Message arguments may be manipulated inside a filter member. A filter object may also collaborate with other classes from within automatic filtering member invocations.

- *Dynamic pluggability:* A filter object can be specified, created and attached to a server object at runtime. Similarly, a filter object may be unplugged or delinked from filtering paths at runtime.

- *Chaining and Grouping of Filters:* Multiple filter objects can be attached to one server object. They are arranged in an order, in which, their services are called. It is also possible to provide multiple filter classes for a server class. A filter object may be made to filter a group of server instances provided that the filter relationship between their respective classes is met.

3 Implementation Model

AspectJ [11] is a JAVA implementation model for separation of concerns based on Aspect Oriented Programming [5]. Filter Objects are a specific suit of protocols at the level of object oriented programming, while AspectJ can be seen as a general framework for separation of concerns. AspectJ has been chosen as a substrate for implementing Filter Objects following the guidelines listed below.

1) A filter object attachment to a server can be modeled as a *cross-cutting Aspect.* An Aspect may be used as a *Filter Pipe* to hold all the filter objects for a given sever object. A FilterPipe may be implemented in various modes such as *per class*, *per server object* or *per client-server pair.*

2) AspectJ has the capability to model message interception in a modular way, which is a requirement of the filter object model. By means of *around advice* on a call, messages can be intercepted and made to go through a FilterPipe. An *around advice* provides enough control over messages so that they can be bounced, passed and manipulated. Intermediate invocations on other objects may be carried out by filter objects.

3) AspectJ has the capability to make the FilterPipe and Filter Objects transparent to server and client. It does not require modification to the source code manually, which is a core property required for implementing filter objects.

As shown in Figure 1, A client invokes a direct method on a server object shown by the dotted line. This is the path taken in for non filterable classes. For a filterable class, the actual path is shown by solid lines. *FilterPipe* is kept transparent in the sense its identity need not be revealed to client or server objects. In the figure, the FilterPipe contains two filter objects. The pipe is attached to two server instances. FilterPipes are bi-directional. All messages to these two server instances and their return results flow through the FilterPipe.

The distributed environment for AspectJ-based filter implementation consists of collaborating objects hosted by multiple JVMs running on a cluster of workstations. Collaborations across JVMs are achieved through JAVA RMI mechanism. An important issue in filter implementations for distributed environment is the location of the filter objects. A filter object may exist on the client machine where method call to the server object is originated, it may exist on the machine on which the instance of the corresponding server class is located, or it may be located on a third-party machine.

4 An Example Implementation

The implementation is discussed with the help of an example. In the example, a requirement on an existing software is specified and a filter object based solution satisfying the same is presented. The solution is based on the Repeater Configuration discussed in [2].

Requirement: An academic registration system provides an Enroller object through which new-entrants can be enrolled into the system. While the application is executing, a new student association is started on the campus. The new requirement states that at the time of enrollment, a new-entrant should also be enrolled into the student's association. Also, the upgradation should be done without shutting down the application.

Solution: It is possible to satisfy the need in the conventional message passing model, if such a need was anticipated by the designer. Also code for the new associations has to be injected in the classes of the application which could be made possible through appropriate componentization techniques. However, in the case of an unanticipated request, there may be concerns that may not have been thought of at the time of designing and compilation. Addressing this need, we now discuss an implementation with Filter Objects handled through addition of a new Aspect can be added at runtime. Note that consistency criteria are of serious concern in such a case and we presume that the same are met through explicit coding while enabling the change. The solution is depicted in Figure 2. The implementation of the same is discussed below.

The following class is a client to class Enroller, which sends a message enroll() to an instance of class Enroller. Class Enroller is the server class to which filter object attachment is provided.

```
 public class Client{
      Enroller  enr;
      public Client() {
              enr = new Enroller();
              enr.enroll(new Student("John"));
      }
}
public class  Enroller {
       static Vector  allStudnets = new Vector();
       public Enroller(){...}
       public void enroll(Student  newS){ allStudents.add(newS); }
}
```

Filter pipe for the server class is modeled as an Aspect to add the filter attachment capability to class Enroller is given below. The aspect has two static member variables named *upFilterVector* and *downFilterVector* of type vector. These two variables hold all the filters held in the filter pipe. A filter is added to a pipe by inserting a filter object into these vectors.

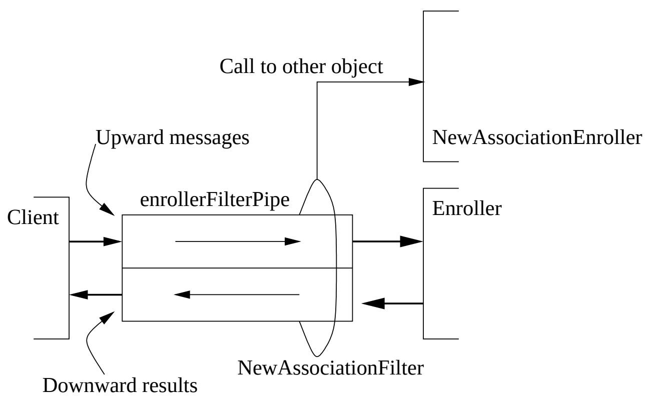

Figure 2: A Repeater

```
aspect enrollerFilterPipe{
   static Vector upFilterVector = new Vector();
   static Vector downFilterVector = new Vector();
   pointcut enrollCall(enroller  enr, String str):
        target(enr) &&  args(str)  && call(public  boolean enroll(String));
   String around(enroller enr,String str ): enrollCall(enr,str) ...
}
```

The *around advice* and *pointcut* capture a method invocation on an instance of class Enroller. Upon a message capture, it invokes the corresponding method on each filter attached in an order of their appearance in the Vector. With this invocation, it provides them with the required context information so that they can perform filter actions of *pass, bounce* and *argument modification.* This context information includes a handle to the target object, array of arguments, and an object of type Result. Upon bounce, the result is stored in this object. If the message is not bounced, original method is called with the possibly modified arguments. Implementation of *around advice* is given below.

```
String around(enroller enr,String str ): enrollCall(enr,str) {
     boolean flag = true;
     Result res = new Result();
     enrollerFilter enrollerFilt;
     Object args[] = new Object[1];
     args[0] = str;
     for(Enumeration e =upfilterVector.elements(); e.hasMoreElements() && flag; ) {
        enrollerFilt = (enrollerFilter)e.nextElement();
        flag = flag && enrollerFilt.filterenroll(enr, args, res);
     }
     if (flag) res.retValue = proceed(enr,(String)args[0]); //not bounced
     for ( Enumeration e = downfilterVector.elements(); e.hasMoreElements(); ) {
       enrollerFilt = (enrollerFilter)e.nextElement();
       enrollerFilt.filterenroll(res);
     }
     return (String) res.retValue;
   }
}
```

The base filter class for class Enroller and its concrete implementation is provided below. The abstract base class EnrollerBaseFilter is extended by every filter class for class Enroller. A filter class

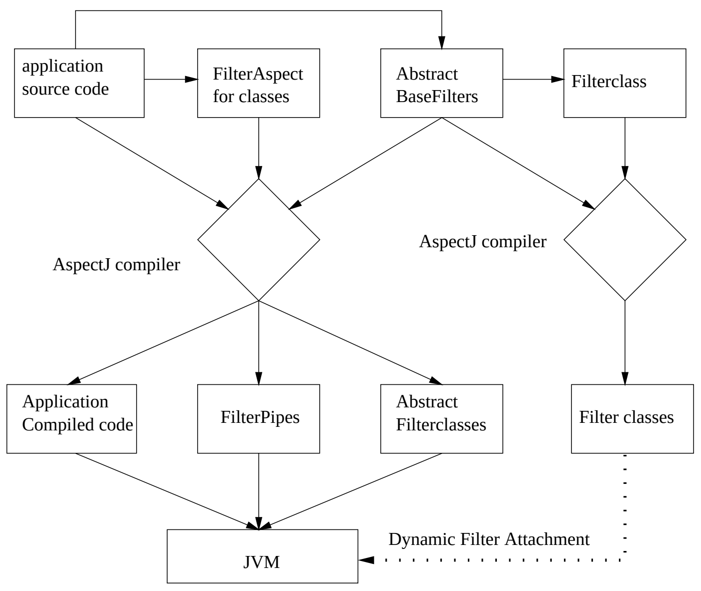

Figure 3: The Life Cycle

EnrollToOtherAssociation given below performs the additional requirement by means of a message sent to an external object.

```
interface EnrollerBaseFilter {
        boolean filterenroll(Enroller target, Object args[], Result res);
        void filterenroll(Result res);
}
class EnrollToOtherAssociation  implements EnrollerBaseFilter {
NewOrg newstudentOrg;
  public boolean filterenroll(enroller enr, object args[], Result res){
      newstudentOrg.enroll((Student) args[0]);
      return true; // message is passed
  }
  public boolean filterenroll(Result res){};
}
```

To attach a filter object to a server class, one needs to insert it in the upFilterVector in the filter pipe. Similarly, a downfilter can be attached to object by inserting a filter instance in to downFilterVector of pipe as shown in Figure 2. The client instance makes a call to an enroller instance. It goes through the filter pipe, intercepted by instance enrolToOtherAssociation, which satisfies the new requirement and passes the message through it. In case of multiple filters (layering) attached to an object, they are invoked in an order of their appearance in the vector.

5 The Development Lifecycle

Figure 3 depicts various stages in the development of a filter application. Once source code for the application is ready, classes for which filter attachment capability is to be provided are identified. For each such class, an Aspect is generated to capture method invoked on it. This Aspect may be manually written or its skeleton automatically generated. For all methods in the server class, a *point cut* with *target* and *call* primitive of AspectJ is included in the Aspect. *Around* advice is used to make

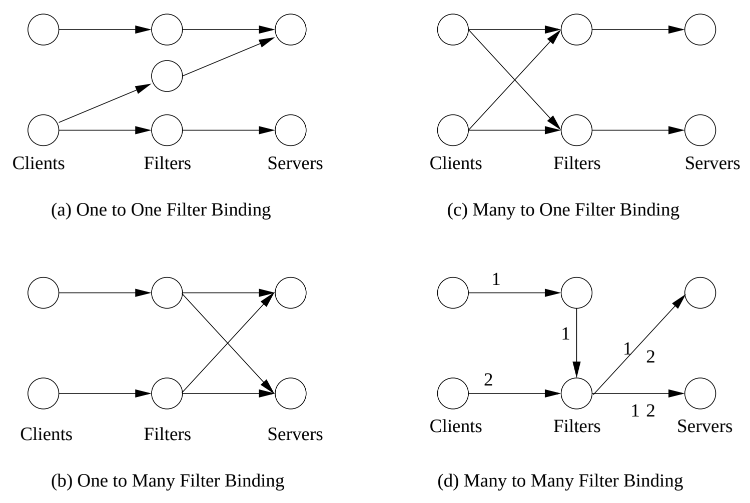
Clients
Filters
Servers
(a) One to One Filter Binding
Clients
Filters
Servers
(c) Many to One Filter Binding
Clients
Filters
Servers
(b) One to Many Filter Binding
1
1
2
1
2
1 2
Clients
Filters
Servers
(d) Many to Many Filter Binding

Figure 4: Four Types of Filter Bindings

all messages pass through the *FilterPipe*. AspectJ compiler is used to weave these Aspects into the application. After the application starts execution, a new filter extending the base filter for the target class can be compiled and attached to the target server object through *Reflection* as shown in the figure through a dotted line.

6 Filter Object Bindings

Filter instances can be categorized into four kinds called one to one, one to many, many to one and many to many based on their bindings with client and server object. These bindings are discussed below.

- **One to One Filter:** This type of filter object is bound with an instance each of server and client class. Figure 4(a) shows this type of binding. This binding is used for <per-instance, per-instance> pairing of client and server respectively. Thus, each filter object is associated with a unique pairing. It can be noted that there could be a series of filter objects associated with a pairing within this constraint.
- **One to Many Filter:** This type of filter object is bound to an instance of client, but can be bound to more than one server instances as shown in Figure 4(b). This binding represents a <per-instance, per-group> paring of client and server. Thus, a unique client object can be identified for every filter object, whereas, a filter object may act as a filter to many server instances.
- **Many to One Filter:** This type of filter object is bound to an instance of a server and many client instances as shown in Figure 4(c). This binding is identified as a <per-group, per-instance> pairing, in which, a filter is associated with a group of clients. However, a filter object is associated with a unique server instance.
- **Many to Many Filter:** This type of filter object is bound to multiple instances of server and client. It can be represented as a <per-group, per-group> pairing of client and server instances. A filter object may serve many clients and many servers as shown in Figure 4(d). In the figure, the lower filter object is of type many-to-many, whereas, the upper object is of type one-to-many.

A group in these configurations can be carved out of a server class giving a per-server class binding. Alternatively, a set of server instances may be identified to form a group. Similarly, at client side, a group may either be considered to consist of all clients of a server object, or a specialized group may be

Table 1: Typical Deployment in a Distributed Environment

One to One Filter	Client side/Server side
Many to One Filter	Server side
One to Many Filter	Client side
Many to Many Filter	Independent machine

formed. For each case, a suitable Aspect implementation needs to be chosen. For <per-instance, per-instance> pairing, the mapping to filters requires two keys consisting of client and server identifiers. Similarly,<per-instance, per-group> requires one key, which is client identifier, provided that it is a per-server class filter. If the server side group is not a class group, a group key is also required. For <per-group, per-instance> pairing, two keys are required to identify client group and server instance if client side group is not formed of all clients. If the group represents all clients, only one key is required. In the case of <per-group, per-group> pairing, two keys are required if groups do not represent all clients and server-class. Otherwise, no key is required in this case. In case of a key-based implementation, a mechanism such as *Hash table* can be used to store mappings.

7 Deployment of Filter Objects and Filter Pipes

Type of the filter object binding can be changed dynamically. They can be converted from one type to another by dynamically plugging/unplugging them to appropriate server objects. In a distributed environment, client and server objects may exist on different JVMs. A filter object in a distributed environment executes within a single JVM, but is available remotely. Typical deployment suggestions for filter objects are given in Table 1.

A location is chosen based on network overheads and also semantics of filtering. For a *one to one* filter, most efficient location could be both client or server side as it does not have any network overhead. It would be efficient to locate a *one to many* filter on the client side, since in this case, messages originate only from one JVM and having filter objects on client side reduces the network overheads. For *many to one* filter object, server side location is preferable. If located elsewhere, all method calls to the server object originating from many client objects existing on different JVMs on different machines will have to be routed through this single JVM resulting in additional network overheads. *Many to many* filter objects incur message routing overheads since calls from clients located on different machines and on the way to servers located on different machines are intercepted by these type of filter objects. The location of *many to many* filter objects may be driven by the loads and resources available on machines.

A filter pipe can be simply stated as a collection of filter objects (strictly, references to filter objects). Given a specific configuration from those discussed above, the designer may choose to implement the same over multiple pipes suitably holding references to filter objects. For example, in a non distributed environment, a *per server class filter pipe* may be implemented by modeling all its methods and member variable as class members or static members of JAVA. This saves memory overhead of having to create a FilterPipe instance. Whereas, in a distributed environment, if only one FilterPipe is designed, all messages to a server object need to be routed through the single pipe. Alternatively, a filter pipe may be implemented on multiple machines for a given server class to exploit a locality of reference.

8 Comparing Filter Objects with Composition Filters

The main difference between filter objects and composition filters is derived from separation of filter specifications from the classes they filter. Moreover, filter specifications themselves are modularized in terms of classes. The filter object model integrates the concept of transparent filtering into object orientation by means of an interclass filter relationship. An instance of a filter class is a first class object in an environment, which is a single address space or a collection of distributed collaborating objects. Filter objects may be independently and dynamically plugged into message paths.

9 Handling Feature Interaction

The feature interaction problem [3] can be briefly stated as a problem of handling undesired effects of features when they interact through a common environment. In the filter object model, it may be possible that a feature added by a filter object may conflict with a feature already supported by an existing filter object. The filter object model provides a notion of *precedences*, with which, message can be made to pass through filter objects in a predecided order. Similarly, filter objects have the capability of collaborating with each other through direct message, and also of sharing objects external to them. New filter objects may be injected into the system at runtime and an existing chain of filter objects can be dynamically rearranged through a sequence of *unplug* and *plug* operations. We believe that these capabilities offer mechanisms for designing elegant solutions to feature interaction problems.

10 Conclusions

Filter Object constructs were described and an implementation of the same based on AspectJ was discussed. Filter Object attachment capability is modeled as aspect. An example implementation of the repeater filter configuration was discussed. Four kinds of filter bindings were introduced with the related implementation issues. These are one-to-one, one-to-many, many-to-one and many-to-many bindings. The bindings are formed based on the number of client and server instances the filter objects are associated with. Per-class filters are viewed as special cases of these bindings. Filter bindings influence implementation of filter pipes and their deployment status.

References

[1] Rushikesh K. Joshi and N. VivekanandD and D. Jankiram, Message Filters for Object-oriented Systems, Software-practice and Experience, (27)6:677-699, June 1997.

[2] Rushikesh K. Joshi, Filter Configurations for Transparent Interactions in Distributed Object Systems, Journal of Object Oriented Programming, 14(2):10-16, June/July 2001.

[3] E. Cameron, A Feature Interaction Benchmark for IN and Beyond, E.J. Cameron et al., A Feature Interaction Benchmark for IN and Beyond, in Feature Interactions in Telecommunications Systems, IOS press, 1-23, 1994.

[4] Maureen M., Rushikesh K. Joshi, Filter Object for JAVA, Computer Science and Engineering, IIT Bombay, October 2000

[5] Gregor Kiczalcs, Anurag Mendhekar, Chris Macda, Cristina Vidcira Lopes, Jean-Marc Loingtier, John Irwin, Aspect Oriented Programming, In proceedings of ECOOP'97, LNCS Vol. 1241, 220-242, June 1997.

[6] G. Srirami Reddy and Rushikesh K. Joshi, Filter Objects for Distributed Object Systems, Journal of Object Oriented Programming, 13(9):12-17, January 2001.

[7] Rushikesh K. Joshi, Modeling with Filter Objects in Distributed Systems, Proceedings of Engineering Distributed Objects 2000 at UC Davis, LNCS Vol. 1999, 182-187, 2000.

[8] Lodewijk M. L. Bergmans, The Composition-Filters Object Model, Ph.D thesis, University of Twente, June 1994.

[9] IONA Technologies Ltd, Orbix Advanced Programmer's Guide, 1995.

[10] M. Aksit and K. Wakita, Abstracting Object Interactions using Composition Filters, Proceedings of ECOOP'93 Workshop on Object Based Distributed Programming, 152-184, 1993.

[11] AspectJ group, AspectJ Programming Guide, Xerox Corporation, 2001.