Implementation of Filter Configurations using Method Call Pointcuts in AspectJ

Naval Vaidya [1]

Computer Science and Engineering Department
Indian Institute of Technology Bombay
Mumbai, India

R.K. Joshi [2]

Computer Science and Engineering Department
Indian Institute of Technology Bombay
Mumbai, India

Abstract

Filter configurations are interaction patterns based on transparency primitives. The paper discusses implementations of filter configurations on top of AspectJ. Implementations of six patterns namely Replacer, Router, Repeater, Value Transformer, Message Transformer and Logger are provided. Implementations in terms of method call pointcuts and advices are provided. The implementations are also compared with those in Filter Objects.

Key words: Filter configurations, Filter Objects, Replacer, Router, Repeater, Value Transformer, Message Transformer, Logger, AspectJ, pointcut, advice, join point.

1 Introduction

Filter configurations are dynamic interaction patterns based on transparency primitives [3]. Transparency primitives can be applied to capture common aspects in transparent objects. In [3], six filter configurations with implementations based on filter objects were discussed. In this paper, the implementations in AspectJ [6] are provided. The configurations identified in [3] as replacer, router, repeater, value transformer, message transformer and logger are discussed.

[1] Email: naval@cse.iitb.ac.in
[2] Email: rkj@cse.iitb.ac.in

In a related work of Hanneman and Kiczales [1], design pattern implementations in aspect way may be found. Similar implementation of Decorator pattern on filter objects was discussed in [2]. The key difference between the former and the AspectJ implementations is that the filter object based implementations use execution time transparent objects, whereas AspectJ based implementation rely on weaving at compile time. However, it may be noted that difference does not manifest at source code level.

Filter configurations capture dynamics on messages as opposed to design patterns which cover structural configurations among collaborating objects. The focus of filter configurations is on messages. In below sections, the implementations are discussed with its related issues.

2 Replacer

A replacer may be used to replace a corresponding service partially or fully. Replacer acting on a message returns results to the caller without the intervention of the callee. In filter objects paradigm, a replacer is implemented via a filter member function that uses a bounce primitive to return the result back to the caller. Whereas, in AspectJ, a replacer may be implemented in terms of a method call join point and around advice.

In an implementation below, the replacer aspect definition for a corresponding application class is shown. The aspect captures the call for foo() from the client and call a method newfoo() on its own. Pointcut for the method call is defined. When the application's member function is dispatched, the replacer's member function is executed.

```
package replacer;
public class Application{
        public static void main(String args[]){
                Application app = new Application();
                String retstr = app.foo();
                System.out.println(retstr);
        }
        public String foo(){
                String str = "I am in foo";
                return str;
        }
}
```

```
package replacer;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect replacerAspect {
        pointcut replace(): call(String foo());
```

```
        Object around(): replace() {
                Object result = newfoo();
                return result;
        }
        private String newfoo(){
                String str = "I am in newfoo";
                return str;
        }
}
```

3 Router

A router is used to redirect requests to other objects. Transparent routers may be programmed in filter objects by means of a filter member function that invokes an outgoing call on the new destination and bounces its return result to the caller. In AspectJ, a router may be implemented in terms of a method call join point and an around advice. In an implementation below, a router aspect definition for a corresponding application class is shown. The aspect captures the call for destfoo() from the client and calls a method Newdestfoo() on the NewDest object. Pointcut for the method call is defined. When the application's member function is dispatched, the NewDest's member function is executed. Router is seen as a replacer that replaces the callee by an alternate server. The response to the call is sent by the alternate server and not by the aspect code itself. A replacer is responsible to process the message and generate a reply on behalf of the caller.

```
package router;
public class Application{
        public static void main(String args[]){
                Dest dt = new Dest();
                String retstr = dt.destfoo();
                System.out.println(retstr);
        }
}

package router;
public class Dest {
        public String destfoo(){
                String str = " I am in destfoo ";
                return str;
        }
}

package router;
```

```
public class NewDest {
        public String newDestfoo(){
        String str = " I am in NewDestfoo ";
        return str;
        }
}


package router;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect routerAspect {
        pointcut route(): call(String destfoo());
        Object around(): route() {
                NewDest ndt = new NewDest();
                Object result = ndt.newDestfoo();
                return result;
        }

}
```

4 Repeater

A repeater is used to dispatch the message to multiple destinations in addition to the original intended callee. Filter object based repeater may be implemented in terms of explicit outgoing calls to a group of additional subscribers and a pass message event to the original callee. One of the return results may be returned back to the caller. It is also possible that a return result is a function of the set of results obtained from the group. In AspectJ, the repeater may be implemented in terms of a method call join point and an and either a before advice or an after advice. In an implementation below, the repeater aspect definition for a corresponding application class with an after advice is shown. The aspect captures the call for enroll() from the client and after the call returns, it calls the same method on two additional application objects.

```
package repeater;
public class Application{

        public static void main(String args[]){
                Application app = new Application();
                app.enroll();
        }
```

```
        public String enroll(){
                String str = " I am in routerApp enroll ";
                System.out.println(str);
        }

}

package repeater;
public class application1 {
        public void enroll(){
                String str = " I am in application1 enroll ";
                System.out.println(str);
        }
}

package repeater;
public class application2 {
        public void enroll(){
        String str = " I am in application2 enroll ";
        System.out.println(str);
        }
}


package repeater;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect repeaterAspect {
        pointcut repeat(): execution(String enroll());
        after() returning: repeat() {
                application1 app1= new application1();
                app1.enroll();

                application2 app2= new application2();
                app2.enroll();
        }

}
```

5 Value Transformer

A value transformer may be used to change the contents of a message. The type of the message is not changed. In an object oriented programming language, a value transformer changes the parameters to a function call before

the call is dispatched. Typical applications of value transformer are found in encryption/decryption aspects which do not change the name of the service requested, but apply encryption/decryption on the content of the message. In filter object based solutions, value transformers are implemented by means of filter member functions that change the parameters, which are passed by reference. The filter member function subsequently passes on the message to the intended callee. Return values may be transformed similarly.

In AspectJ based implementation, a value transformer may be implemented in terms of a method call join point and an around advice. In an implementation below, the value transformer aspect definition for a corresponding application class is shown. The aspect captures the call for process_msg() from the client. After the information about the message is captured, we can transform the arguments of the message. After the message is transformed, a proceed() call is invoked to transfer control back to callee. The return result from proceed() may be transformed in the around advice before control is passed back to the caller.

```
package valTrans;
public class Application{
    public static void main(String[] args){
        application1 app = new application1();
        Object result = app.process_msg("encrypted message");
// use result
    }
}

package valTrans;
public class application1{
        public Object process_msg (String decrypted_msg){
//...
        }
}


package valTrans;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect valTransAspect {

        pointcut valtrans(): within(Application)
                                && call(void process_msg(String));
        Object around(): valtrans() {
                System.out.println(" Decrypting message");
```

```
                //here get the args from thisJoinPoint and change them.
Object result = proceed();
//code to encrypt message
return result;
        }
}
```

6 Message Transformer

A message transformer may be used to transform the type of the message. The new message is either passed to the intended destination or sent to a new destination. In filter object based message transformer, the filter member function calls the new message onto the intended callee through a pseudo handle which gives access to intended callee. If the new message is to be sent to a new destination, it is handled as in the case of router except that the call is not the same as the call intended by the client.

In AspectJ, a message transformer may be implemented in terms of a method call join point and around advice to change the destination and the message. In an implementation below, the message transformer aspect definition for a corresponding application class is shown. The aspect captures the call for destfoo() from the client and call a method Newdestfoo() on the NewDest object. When the application's member function is dispatched, the new destination's desired member function is executed. If the new destination for the changed message is the same as the old destination, around advice may be used to extract the object's id, which may be used to call the new message.

```
package msgTrans;
public class Application{
        public static void main(String args[]){
                Dest dt = new Dest();
                String retstr = dt.destfoo();
                System.out.println(retstr);
        }
}
```

```
package router;
public class NewDest {
        public String NewDestfoo(){
        String str = " I am in NewDestfoo ";
        return str;
        }
}
```

```
package router;
public class Dest {
```

```
        public String destfoo(){
                String str = " I am in destfoo ";
                return str;
        }
}
```

```
package msgTrans;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect msgTransAspect {
        pointcut msgtran(): call(String destfoo());
        Object around(): msgtran() {
                NewDest ndt = new NewDest();
                Object result = ndt.NewDestfoo();
                return result;
        }
}
```

7 Logger

A Logger may be used to log information related to a message before or after dispatches to service implementation. A logger may be implemented in terms of a call to an external logger service in a filter member function before the call is passed on to the intended destination. After the call returns, the downfilter may be used to log the return values before it is returned back to the caller.

In AspectJ, a logger may be implemented in terms of a method call join point and an around advice. However it is possible to use after or before advice also. Around advice is used for convenience of carrying out logging activity in both directions in a single advice. In an implementation below, the logger aspect definition for a corresponding application class is shown. The aspect captures the call for go() from the client and logs the information about the call. It then proceeds with the execution of the callee.

```
package logger;
public class Application {

    public static void main(String[] args){
        Application d = new Application();
        d.go();
    }

    void go(){
        System.out.println("Executing go");
    }
```

```
}

package logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

aspect loggerAspect {
        pointcut goCut(): execution(void go());
        Object around():  goCut() {
           System.out.println("Intercepted method: " +
                thisJoinPointStaticPart.getSignature().getName());
           System.out.println("in class: " +
                thisJoinPointStaticPart.getSignature().
                                getDeclaringType().getName());
           System.out.println("Running original method: " );
           Object result = proceed();
   // logging of return results may be performed here
           return result;
        }

}
```

8 Comparing AspectJ with Filter Objects

In this paper we have used method call pointcuts in AspectJ to implement filter configurations. These filter configurations can also be implemented using Filter Objects [5]. AspectJ implementation uses join points as a basic filtering abstraction. Pointcut definition is used to select a particular join point. Filter Objects are server specific, whereas AspectJ implementation is method call specific. Since it is method call specific, we can weave the aspect with any application using those method calls. Whereas, Filter Objects can only be used with the application for which it was developed. Re-usability is one of the biggest advantage of AspectJ implementation. Filter objects can be plugged and unplugged dynamically, whereas aspects need to be weaved with the application at compile time. Filter Objects provide the capability of filtering incoming messages and outgoing results using upfilter and downfilter respectively. But, They are not capable of filtering outgoing messages and incoming results. In AspectJ, the capability of filtering incoming messages and outgoing results is provided using before advice and after advice respectively, at method call reception. They also provide the capability of filtering outgoing messages and incoming results using before advice and after advice respectively, at method call. In Filter Objects, if we want to use multiple filters then we need to plug multiple filters, whereas in AspectJ, we can define multiple advice on the same join point.

9 Related Works

In this section we briefly provide some work done in this area.

- [4] discuss implementation of filter object using AspectJ. It discuss filter object constructs and gives an implementation of the same using AspectJ. It provides an example implementation of the repeater filter configuration. This implementation scheme just models the filter object attachment capability as aspect.
- [5] discuss an implementation of filter objects in terms of TJF, a filter object implementation for Java. It provides the design and implementation of a model for first-class dynamically pluggable filters for the Java programming language. It provides the implementation of Visitor design pattern.
- [2] discuss the modeling capabilities of a first class filter object model in the context of distributed systems. It provides the implementation of Decorator design pattern using filter objects.
- [1] discuss the implementation of design patterns using AspectJ. Also provides a brief discussion on the desirable properties of their AspectJ implementation.

10 Conclusion

Implementations of six filter configuration in AspectJ were provided. The filter configurations may be implemented in terms of method call join points and around advices. Filter member functions in filter objects are similar in capabilities to around advice and method call join points, except that the former relies on a weaving technique with static transformations. Filter objects being first class objects, they are useful in maintaining the first-class nature of the aspects at runtime. AspectJ on the other hand modularize concerns. AspectJ gives the capability of filtering messages and results transparently, without the knowledge of client or server.

References

[1] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[2] R.K. Joshi. Modeling with filter objects in distributed systems. In *Proceedings of the 2nd Workshop on Engineering Distributed Objects*, pages 182–187. LNCS 1999, 1999.

[3] Rushikesh K. Joshi. Dynamic modeling techniques/patterns using filter objects. *Journal on Object Oriented Programming*, 14(2):10–16, 2001.

[4] Rushikesh K. Joshi and Neeraj Agrawal. Aspectj implementation of dynamically pluggable filter objects in distributed environment.

[5] Rushikesh K. Joshi, Maureen Mascarenhas, and Yogesh Murarka. Filter objects for java. *Softw. Pract. Exper.*, 33(6):509–522, 2003.

[6] The AspectJ Team. *The AspectJ Programming Guide*. Xerox Corporation, 2001.