

The Aspect Pattern

Richard Shapiro, John Zinky, Paul Rubel
 BBN Technologies, a Verizon company
 Cambridge MA 02138
 rshapiro@bbn.com

Abstract

By dynamically chaining together a series of delegating wrappers around a collection of base instances created by Factories, a simple but effective form of Aspect Oriented Programming (AOP) with dynamic weaving can be constructed directly in ordinary OOP languages like Java and C++. This form of AOP is particularly beneficial in the handling of dynamic adaption in distributed systems.

Example

Consider the problem of adding dynamic quality of service (QoS) to the message-passing infrastructure of a distributed agent-based system built on a Component Model (see COUGAAR: <http://www.cougaar.org/>). The core structure of message passing is straightforward: find the destination Agent, choose an appropriate protocol, establish a connection if necessary, and send the data. Queues would probably be useful at several points to ensure that messages are processed in order and to prevent threads from blocking. One workable message flow structure follows:

```

Sending Agent => Send Link =>
Send Queue [per sender] =>
Router =>
Destination Queue [per destination] =>
[protocol-specific send-side processing] =>
[protocol-specific receive-side processing] =>
Deliverer =>
Receive Link [per destination] => Receiving Agent
  
```

The Send Link and Receive Link are simply service entry and exit points. The Send Queue keeps messages in order and detaches the caller's thread from the message-processing thread. The Router finds the appropriate Destination Queue for the message, and detaches the Send Queue thread from the Destination Queue threads. The Destination Queue selects an appropriate protocol for the communication, attempts to send the message with that protocol and handles retries if necessary. The Deliverer finds the Receive Link for the message. The protocol-specific processing might, as one example, resolve a remote reference using a name-server and make a CORBA remote call on the send side, and invoke the Deliverer from the CORBA servant on the receive side.

This structure is simple, modular and easy to maintain, but nonetheless provides a fairly general solution for point-to-point communication using a range of potential protocols. Although sufficient for the most common case, such a structure is also too simple for other useful cases (eg multicast) and too rigid for an adaptive system. In a real-world system it would need extensions of various sorts, to be mixed and matched under various circumstances.

For example, suppose we need to add support for multicast. One approach would be to create a special multicast-dispatch address for each process in the system. The Send Link would be modified

to detect messages with multicast destinations. When processing such a message, it would loop through the list of multicast-dispatch addresses for each process in the system, dropping a copy of the message into the Send Queue for each one. The Deliverer would likewise be modified to detect multicast-dispatch addresses, forwarding the message to each local Agent which matched the original multicast.

Or consider a dynamic adaptation that would compress data on the fly when bandwidth to the destination was lower than some threshold. In this case, the network stream over which the data is being sent would be filtered by a deflating stream in the sender's process and by an inflating stream in the receiver's process.

Extending this kind of message-passing structure by modifying the internal implementations of several of the various stations in the message flow, possibly spanning multiple processes on multiple hosts, is clearly undesirable. Not only does this violate the principle of *separation of concerns*, it quickly becomes completely intractable, given the combinatorics of potential extensions and adaptations.

One obvious solution would be to use the principles of Aspect Oriented Programming (AOP), which are specifically designed to deal with cases like this. Several AOP paradigms exist and could be used to solve this problem. Unfortunately they're not quite appropriate for large-scale distributed systems, for two reasons. First, the "weaving" of aspect code is done statically, whereas at least some crucial extensions in a distributed system are dynamic in nature, as in the second example above. More pragmatically, but no less importantly, administrative concerns may very well make it impractical or even impossible to introduce special-purpose languages into an already large and complex system.

Context

Adding modular dynamic adaptivity, eg QoS management support, to a distributed system.

Problem

Consider the problem of adding dynamic quality of service (QoS) management to a large-scale distributed system built around a Component Model (eg Java beans or the CORBA Component Model). Recognizing and adapting to changes in QoS in such a system is, almost by the very nature of the design, bound to cross-cut both the functional model (the domain-specific Components and Services), as well as the distribution model (which Components run where).

The variety and dynamism of QoS issues strongly suggests that they be treated as *Aspects* in the Aspect Oriented Programming (AOP) sense. Adding support for QoS directly into the Component model greatly complicates the design and maintenance of each Component, while at the same time obscuring the design of the QoS management itself. The principle of *separation of concerns* implies that QoS management should be handled as a set of encapsulated modules which cross-cut the Component Model.

Therefore dynamic adaption in a large-scale distributed Component system must resolve the following forces:

- Adaptation will by its nature *cross-cut* the functional Component Model. The principle of separation of concerns tells us we should modularize the adaptation into aspects rather than

- spreading it around in the functional model.
- Adaptation is *dynamic* and can't depend on the compile-time weaving of aspect code and functional-model code.
- Adaptation will cross-cut the distribution. Two or more processes must adapt in a compatible way simultaneously.
- For pragmatic reasons, the scale and complexity of the system may prohibit or at least strongly discourage the use of new programming languages or special-purpose software tools.

Solution

Introduce objects which can provide a set of delegates on demand for a corresponding set of interfaces. Use these objects to attach delegates to default implementations on the fly. Communicate the structure across the process boundary as meta-data.

In detail: Each potential Aspect will be represented explicitly as an Object which has the ability to create a delegate for a given interface when asked to do so. Factories which create default instances of particular interfaces will request delegates for a particular set of Aspects, chaining the delegates together in series. The choice of Aspects will be made dynamically and when necessary sent to remote processes.

Structure

The structural components are as follows:

- Interfaces (abstract types) for each type to which aspects can be added.
- A Factory for each interface, the one and only place where instances of that interface are made.
- Default delegation classes for the interfaces
- Aspect classes, which comprise a collection of decorator classes (extensions of the default delegation classes), a dynamic state, and a mechanism for creating delegating decorators on the fly.
- Aspect metadata, probably implemented as a sequence of string identifiers.
- An Aspect Attachment Service, which can handle aspect registration and generic aspect delegate attachment.

Dynamics

Suppose an an object of interface (abstract type) I is constructed as an instance of implementation class I_Impl by factory I_Factory, and suppose a set of Aspect instances has been defined independently. If the I_Factory uses the Aspect Attachment Service to give each Aspect in turn an opportunity to provide a delegate implementing I, the result is a cascaded series of I's grounding out in the original I_Impl.

This is a simple but powerful and dynamic form of AOP. The methods in the interfaces above define collections of *join-points* while any specific Aspect instance implicitly defines a *point-cut* (depending on which interfaces it chooses to offer delegates for) as well as *advice* for each relevant join-point (the actual code in the delegate classes). The enclosing Aspect instance provides state as well as dynamic control over the delegation.

When a call sequence crosses a host or virtual machine boundary, the current list of relevant Aspects can be passed as meta-data.

The core abstract operation on an Aspect is

```
Object getDelegate(Object delegatee, Class interface)
```

This method returns a delegate for the given delegatee which implements that part of the given Aspect which is specified by the given interface. The delegate and delegatee should both match the interface.

The core abstract operation on the Aspect Attachment Service is

```
Object attachAspectDelegates(Object base_instance,  
                             String[] aspects,  
                             Class interface)
```

This method walks through the list of named Aspects, offering each in turn an opportunity to attach a delegate of the given interface.

Implementation

A system built on a Component Model will typically contain some number of abstract interfaces, the instances of which are created in a single place, for example by a Factory (cite). Given such a structure, a simple but effective form of AOP can be implemented as follows:

- Create a collection of Aspect instances. Each such instance represents a modular, self-contained, stateful implementation of an adaptation aspect, and has the ability to provide wrapper instances or *decorators* (cite) on the fly for one or more interfaces. In Java the decorator classes would typically be represented as inner classes of the Aspect class.
- Create the Aspect Attachment Service. The central operation of the service is `attachAspectDelegates`. The implementation would walk through the list of aspect names, find the corresponding Aspect instance for each name (here we ignore the registration mechanism which would support this lookup), and give each Aspect instance in turn a chance to add a delegate wrapper by invoking its `getDelegate` method.

```
Object attachAspectDelegates(Object base_instance,  
                             String[] aspects,  
                             Class interface)  
{  
    Object instance = base_instance;  
    Aspect aspect = null;  
    Object delegate = null;  
    for (int i=0; i<aspects.length; i++) {  
        aspect = lookupAspect(aspects[i]);
```

```

        delegate = aspect.getDelegate(instance, interface);
        if (delegate != null) instance = delegate;
    }

    return instance;
}

```

- Modify the Factories to give each relevant Aspect instance in turn a chance to provide a decorator for the instance being created. The result is a cascaded series of decorators matching the original interface which wrap the original implementation instance.
- If a call sequence crosses a host or virtual machine boundary, pass *Aspect meta-data* as part of the call. This keeps the two sides in sync at runtime.
- To ease the coding of decorator classes, as well as to clarify the specifics of each decorator class, define a collection of base delegation classes for each interface to which aspects can be added. These classes, which simply delegate every method to another instance of the same type, will be used as the base classes for the Aspect-specific decorators. The latter can then be described entirely by the methods whose behavior they modify. By convention the pure delegation base classes have names of the form XYZDelegateImplBase, where XYZ is the interface for which the class provides delegation methods.

Example Resolved

Multicast

In this case the solution is quite simple. The Send Link delegate provided by the Multicast Aspect does the check and the iteration. It does this extending the default delegating implementation for SendLink and by overriding the one method that's relevant to this Aspect (sendMessage).

Note in particular the super calls: these will invoke the next delegate in the chain, or the original implementation if this is the inner-most delegate.

```

class MulticastSendLink extends SendLinkDelegateImplBase
{
    MulticastSendLink(SendLink delegatee) {
        super(delegatee);
    }

    void sendMessage(Message msg) {
        MessageAddress destination = msg.getDestination();
        if (destination instanceof MulticastMessageAddress) {
            // Remember the original multicast address
            msg.setAttribute(MCAST, destination);
            MulticastMessageAddress dst =
                (MulticastMessageAddress) destination;
            Iterator itr = findRemoteMulticastDispatchers(dst);
            while (itr.hasNext()) {
                dispatcherAddr = (MessageAddress) itr.next();
                copy = new Message(msg);
                copy.setDestination(dispatcherAddr);
                super.sendMessage(copy);
            }
        }
    }
}

```

```

        } else {
            super.sendMessage(msg);
        }
    }
}

```

The Deliverer delegate provided by the MulticastAspect handles the local dispatch. Again, does this by extending the default delegating implementation for Deliverer and by overriding the one method that's relevant to this Aspect (deliverMessage).

```

class MulticastDeliverer extends DelivererDelegateImplBase
{
    MulticastDeliverer(Deliverer delegatee) {
        super(delegatee);
    }

    void deliverMessage(Message msg) {
        MulticastMessageAddress mcastAddr =
            (MulticastMessageAddress) msg.getAttribute(MCAST);

        if (mcastAddr != null) {
            Iterator i = findLocalMulticastReceivers(mcastAddr);
            MessageAddress localDestination = null;
            while (i.hasNext()) {
                localDestination = (MessageAddress) i.next();
                super.deliverMessage(copy, localDestination);
            }
        } else {
            super.deliverMessage(msg, destination);
        }
    }
}

```

The MulticastAspect is now quite simple. It need only provide the appropriate delegate for the two interfaces of interest, ignoring any others.

```

class MulticastAspect extends Aspect
{
    Object getDelegate(Object delegatee, Class iface) {
        if (iface == SendLink.class) {
            return new MulticastSendLink((SendLink) delegatee);
        } else if (iface == Deliverer.class) {
            return new MulticastDeliverer((Deliverer) delegatee);
        } else {
            return null;
        }
    }
}

```

Compression

In this case we assume the existence of a protocol-specific factory in the sender process which returns a tcp stream connected to a server in the receiver process. Such a stream might be used for RMI, CORBA, email, etc. The Compression Aspect will attach delegates on both ends in the form of filtering streams. The sender can send meta-data to the receiver on the raw stream to let it know that it the Compression Aspect should be used to process the message data.

The details of the streams are too complicated to provide here. The important point to note is that any number of such streams can be linked together in sequence; that this linkage is determined dynamically, not statically; and that the base sending and receiving code needs no knowledge of the filters. In the sender, the selection of filters is made at runtime, on the basis of, say, the current bandwidth between the two hosts. In the receiver, the selection of filters comes from meta-data communicated by the sender on the base stream, for example as an array of Aspect names.

Note also that the runtime order of aspects has semantic significance. This can be seen clearly in the case of two stream aspects, one of which compresses the data on a stream and the other of which counts bytes on a stream for statistics gathering.

Consequences

In common with other forms of AOP, the primary benefit provided by the Aspect Pattern is separation of concerns, the ability to encapsulate the implementations of distinct software concerns. This greatly improves the maintainability and extensibility of large software systems.

The Aspect Pattern offers three additional benefits vis-a-vis other approaches to AOP:

- Dynamic code-weaving. The implementations of the various concerns are selected and combined at run-time in the Aspect Pattern, not at compile or configuration time.
- Dynamic meta-data. The dynamic selection of concerns in the Aspect Pattern can be communicated between processes in a distributed system.
- Ordinary programming languages. The Aspect Pattern is written directly in ordinary program languages like Java and C++ and requires no third-party preprocessors or code generators.

On the other hand, the Aspect Pattern has two significant liabilities:

- Delegation only. Full AOP tools are more flexible and general than the Aspect Pattern, which is restricted to implementations that can be combined by delegation.
- Modifications to existing code-base. Full AOP tools do not require any modifications to an existing code-base. With the Aspect Pattern, factories must be modified to use the Aspect Attachment Service.