The Aspect-Oriented Interceptors' Pattern for Crosscutting and Separation of Concerns using Conventional Object Oriented Programming Languages

John Zinky and Richard Shapiro

BBN Technologies
Cambridge MA, USA
jzinky@bbn.com, rshapiro@bbn.com
http://quo.bbn.com http://cougaar.org

Abstract

With disciplined use of the aspect-oriented interceptors' pattern [10], limited but effective crosscutting techniques can be used with conventional programming languages such as Java and C++. We have developed[1] this pattern for use in Cougaar [7], a comprehensive infrastructure for supporting distributed agents. Cougaar can adapt to changes in the runtime environment, supporting such dynamic features as performance tuning, security, dependability, and agent mobility. Adaptation in this context affects not what the system does, but how it does it. Adaptive features, developed by various programming teams, must be dynamically enabled at runtime based on policy assertions and resource constraints. Adaptive features touch every part of the system, hence they are said to crosscut the dominant decomposition (which is based on class hierarchies). The pattern presented in this paper helps control these features by separating them into explicit components and by allowing the components to be attached to the base system at multiple points. The pattern shows interesting use of crosscutting, not only for ease of implementation (reuse), but also for dynamic control and composition of features. The paper presents some example adaptive features to illustrate how aspect-oriented interceptors' pattern is used in the implementation of the Cougaar agent-based middleware. The paper concludes with a discussion of how the aspect-oriented interceptors' pattern compares with emerging Aspect Oriented Programming languages.

1. Introduction

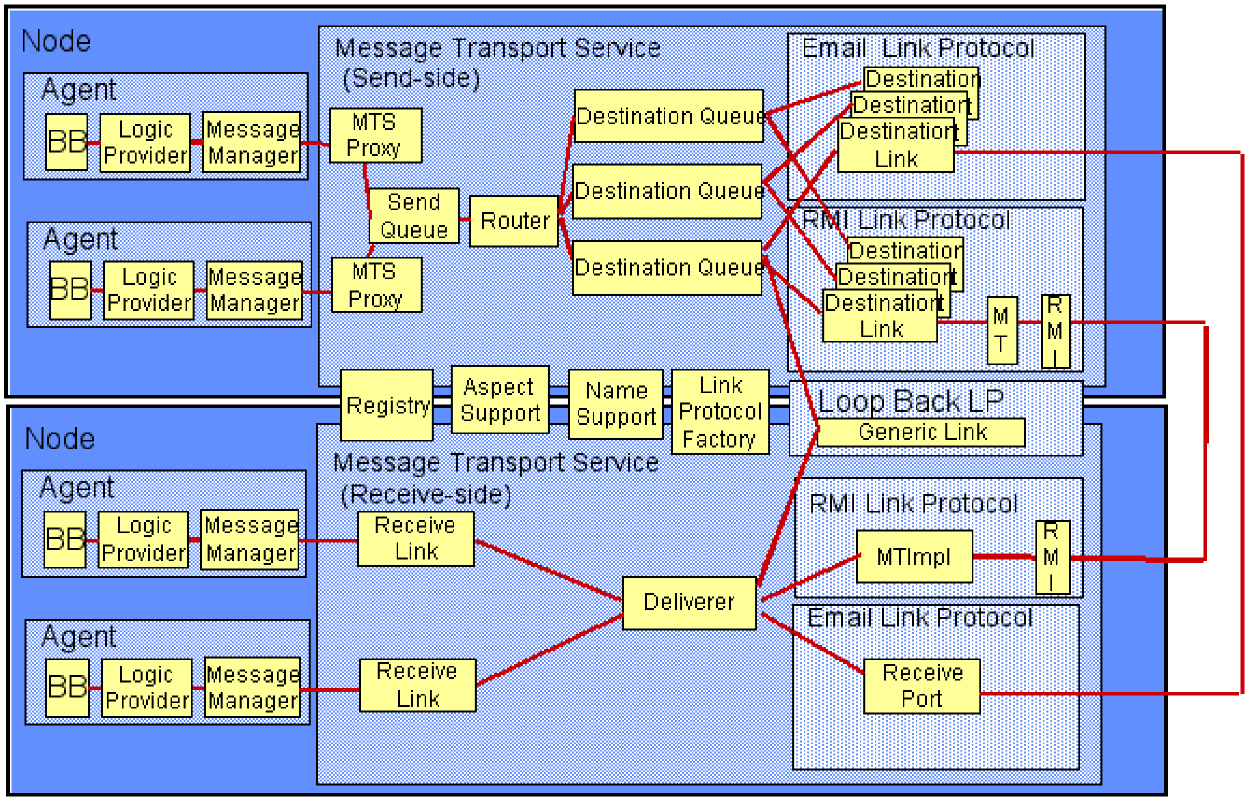
One category of crosscutting features is concerned with system issues, such as performance, security, dependability, and time constraints. This is because the application's dominant decomposition is based on the functionality of the application (what it does), and not on the system issues (how it is done). Adaptive features gather information about policies and resource constraints from many parts of the system, local and remote. The system information is used to decide how the application should implement its functionality and must be coordinated across all the relevant parts.

Crosscutting is often equated with helping the software engineering process by allowing a finer grain of code reuse [1,2,5]. In the reuse case for crosscutting, the same block of code can be woven into many different parts of the system. This results in a kind of incremental implementation of classes from many pieces. Also, it allows the different pieces to be named and hence controlled independently at code-weave time. But code weaving does not help with runtime control of crosscutting features, in fact it may actually make that job harder. For example, in AspectJ, when an aspect is added to a class, the aspect's code fragments are added to the implementation of the class. The class name does not change, but the class now contains both the base functionality and the newly woven-in feature. If at runtime the program needs to instantiate an object without the new feature, it will not be able to do so because the base class is no longer available (because AspectJ globally replaced its implemenation). Even if the weaving process made both classes available, the solution would only work for a small number of feature weaves, until the combination of classes explodes.

Crosscutting needs to be extended beyond weave-time to allow for control of adaptive features at runtime. When objects are instantiated, the instance needs to choose which features to enable. When a client gets a reference to an object, the client should be able to choose between an object instances that has the feature and one that does not. Further, at runtime, the features themselves must coordinate the interaction among the many objects that contain them, implying that features need to be first-class objects. Adaptive features must be made explicit at runtime and they need to be named. Further, they need well-defined inputs and outputs and they need to know their dependencies on other features and on domain objects. Finally, they need to know how to connect to domain objects to get information and to assert control.

Large distributed applications, such as Cougaar [7], are written in conventional programming language, such as Java and C++. These adaptive features are developed by different software-development groups and need to be enabled dynamically at runtime. If two groups need to add their crosscutting features to the same object implementation, both need to extend the base class. Even though their features may not overlap, one feature must extend the object through inheritance before the other feature. Similarly, removing a feature requires using a different class that was not extended with that feature. This implies that the program needs to define all the combinations of features, with some features in and some features out. If a feature needs to be disabled at runtime, the right class must be chosen for all the objects involved in the crosscut at instantiation time. Also, once

[1] This work was sponsored by the DARPA Ultralog Program under Contract MDA972-01-C-0025.



the object is instantiated, the feature cannot be removed without destroying the object.

2. Aspect -Oriented Interceptors' Pattern

The aspect-oriented interceptors' pattern [10] is about *controlling adaptive features at run-time*, rather than code reuse. The pattern enables control of adaptive features at multiple times in the application's runtime life-cycle. An adaptive feature is encoded as a class and created as an explicit object/component at runtime. Adaptive features decide at runtime what adaptive code to attach, if any. Also, they can expose interfaces for exchanging information among features and other external clients. But since the patterns can only add behavior to explicitly exposed places in the dominant decomposition of the application, the actual feature code is bound to the application and cannot be used in other contexts.

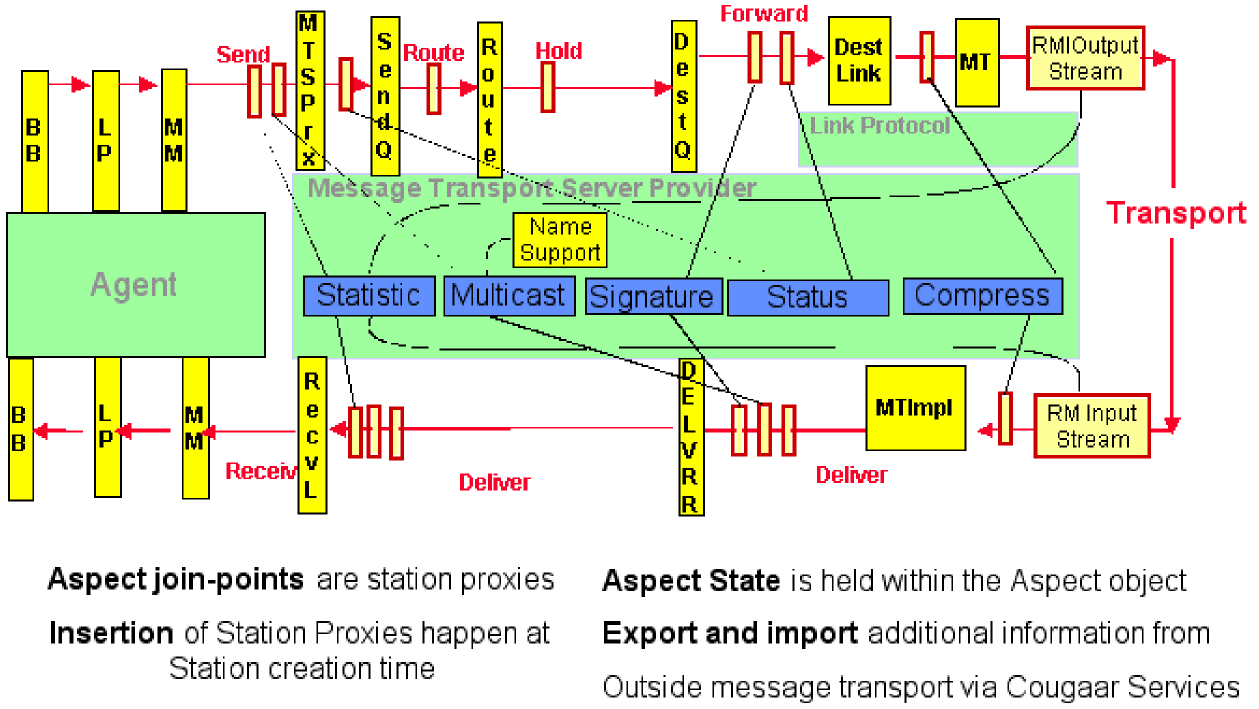
The requirements for using this pattern are very simple:

- Each of the objects over which the cross-cutting is done must have an explicitly defined interface. We call the objects that will participate in cross-cutting *Stations*
- Each Station needs a base implementation class that performs the core functionality described in its interface.
- For each interface/implementation pair, the base implementation instances should be made in a single place (effectively, a *Factory*).

An *Aspect* is then an object that can create implementation delegates for one or more Station interfaces. The Aspect class will typically include inner implementation classes for each Station interface for which the Aspect is providing a delegate. Also, Aspects may keep state.

The mechanism is very simple. Suppose the Station interface in question is *Iface* and the default implementation class is *IfaceImpl*. In the one place where *IfaceImpls* are made (i.e., the *Iface Factory*) we allow each known Aspect in turn to attach a delegate for *Iface* if it wants to. This results in a cascaded series of delegate objects, each of type *Iface*. The last object in the chain is the original *IfaceImpl*. The first object in the chain is returned as the newly created *Iface* (if no Aspect attaches a delegate, the first object will of course be the *IfaceImpl* itself). Any Aspect that wishes to attach a delegate to *Iface* would then define its own *Iface* implementation class, typically as in inner class.

One problem with this simple mechanism is that the Aspect delegates always run in the same sequence. This ordering is too restrictive. For example, in a communication subsystem, a common paradigm is that the work done in the sender needs to be accommodated in reverse order in the receiver. To handle this we added a bit of complication to the attachment of delegates. Aspects are in fact given two opportunities to attach delegates. One set of delegates will run "forward" (i.e., earlier delegates will run before later ones), the other will run in "reverse" (earlier delegates run after later ones).



Aspect join-points are station proxies

Insertion of Station Proxies happen at Station creation time

Aspect State is held within the Aspect object

Export and import additional information from

Outside message transport via Cougaar Services

3. Example Application: Cougaar Message Transport

Cougaar [7] is a comprehensive infrastructure for supporting distributed agents. Cougaar Message Transport Service uses the aspect-oriented interceptors' pattern to add adaptive features to communication among agents. Cougaar is a large system with over 500k lines of code developed by several groups across several independent projects. Ultra*Log is one such project, designed to make Cougaar robust in the face of chaotic changes in its environment such as simultaneous system failure, security attacks, and global shifts in usage patterns. Using dynamic adaptation to manage the interaction among agents is a key feature that Ultra*Log will be adding. The Cougaar Message Transport Service will be the locus of much of this adaptation, and will be developed simultaneously by various groups.

The Message Transport architecture was designed to manage the flow of messages among agents. The internal design is open, i.e., it consists of a number of abstract interfaces with a variety of implementations. The constituents of the Cougaar Message Transport are analogous to CORBA interceptors and pluggable protocols; that is, new communication features can be added without modifying the base code. Messages flow through the abstract constituents, or "Stations", in a predefined sequence, but the behavior at each Station is determined dynamically.

The Cougaar Message Transport is divided into a dozen such Stations. In the simple case, in which all the Stations provide their default behavior and in which no errors have occurred, a message flows from one Station to the next with minimal processing. In this case the complexity of architecture may look like overkill, since most Stations act as pass-throughs, at most with buffering. In the real world, errors can occur and changing system conditions can cause the default Station behavior to be inadequate. In this more realistic case, the Stations need to do their jobs differently, but in a coordinated way. Combining the Aspect pattern with the Message Transport's open implementation provides a clean solution via crosscutting. In other words, an Aspect provides code that cuts across the dominant decomposition provided by the Station interfaces.

The Stations are described in the "top view" diagram (see Figure 1). While strict layering is not used, Stations can be grouped into the traditional communication layers. Stations handle issues for end-to-end (on the left in the figure), routing (middle), link protocols (right). Note that the "physical layer" is made from full communication stacks, such as RMI, CORBA, Email, or raw sockets. Adaptive features in the message transport tend to reimplement the classic communication services, such as addressing, flow control, retransmission, etc., but also employ extra knowledge from the host and the application domains. The reimplementation of lower-level services by higher-level services is an ongoing issue addressed by several technologies, such as micro protocols . We use crosscutting to insert the features across multiple layers.

4. Example Aspects

Cougaar has over 20 Aspects implemented, which handle a diverse set of adaptive features. Cougaar applications can be configured to include any of these Aspects to match the external requirements of the system. Hence Cougaar can be configured to run as an embedded controler with minimal functionality, or as a robust distributed system with security, robustness, and performance-tuning features enabled.

The following examples show some different uses of Aspects. The "side view" (Figure 2) shows how these Aspects could be combined into a specific system configuration. Note how various Aspects insert themselves into the message flow at various Stations. Aspects allow an adaptive feature to obtain access to the parts of the message flow where it needs to add behavior, and to ignore the rest.

4.1 Message Statistics

Instrumenting code for debugging is the classic AOP example[6]. All the trace and logging code is removed from the Stations and placed in Aspects. The observation Aspects can be hooked into any of the Stations and can correlate measurements across interfaces. Also, the summary of the observations is kept in the Aspect state. The Aspects can expose service interfaces so that their observations can be viewed by external components or other Aspects.

4.2 Message Multicasting

Message Multicasting detects the Multicast message type and forwards it to all the agents in a society. The Multicast messages are expanded at multiple levels. First the message is sent to all the nodes in the society and then to each agent in the node. Thus, Multicasting has to insert itself at many Stations, to convert message types, to look up the addresses of remote nodes and local agents, and to copy messages. Some of these tasks happen when Multicast messages are sent and others occur when agents register with its node or move. Thus, Multicasting crosscuts the Station decomposition.

On the one hand, Multicasting is a single, fairly simple, concept. One would expect a good software design for Multicasting to be implemented in a single class. Otherwise it's a nuisance to maintain. On the other hand, a typical message-handling system would handle sending in one class and receiving in another, for all the usual OOP reasons. Since Multicasting requires changes both on the sender side and receiver side, we cannot use traditional OOP to implement it unless we are willing to violate the first point (i.e., keeping the Multicasting code as a self-contained unit).

The Aspect pattern resolves this difficulty. By implementing Multicasting using an Aspect, the core message-handling code remains simple and stable, while all the Multicasting code lives in a single place where it is easy to maintain.

4.3 Message Serialization

One of the Stations exposes an interface when a message is serialized or deserialized. Different read and write filters can be added dynamically, for such things as encryption, compression, signiture, and byte-counting statistics. Some of these serialization features need to be added at both the sender side and the receiver side. For example, a compression Aspect may want to add compression to the message serialization, when the message goes to a destination that has a low bandwidth path, but not to other destination that has a high-speed connection. The Aspect must tradeoff CPU cost to compress the message against the savings in bandwidth. When compression is used, the sender-side Aspect must signal the receiver-side Aspect to add the decompression filter to the deserialization Station. Signaling is done by adding an attribute to the header of the message. The message header actually carries a list of the Aspects to be called to add filters on the receiver side. So the sender and receiver Aspect instances cooperate to dynamically add behavior to the system.

4.4 Heard-From status

Determining if the connection to a remote host/agent is work requires correlating information that comes from many sources. One indicator may be receiving a message from a remote agent, implies that the agent's host and communication path is working. Likewise, when an acknowledgment is received for a message sent to an agent. The heard-from Aspect inserts itself in the connections to multiple agents, determines agent's host and maintains state about when the last time the host was heard-from.

5. Comparison to other AOP technologies

The Quality Objects (QuO) Project [8] builds adaptive middleware for distributed and embedded systems. The QuO middleware offer support of QoS adative behavior at both design time and runtime. The QuO is is used to implement some of the Cougaar Aspect, helping to structure the implementation of Service Proxies and Aspect Delegates But QuO has no direct support for Cougaar Aspect objects themselves. QuO needs to be extended to handle bind-time issues and managing Aspect state based on information gathered from multiple delegates.

Aspectual collaborations [2] extend the concept of advice by allowing aspects to be parameterized over the class and method names that are to be advised. This extension is important for connection aspects, where connection patterns may be reused several times between Cougaar Stations. Collaborators are analogous to reusable Cougaar Aspect objects. The Collaboration roles are like the specification for where the Aspect inserts its delegates. Also, Collaborations can have their own state, just like Cougaar Aspects. The advantage of Collaboration is they are like templates that can be bound to different interfaces. Hence Collaborations could be used to reuse adaptive behavior between Cougaar Aspect that modify a specific Link Protocol, such as email or RMI.

AspectJ [5] implements aspects as wrappers (called "advice") that can be executed before, after, or around program points such as method calls. The Cougaar Aspect delegates are similar, but they can only wrap the interaction at the server side and not where the call is made (client-side) like AspectJ. AspectJ can also insert advice at finer granularity, than just the predefined Station interfaces.

Subject-oriented programming [4] and its derivative HyperJ [3] are other AOP approaches. Hyper-J allows the extraction of classes from an existing class decomposition. This allows would allow the extensions to Station classes to be developed independently and combined relatively easily. Hyper-J class are composed at class loading time, which would allow some dynamic composition. But Hyper-J does not support for adaptation at service lookup time or message forwarding time.

Composition Filters [1] are similar in some respects to Cougaar Aspect delegates, providing wrappers for class methods that can change class behavior. Composition filters compose with formal semantics so that they can be used to infer the composed properties from its pieces. This is needed when Cougaar Aspects begin to interact in more complicated situations. However, they lack some of the features of Cougaar Aspects: the ability to measure and react to external, systemic conditions and coordination among filters that are inserted at several Stations.

6. Conclusions

The aspect-oriented interceptors' pattern developed for the Cougaar distributed agent system, allows the message transport to be extended by multiple development groups without modifying the base code. Dynamic adaptation at runtime is supported by exposing multiple times and places in the code base for which adaptive code can be inserted. Unfortunately, coordinating this code is crosscuts the dominate decomposition and new patterns were needed to keep the code maintainable and to enable the dynamic adaptation at runtime. While current AOP techniques hold promise for improving the maintainability of the crosscut code, they offer very little support for runtime adaptation. We hope that this paper will show a real world application of crosscutting and an interesting pattern for dealing with it. We hope that future programming languages will support dynamic crosscutting

7. Bibliography:

[1] Bergmans L, Aksit M. "Composing Multiple Concerns Using Composition Filters," Communications of the ACM, special issue on AOP, October 2001.

[2] Lieberherr K, OvlingerJ, Mezini M, and Lorenz D, "Modular Programming with Aspectual Collaborations", College of Computer Science, Northeastern University , Tech report NU-CCS-2001-04, March 2001

[3] Ossher H. and Tarr P, "Using Multidimensional Separation of Concerns to Reshape Evolving Software. CACM) Oct 2001, pp 43. http://www.research.ibm.com/hyperspace

[4] Ossher H, Kaplan M, Katz A, Harrison W, Kruskal V. "Specifying Subject-Oriented Composition," Theory and Practice of Object Systems, Vol. 2, No. 3, Wiley & Sons, 1996.

[5] Kiczales G, Hilsdale E, Hugunin J, Kersen M, Palm J, Griswold W. "An overview of AspectJ," Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2001.

[6] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Grisold W, "Getting Started with AspectJ" CACM Oct 2001, page 59.

[7] Cougaar Distributed agent system, open source at http://cougaar.org

[8] Zinky JA, Bakken DE, Schantz RE. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, April 1997. http://www.dist-systems.bbn.com/tech/QuO

[9] Ultra*Log DARPA Program on Logistics Information System Survivability, http://www.ultralog.net/

[10] Shapiro R, Zinky J., Rupel P. The Aspect Pattern. OOPSLA 2002 Workshop. Patterns in Distributed Real-time and Embedded Systems, November 5, 2002, Seattle, Washington.