

Building Customizable Middleware using Aspect Oriented Programming

Frank Hunleth, Ron Cytron, and Christopher Gill *

{fhunleth, cytron, cdgill}@cs.wustl.edu

Department of Computer Science

Washington University, St.Louis, MO 63130, USA

1 Introduction

Traditional Distributed Object Computing (DOC) middleware, such as CORBA [1], COM+ [2], and Java RMI [3], is designed and built to provide a wide feature set to suit the needs of multiple problem domains. This technique increases the appeal of using the middleware, but perhaps at the expense of supplying extra features that may contribute to unnecessary code bloat and configuration complexity for some applications. Extensible middleware such as The ADAPTIVE Communication Environment (ACE) Object Request Broker (ORB) (TAO) [4] tries to contain such bloat by extensively strategizing features within the code base. Currently, such specialization is accomplished using one of the following approaches:

- The selectable features are preconceived, and the code is developed initially with such selectivity as a design requirement.
- The selectable features are identified *after* the corpus of code is developed. The original code must then be refactored to achieve the desired selectivity of features.

Both of these approaches been used within the TAO project. The pluggable-protocol framework [5] allows customization of TAO with respect to protocols—a small-footprint, low-functioning protocol may suffice for some applications. On the other hand, the real-time features of TAO were recently refactored out of the main corpus, to obtain smaller footprint where real-time concerns are absent. The need for refactoring is not surprising—until the need for a selectable feature arises, it may go unnoticed that such selectability exists.

Both of the above approaches have big disadvantages. Pre-conception of *all* selectable features is very unlikely: inevitably, an application will arise in which some feature need not be present. While frameworks such as ACE ease the task of making features pluggable, refactoring existing code to obtain selectable features is still a time-consuming, error-prone activity. In part, this is because many feature-hooks *crosscut*[6] much of the core code, making code maintenance more difficult.

Customization resulting from the above approaches is also somewhat unsatisfactory. Hooks remain in the core code and null strategies typically substitute for unused features. The resulting code contains remnants of excluded features, which increases the code size and complicates human understanding of the code.

To address these shortcomings, we are taking an alternative approach to developing middleware—we use *aspects* [6] to introduce features incrementally. Additionally, we aim to make features as independent as possible; this allows middleware

users to reconfigure the software to meet their needs without adding any extraneous interfaces, code bloat or null strategies. At a high level, each feature is implemented as one or more aspects. Through the use of a configuration file, the appropriate aspects can then be woven through the base middleware code (and possibly user code) to achieve the desired configuration. In support of this kind of customization, the middleware can export an aspect-oriented Application Programming Interface (API) to the user for additional configuration control.

Using this approach to implement flexible and extensible middleware presents several novel challenges:

1. *The base code should consist of a subset of features common to all configurations.* To obtain the smallest footprint and the simplest implementation, we seek a small *common* code base. Aspects support this approach due to their additive nature.

However, taken to an extreme, this would reduce the base implementation to contain only the sufficient structural elements to allow for the specification of *join points*—well defined places in the execution of the program [7]. The behavioral elements would then be contained separately and distributed in various aspects. This extreme is counter to our desire to have the base code provide some basic functionality. Therefore, the granularity of the abstractions in the base code should be somewhat coarser than the needs of the aspects' join points, to keep complexity of the aspects themselves in check.

2. *Features should be selectable solely based on user requirements.* The aspects that implement these features should be managed in such a way that the user is shielded from the aspects' interdependences.
3. *The number of features and their aspects should be scalable.* We would like to be able to introduce new functionality without explicit concern for all of the possible user configurations.
4. *A framework should exist to be able to test every possible configuration of the middleware.* Given a level of testing present for the code base (*e.g.*, correct event notification), that same level of testing should be available for all configurations that can be generated from the code base and aspect suite. At the same time, developers who write the tests should not have to concern themselves with satisfying every possible configuration. They should focus merely on a simple configuration that has their feature, and the test framework should take into account any other configurations that may impact the test.

To investigate the use of aspects to build customizable middleware, we are implementing a real-time *event channel* in

* Sponsored by DARPA under contract F33615-00-C-1697

AspectJ[7], an aspect language built on top of Java[8]. In its simplest form, an event channel is a middleware component that acts as an intermediary between event suppliers and event consumers. The event channel exemplifies both the Mediator and Observer design patterns [9], decoupling supplier and consumer processes and their interactions. Event channels are frequently found in DOC middleware, and the one implemented here draws on concepts from three Common Object Request Broker Architecture (CORBA) event services that have been implemented in TAO. These are the Object Management Group (OMG) Event Service [10], the OMG Notification Service [11, 12], and the TAO Real-Time Event Channel (RTEC) [13].

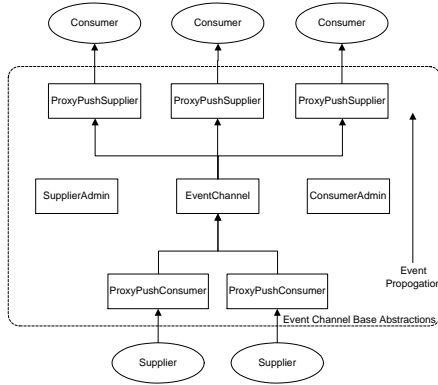


Figure 1: Classes involved in the base event channel.

A depiction of the classes involved with a base event channel implementation is shown in Figure 1. The `EventChannel` class provides the event forwarding functionality. Subscriptions to the event channel are handled indirectly through the `ConsumerAdmin` and `SupplierAdmin` classes. Both of these classes are factories for `ProxyPushSupplier` and `ProxyPushConsumer` objects to which the actual suppliers and consumers use to connect to the event channel. Note that even this simple abstraction allows for multiple event channels to be formed into federations by connecting `ProxyPushConsumer` instances with `ProxyPushSupplier` instances on other event channels.

In addition to providing basic event forwarding services, the RTEC provides various types of event filtering, scheduling, correlation, and dispatching mechanisms[14]. In both the standard event channel and the RTEC, event distribution can be local or distributed.

The remainder of this paper describes our initial experiences applying aspects to a Java real-time event channel and is organized as follows: Section 2 describes the organization of our build environment to manage the number of aspects used in the event channel; Section 3 documents an initial design pattern to manage API changes among aspects; Section 4 describes the effect of using aspects on interfaces specified using the CORBA Interface Definition Language (IDL) and how we deal with this effect; Section 5 describes the verification involved with checking that a user specified event channel configuration results in a consistent set of aspects being applied; Section 6 describes the effect that using many independent aspects has on the testing process and our method of handling this additional complexity. Finally, Section 7 describes our

planned future work applying aspects to flexible middleware components and services.

2 Aspect Configuration and Build Environment

In their simplest form, Aspect-Oriented Programming (AOP) programs contain independent and relatively few aspects that are easily managed. For example, aspects that trace execution are independent of aspects that introduce memoization into a program or that select different forms of synchronization. Aspects “in the small” are easily managed because the number of combinations of meaningful programs constructed by aspect inclusion is also small. AspectJ provides a simple file-list facility so that combinations of meaningful programs and aspects can be prespecified.

Middleware is different in that it must target many application domains that have different feature requirements. As a result, the number of aspects and their dependence structure is sufficiently complicated so that the number of meaningful combinations is quite large. For the event channel, prespecification of each meaningful combination could result in over 100 lists, which would be difficult for any user to manage.

Therefore, the actual building of the event channel software needs to be organized to support applying the various aspects in a simple and straightforward way for both developers and users. This section presents our initial experience in this area and also serves as background for some of the subsequent sections.

The standard mechanism for invoking the AspectJ compiler, `ajc`, is to specify a file that contains all of the Java source files that should be compiled. This facilitates compilation a set of files and construction of file-lists depending on the aspects that should be applied. However, this mechanism does not scale well for the number of configurations that are likely to be supported by the event channel. In effect, a list file would need to be created for every possible configuration or a script would be needed to generate the desired list file for a particular configuration.

We chose to use the `ant` build tool to determine the necessary Java source files for the desired configuration. The `ant` build tool is an alternative to the `make` program to direct program compilation. One of its main advantages over `make` is that it has built-in support for many of the tasks needed to build Java programs. A second advantage is that building customized file lists is straightforward and easy.

We constructed the directory structure for the event channel code by placing the base implementation in the base directory for its Java package. Code for each atomic feature’s aspect source code is placed in separate directories off this base directory.

`ant` allows users to specify properties to be set or loaded at any point in the build process. These properties are similar to `make` macros with the exception that once they are set, they cannot be changed. For the event channel, we have specified a property file that specifies which features should be enabled. In the main `ant` build file, these property settings direct which subdirectories get scanned for Java source files, and hence get used when building the event channel.

3 Using Parameter Classes to Support Feature Addition using Aspects

One recurring pattern in the development of the event channel is the usefulness of defining parameter-holding classes. We call this the Encapsulated Parameter pattern. The design forces that motivate this are (1) the need to add parameters to method calls to support some features and (2) the need to keep the API simple. For example, one way to add a feature is to introduce a new method with an additional parameter that could have a default value. An important issue arises when two features are needed that must add different parameters to a method: the method's signature must be derived appropriately when this occurs.

Our approach creates a parameter-holding class for the each event-channel method that may be passed different parameters based on the current aspect configuration. Because aspects are introduced at compile-time, the presence of a parameter can be guaranteed. Thus, there is no need for dynamic lookup of parameter values based on their name; instead instance variables can be introduced by aspects that bear the name of a parameter. From the perspective of the aspect developer, this allows for the easy introduction of new parameters. Additionally, the parameter-holding class constructor can set these parameters to defaults so that the user's code need not concern itself with them. Of course, if this were the general case, the user should change the configuration to exclude such features. This solution also fits nicely into the event-channel framework. The two main methods that use this pattern are the push method to send events and the consumer registration method.

In the case of the push method, the instance variables add support options such as event source, type identification, and Time To Live (TTL). For example, the following classes are used as the base implementation for sending an event that has a header and body.

```
public class EventHeader {
}

public class Event {
    public EventHeader header;
}
```

Aspects can then be used to introduce additional fields into the above classes to support additional features and add the type contained in the body of the message. The following aspect was used to support the TTL feature.

```
aspect TtlAspect {

    public int EventHeader.ttl;

    /* Advice to check and decrement
       the TTL field */
}
```

The second principal use in the event channel is for consumer registration. Depending on the configuration of the event channel, consumers may want to establish filters and register their real-time processing behavior. Since event filtering and real-time processing features may both affect the temporal

properties of events, the aspects that introduce them may need to both (1) introduce their options in the parameter block and (2) register hooks as described in Section 5, to support higher-level aspect consistency mechanisms.

Another benefit of using the Encapsulated Parameter pattern is that it simplifies the *advice* (code associated with each join point[7]) that needs to be written, since a parameter class instance can be passed around inside the advice and queried accordingly. The parameter class, which may be empty in the base case, also provides a convenient hook to use as a join point.

Finally, a countervailing force raised by introducing parameter classes is that they may potentially make the less-featured versions of the event channel more difficult to use. In these cases, the parameter classes have very few or no instance variables, which makes their presence seem superfluous. For example, an event channel user may only need to send a payload from suppliers to consumers and does not need any event header fields. Using the `Event` and `EventHeader` classes above requires that an `Event`, `EventHeader`, and a payload class instance be created for every event sent. From the user's standpoint, the `Event` and `EventHeader` classes are just part of the cost of using the middleware.

In the event channel, we can minimize the effect of this force by having the push method in the base version take no parameters, and then using simple aspects to introduce either the parameter classes (`Event` and `EventHeader`) or just a payload class (`java.lang.Object` or a CORBA `Any`). Note that since the base code does not pass event data at all, it is more like an interrupt notification service. This base reduction of functionality could become a concern if it begins to no longer look like the code for an event channel.

4 AspectIDL

Since the event channel can use CORBA to send events from remote suppliers and remote consumers, it needs to specify its interface definitions via CORBA's IDL. This has the added advantage that remote event-channel clients need not be written in `Java` to take advantage of a specifically tailored event channel. A limitation of the IDL language is that it currently lacks an aspect preprocessor. As a result, we have written a primitive script to insert IDL definitions into the base event-channel IDL description using a search and replace style mechanism. We call this script the `AspectIDL` preprocessor; as future work, we plan to develop a more robust implementation by augmenting an IDL compiler with the ability to parse aspect introductions.

Since IDL only allows the specification of interfaces, `AspectIDL` conceptually supports the following kinds of introductions:

- Interface method and field introduction
- Interface super class introduction
- Structure field introduction
- Oneway method specifier introduction
- IDL typedef and enumeration introduction

The use of IDL for the event channel brings up another form of crosscutting concerns, namely concerns that crosscut programming languages. For example, when a public interface or a new field in the event header is introduced as the result of a feature, it changes both the IDL declaration and the Java code. It would be desirable to include the IDL introductions together in the same file with the Java introductions and the advice that implements it. Even though the event channel IDL introductions mostly apply only to one class or structure, it would be useful to introduce methods across many IDL classes, as can be done using AspectJ. Possible applications for this include adding monitor registration points on each class to enable event channel monitoring tools to register observation callbacks.

5 Aspect Consistency Checking

It is important to determine whether a given set of features is meaningful, in terms of yielding a viable event channel, since a user should be able to enable and disable features at will without going unwarned about configuration errors. These errors can be of several types.

- The aspects of some features depend on the presence of other aspects, as they may apply additional advice to them. It is an error if the aspects being depended on are missing. We would like to catch this as soon as possible but cannot depend solely on the AspectJ compiler. This is the case since the missing aspect may have added a crucial join point to the code that is needed by the dependent aspect. Alternatively, if the AspectJ compiler does catch the error, it may generate a misleading error message. *E.g.*, the dependent aspect may use a public method or instance variable that would have been introduced by the missing aspect.
- Two mutually-exclusive features might be selected. The aspects from both will be applied, but the resulting code will almost certainly not perform as expected. For example, consider the choice of *payload type* for an event channel. For many CORBA applications, a CORBA Any type is sufficient to carry the payload, and it has the advantage of matching the type used in the standardized CORBA Event and Notification Services. A mutually exclusive alternative to the Any is the structured-event message type, which includes a predefined message header. This latter option is useful, since it exposes important event fields to the event channel such as the message source, type, TTL, etc that the event channel can use internally. We would like to prevent selection of these two features at the same time.
- At run-time in a distributed environment, it is possible that the code from two event channels may interact. Therefore, it is desirable to check that both channels have compatible configurations through an initial handshake mechanism. Ideally, if such a misconfiguration exists, appropriate aspects could be woven to make both event channels consistent. Currently, we assume that all event channels that communicate with each other have compatible configurations and leave as future work the ability

to verify configurations at runtime and weave aspects remotely to resolve mismatches.

To solve the above problems, the `AspectRegistry` class was added to the base code. The intention of this class is to maintain a list of all aspects that have been woven into the event channel, their dependence relationships, and any mutually-exclusive relationships. The `check` method can then be used to check all of the relationships between the aspects for consistency. Event channel aspects then advise the `buildList` method of `AspectRegistry` to register their consistency information.

Those aspects that only need to register their presence with the `AspectRegistry`, can simply extend the `AutoRegisterAspect` abstract aspect which is shown in Figure 2. By concretizing this aspect, the proper advice gets added to `AspectRegistry`.

```
abstract aspect AutoRegisterAspect {
    after(AspectRegistry ar) :
        executions(void ar.buildList()) {
            ar.registerAspect(
                this.getClass().getName());
        }
}
```

Figure 2: Automatic registration aspect

Dependence relationships are currently registered in the concrete aspects themselves with advice as above. Mutual exclusion relationships also use aspect inheritance from a common “mutual exclusion” class to automatically register a concrete aspect with the `AspectRegistry`. `EventTypeAspectMutex` is an example of a mutual exclusion and is shown in Figure 3.

```
abstract aspect EventTypeAspectMutex
    extends AutoRegisterAspect {
    after(AspectRegistry ar) :
        executions(void ar.buildList()) {
            // Register this aspect as holding
            // the EventTypeAspectMutex.
            ar.registerMutex(
                this.getClass().getName(),
                "EventTypeAspectMutex");
        }
}
```

Figure 3: Abstract aspect for event type aspects

Finally, since aspect consistency checking involves very little code (only the `AspectRegistry` class and the feature aspects are needed), it can be done separately from the main build process. This allows the event channel build-system to validate the configuration and prevent compiletime and runtime failures. In the future, we intend to use the dependence and mutual exclusion information in the `AspectRegistry` class to generate a list of all possible valid event channel configurations, and then automatically build and test each one.

6 Automated Testing

An important element of any software development effort is the use of automated unit-testing. The use of aspects to de-

velop customizable middleware makes this even more important, especially as the number of possible software configurations grows. This presents two main issues:

- For any configuration, only those tests that exercise enabled features should be run.
- Tests should be written to work on the minimum possible configuration that has the desired feature to reduce the number of issues that might arise.

Using aspects proved to be very convenient in addressing both of these issues. The resulting software configuration can be depicted as in Figure 4 where aspects are used both in the configuration of the middleware and in the configuration of the test framework. In the figure, the feature aspects have both implementation and test framework pieces. As expected, the implementation can be used independent of the test, but the test framework and the test aspects both depend upon the implementation.

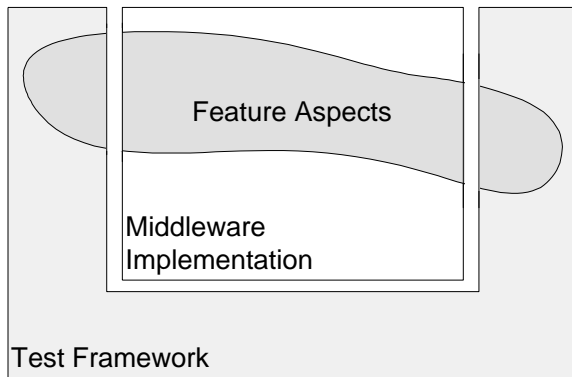


Figure 4: Aspect use in the test framework.

The event-channel implementation uses the `jUnit`[15] test framework for organizing and running test cases and detecting and reporting error conditions. The basic procedure for creating a `jUnit` test is to create a subclass of the `TestCase` class and create methods to test one or more conditions of a feature. These `TestCase` subclasses are then aggregated in a subclass of `TestSuite`. The `jUnit` test runners then take a `TestSuite` instance and run all of the `TestCases` within the suite.

To determine which unit tests to run for the current configuration, the `TestCase` subclass for a feature is placed in that feature's directory. In the same file that contains the test case, an aspect registers the `TestCase` subclass with the common test suite. By convention (and for the event channel), the common test suite is the `AllTests` class.

For example, one of the features of the event channel is to add a `TTL` field to the event structure. This feature has an aspect that adds the necessary advice to check and decrement the `TTL` field for each pass of that event through the event channel. The unit tests for this feature simply test that the `TTL` field is appropriately decremented and events are dropped when the field is zero. These tests are stored with the `TTL` feature aspects and before advice is used to on the main test suite to register the unit tests.

The use of aspects to attach test cases has several consequences:

- Aspects localize the unit test registration to the unit tests themselves. Without a mechanism to reflectively find appropriate tests, adding a unit test would always involve modifying two files—the test case and the containing test suite. From a software engineering standpoint, this ability to maintain all of the code involved with a unit test in one file is advantageous in terms of code readability and maintenance. It also helps avoid the common mistake of forgetting to register new unit tests or deregister outdated tests. Comparing the use of aspects to add tests to a meta-programming technique of scanning all unit tests and querying them to see if they support the specified configuration, the former appears both simpler and more efficient.
- With aspect-testing physically distributed among the aspects themselves, it may be difficult for a developer to perform isolated testing independent of the aspects. If all of the tests were registered in a central location, this task would be trivial. Luckily, the `jUnit` framework provides the ability to run one test out of a suite using its graphical interface. Additionally, the `jUnit` framework encourages the creation of short tests so that testing can occur frequently during the development process. From the experience in developing the event channel, the downside of running unnecessary tests has been minimal as even those tests that send events among many suppliers and consumers complete quickly.

The second issue with testing a customizable event channel is how to write tests in such a way that they can be run under any compatible configuration. For example, those tests that verify proper routing of basic event-channel messages should not be aware of the `TTL` feature. The `TTL` feature is completely irrelevant to the routing tests in that there are many configurations that should support proper event dispatch without use of the `TTL` field. However, when the `TTL` feature is enabled, the `TTL` field of every event message is checked, and must be set by an application or unit test before sending the message. To support tests that lack knowledge of the `TTL` feature when it is enabled, another aspect is used to set the `TTL` feature to a default value in other unit tests that stress other features.

A more complicated example is used for the event dependence feature. For this case, consumers must register a list of event types that they would like to receive. In other words, the processing of these consumers depends on the event generation characteristics of their suppliers. A real-time scheduler can then perform feasibility analysis for the event channel and warn when cycles exist in the dependence graph. The solution of advising other unit tests to register their consumers as dependent on all suppliers could fail if a consumer is a supplier. The current approach to this situation is to add a flag to indicate a consumer that has not been constructed with these constraints in mind and to set that flag when advising the other unit tests.

7 Future Work and Concluding Remarks

We are continuing work on the event channel to provide it with many of the features of the TAO RTEC and to continue our research into constructing customizable middleware using AOP. We will also be working to develop a more robust AspectIDL preprocessor as the use of aspects in Java will continue to affect the exported CORBA event channel interfaces. In addition, we are investigating methods for refactoring the base code to move the CORBA elements into separate aspects.

We are in the process of developing a highly customizable real-time event channel in AspectJ. Our approach has been to identify and extract features as separate aspects of a small, sleek, core event channel implementation. From our work so far, we have investigated and identified several techniques to support and manage large numbers of selectable aspects. These techniques include designing an API that includes parameter blocks, using an AspectRegistry to verify use of consistent aspects, and developing procedures for unit-tests to allow them to be run under as many configurations as possible.

Acknowledgements

We thank Morgan Deters for help with aspects and AspectJ. We thank Michael Plezbert for help with patterns.

References

- [1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.
- [2] J. P. Morgenthal, "Microsoft COM+ Will Challenge Application Server Market." www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.
- [3] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.
- [4] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/~schmidt/TAO.html, Washington University.
- [5] F. Kuhns, C. O’Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Proceedings of the IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.
- [6] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.
- [7] The AspectJ Organization, "Aspect-Oriented Programming for Java." www.aspectj.org, 2001.
- [8] J. Gosling and K. Arnold, *The Java Programming Language*. Reading, MA: Addison-Wesley, 1996.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [10] OMG, *CORBAServices: Common Object Services Specification, Revised Edition*. Object Management Group, 97-12-02 ed., Nov. 1997.
- [11] Object Management Group, *Notification Service Specification*, OMG Document telecom/99-07-01 ed., July 1999.
- [12] P. Gore, R. K. Cytron, D. C. Schmidt, and C. O’Ryan, "Designing and Optimizing a Scalable CORBA Notification Service," in *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, (Snowbird, Utah), ACM SIGPLAN, June 2001.
- [13] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, October 1997.
- [14] I. Pyarali, C. O’Ryan, and D. C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.
- [15] Erich Gamma and Kent Beck, "JUnit." www.xProgramming.com/software.htm, 1999.