Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2002-7

2002-04-10

Building Customizable Middleware using Aspect-Oriented Programming - Master's Thesis, May 2002

Frank Hunleth

In order to support a wide range of applications, Distributed Object Computing (DOC) middleware frameworks such as ACE and TAO have grown to include a vast number of features. For any one application, though, unused functionality either contributes to code bloat, degrades performance or both. When applied to embedded and realtime systems, these issues can preclude the use of middleware altogether. Currently, to address these concerns, middleware developers continually refactor code to relegate functionality to separate libraries. This process is tedious, time-consuming, and adds complexity for both users and developers. To address the difficulties of creating subsettable middleware, we... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Hunleth, Frank, "Building Customizable Middleware using Aspect-Oriented Programming - Master's Thesis, May 2002" Report Number: WUCSE-2002-7 (2002). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1164

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1164

Building Customizable Middleware using Aspect-Oriented Programming - Master's Thesis, May 2002

Frank Hunleth

Complete Abstract:

In order to support a wide range of applications, Distributed Object Computing (DOC) middleware frameworks such as ACE and TAO have grown to include a vast number of features. For any one application, though, unused functionality either contributes to code bloat, degrades performance or both. When applied to embedded and realtime systems, these issues can preclude the use of middleware altogether. Currently, to address these concerns, middleware developers continually refactor code to relegate functionality to separate libraries. This process is tedious, time-consuming, and adds complexity for both users and developers. To address the difficulties of creating subsettable middleware, we have developed a novel method for constructing middleware using Aspect-Oriented Programming (AOP) and applied it to develop a realtime CORBA Event Channel called the Framework for Aspect Composition of an EvenT channel (FACET). FACET consists of a small, essential core that represents the basic structure and functionality of any event channel. By using aspects, additional features are woven not the core so that the resulting event channel supports all of the features needed by a given embedded application.

Washington University in St. Louis

School of Engineering & Applied Science

Short Title: Customizable Middleware using AOP Hunleth, M.Sc. 2002

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

BUILDING CUSTOMIZABLE MIDDLEWARE USING ASPECT-ORIENTED PROGRAMMING

by

Frank Hunleth

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

BUILDING CUSTOMIZABLE MIDDLEWARE USING ASPECT-ORIENTED PROGRAMMING

by Frank Hunleth

ADVISOR: Dr. Ron K. Cytron

May, 2002
Saint Louis, Missouri

In order to support a wide range of applications, Distributed Object Computing (DOC) middleware frameworks such as ACE and TAO have grown to include a vast number of features. For any one application, though, unused functionality either contributes to code bloat, degrades performance or both. When applied to embedded and realtime systems, these issues can preclude the use of middleware altogether. Currently, to address these concerns, middleware developers continually refactor code to relegate functionality to separate libraries. This process is tedious, time-consuming, and adds complexity for both users and developers.

To address the difficulties of creating subsettable middleware, we have developed a novel method for constructing middleware using Aspect-Oriented Programming (AOP) and applied it to develop a realtime CORBA Event Channel called the Framework for Aspect Composition for an EvenT channel (FACET). FACET consists

of a small, essential core that represents the basic structure and functionality of any event channel. By using aspects, additional features are woven into the core so that the resulting event channel supports all of the features needed by a given embedded application.

A feature-management framework was developed to cover all supported features and validate their combinations. To ensure correct operation, every feature has a corresponding set of unit tests. Since arbitrary compositions of features may lead to unforseen behaviors, the FACET test framework can enumerate and test all valid feature combinations of the middleware. This provides a high degree of confidence in the event channel in any environment.

Additionally, we present quantitative results on the impact of features on the footprint and performance of FACET. Several typical configurations are also identified and compared to show their significant advantages over fixed feature set event channels. Finally, several key design patterns for the development of middleware using AOP are presented.

To Christy

Contents

List of Figures . . . . **vii**

Acknowledgments . . . . **ix**

1 Introduction **1**

2 Background **5**

2.1 Middleware 5

2.2 Current Practices to Subset Middleware 6

2.2.1 Subsetting Techniques 7

2.2.2 Case Study: Subsetting TAO 10

2.3 Advanced Separation of Concerns 11

2.3.1 AOP and AspectJ 12

2.3.2 Multi-Dimensional Separation of Concerns 13

2.3.3 Composition Filters 14

2.4 Compositional Middleware 15

2.5 Event Channels 15

2.5.1 OMG Event Service 17

2.5.2 OMG Notification Service 18

2.5.3 TAO Real-time Event Channel 19

2.5.4 Feature Summary 19

3 FACET Architecture **21**

3.1 High Level Overview 21

3.2 Defining the Base 24

3.3 Features 25

3.4 Adding New Features to FACET 29

4 Feature Management **31**
4.1 Feature Registry 31
4.1.1 Types of Features 32
4.1.2 Relationships between Features 33
4.1.3 Feature Cycles 34
4.1.4 Feature Registration 34
4.1.5 FACET Feature Dependence Graph 35
4.1.6 Combining Features 36
4.2 Build Environment 37
4.2.1 Feature Organization 38
4.2.2 Feature Selection 39
4.3 Aspect Support for Multi-Languages Environments 39

5 Testing **41**
5.1 Test Framework 41
5.1.1 Running the Appropriate Subset of Tests 42
5.1.2 Automatically Upgrading Tests 44
5.2 Verifying All Combinations 48
5.3 Common Mistakes when Writing Feature Unit-Tests 49
5.4 Genericity of the Testing Framework 50

6 Aspect Oriented Design Patterns **51**
6.1 Encapsulated Parameter Pattern 52
6.2 Template Advice Pattern 59
6.3 Interface Tag Pattern 64

7 Experimental Results **68**
7.1 Footprint 68
7.1.1 Quantifying the Footprint Increase of Individual Features . . . 69
7.1.2 Footprint Sizes for Common Configurations 71
7.1.3 Impact of External Libraries 75
7.2 Performance 77
7.2.1 Performance Effect of Enabling Features 79
7.2.2 Performance of Common Configurations 79
7.3 Savings from Using Aspects 82

8 Conclusions and Future Work 85

Appendix A Glossary 89

Appendix B FACET Features 90

B.1 Basic Counters 91
B.2 Body Any 91
B.3 Body Octet Seq 91
B.4 Body String 92
B.5 Consumer Filtering 92
B.6 Consumer Qos 92
B.7 Context Free Filter 93
B.8 CORBA Oneway 93
B.9 Correlation Filter 94
B.10 Depend 94
B.11 Event Channel Tracing 94
B.12 Event CORBA Any 95
B.13 Event Header 95
B.14 Event Pull 95
B.15 Event Sets 96
B.16 Event Struct 96
B.17 Event Type 96
B.18 Event Type Filter 97
B.19 Event Type Mutex 97
B.20 Profiling Support 97
B.21 Supplier Dispatch 98
B.22 Timestamp 98
B.23 Time-To-Live 98

References 100

Vita 105

List of Figures

2.1 Subsetting code fragments from multiple classes to a separate feature. 8
2.2 Resulting structure of code after subsetting. 9
2.3 Main participants in an event channel. 16
2.4 High level event channel feature summary. 20

3.1 The main components in FACET. 22
3.2 Components of FACET features. 26

4.1 A feature interface and registration aspect. 35
4.2 Feature Dependence Graph: Oval nodes are concrete features, diamond nodes are abstract features, and rectangular nodes are mutual exclusion features. 35
4.3 An example configuration file. 39

5.1 TestSuiteAdder aspect. 43
5.2 Typical use of the TestSuiteAdder. 43
5.3 The feature interface for the event type feature. 46
5.4 The upgrader aspect for the TTL feature. 46
5.5 Encapsulating the upgradable concern within an aspect. 47
5.6 Upgrading an upgrader. 47
5.7 IDL generated convenience constructor. 50

6.1 Encapsulated Parameter pattern structure. 55
6.2 Template Advice pattern structure. 60
6.3 AutoRegisterAspect abstract aspect. 62
6.4 RegisterTtlFeature registration implementation. 62
6.5 Interface Tag pattern structure. 65

7.1 Feature set sizes. 70

7.2 Feature sets. 70
7.3 Class file measurements for FACET feature sets. 72
7.4 GCJ object file measurements for FACET feature sets. 73
7.5 Enabled features under various configurations. 75
7.6 FACET Library sizes under different configurations. 76
7.7 Increase in external library under various configurations. 78
7.8 Feature impact on throughput. 80
7.9 Throughput degradation measurements for FACET feature sets. . . . 81
7.10 Throughput results normalized to the base throughput. 82
7.11 Measured throughput of common configurations. 83
7.12 Overhead of using *if* statements for tracing rather than aspects. . . . 84

B.1 Feature Dependence Graph. 90

Acknowledgments

First, I thank my advisor, Ron K. Cytron, for all of his help with this work from introducing me to Aspect-Oriented Programming to coauthoring two papers with me and for giving me quite a few ideas on how to improve FACET. I also thank Chris Gill for teaching me quite a bit about distributed and realtime middleware and coauthoring the original FACET paper.

I would like to thank Doug Schmidt for creating ACE and TAO, and for making them so popular that subsetting middleware would be an important research issue. He also provided much encouragement for my work on ACE and TAO over these two years, in which I learned more about distributed middleware than I had ever expected. Likewise, I thank the rest of the DOC Group at Washington University and the University of California Irvine for teaching me about the inner workings of TAO and for many enjoyable programming experiences. These include Kitty Balasubramanian, Sharath Cholleti, Irfan Pyarali, Pradeep Gore, Nanbor Wang, Venkita Subramonian, Jeff Parsons, Yamuna Krishnamurthy, Ossama Othman, Carlos O'Ryan, Martin Linenweber, Luther Baker, Michael Plezbert, Matt Hampton, Steven Donahue, and Ravi Pratap. In particular, I thank Angelo Corsaro who was my XP programming partner when we subsetted RTCORBA out of TAO, Bala Natarajan who helped fix some of my bugs in TAO when I was too busy working on this thesis, and Morgan Deters who answered numerous AspectJ programming questions.

I would like to thank Mike Henrichs for laying out the feature dependence graph, writing a poll for the TAO users group, and for carefully proofreading previous work that was included in this thesis. I would also like to thank Joachim Achtzehnter, Lothar Werzinger, Tommy Carlsson, Oliver Kellogg and Brian Mendel for their comments on the event service feature configurations used in their applications.

Finally, I thank DARPA for supporting my research under contract F33615–00–C–1697.

Frank Hunleth

Washington University in Saint Louis
May 2002

Chapter 1

Introduction

Traditionally Successful Distributed Object Computing (DOC) middleware, such as the Common Object Request Broker Architecture (CORBA) [35], COM+ [33], and Java Remote Method Invocation (RMI) [50], provides a rich feature-set to increase its applicability across diverse problem domains. Not surprisingly, any particular application tends to use only a limited subset of features. This observation, coupled with the practical reality that computation and memory resources are limited, leads software architects and designers to include frameworks for customizing a feature set for a particular application. This thesis describes new techniques for building customizable middleware using Aspect-Oriented Programming (AOP) and describes the design and performance of the Framework for Aspect Composition for an EvenT channel (FACET), an event notification service built using AOP.

An increasingly important area for DOC middleware is embedded and real-time systems. In general, these systems have stricter requirements for predictability and often impose harsher limitations on the available resources for computation and storage. To support these environments, middleware such as the ADAPTIVE Communication Environment (ACE) [44] and The ACE Object Request Broker (ORB) (TAO) [14] have both been designed with customizability in mind and have been subsetted extensively. However, current techniques for subsetting middleware such as ACE and TAO have numerous shortcomings:

1. Standard subsetting techniques such as the use of macros to include code selectively, or the use of design patterns [22] such as Strategy or Template Method

require *a priori* knowledge of customization points. [1] If subsetting is not considered in an application's design up front, the code around the customization point must be refactored [21] to be amenable to applying these design patterns. By using AOP, new functionality can be added to the core application after it has been written, without any refactoring. In other words, feature customization points need not be known ahead of time, and customization can be accomplished subsequently without refactoring.

2. As a result of adding macros and strategizing customization points, the basic functionality of core code can be obfuscated by the customization infrastructure. This complicates program maintenance and evolution. As demonstrated in FACET, this problem can be eliminated by AOP, and implementing new extensions can be further simplified in several ways.

3. The Strategy and Template Method patterns introduce code that is present at runtime, even for those features *excluded* from a particular runtime configuration. At best, this introduces an insignificant amount of runtime overhead—for example, to check a strategy—and this overhead may be acceptable. However, if a customization point is in a performance-critical loop, or if the call is made to another software context (such as across shared libraries), then the additional method-call can impact system performance and predictability.

Chapter 2 provides further background on developing customizable middleware and the requirements of embedded systems. Additionally, observations and experience from subsetting significant components of TAO are described.

Then, Chapter 3 describes the high level architecture of FACET and the interactions between the major components. This framework for using AOP to build customizable middleware is general enough that it can be applied to other types of middleware as well.

Another often-overlooked issue when subsetting middleware features is to provide a mechanism to identify dependences between features and to signal errors when two mutually exclusive features are selected. The most common technique for solving this problem is to use conditional compilation macros to check for all possible violations. This method is error prone, due to the amount of manual work involved. Additionally, as shown in FACET, AOP allows one to develop many fine-grain features

[1] An overview of design patterns can be found in Chapter 6.

making the feature-management problem more severe. To solve this problem, this thesis presents a feature-management framework that automatically validates feature configurations and simplifies management of the features' dependences. Chapter 4 describes this framework in detail.

Next, reliability is always a concern when developing software, especially for embedded or realtime systems that may be located in remote locations or perform safety-critical tasks. An important tool to ensure software quality is the creation and automated execution of unit tests. Additionally, to ensure the quality of customizable middleware, not only should every feature be validated, but also all meaningful *combinations* of features should be checked to identify unintentional interference between features. A naive approach of exhaustive feature enumeration is intractable, since the number of (valid and invalid) feature combinations grows exponentially.

However, by using the feature-management framework in FACET, it is possible identify only those feature combinations that produce a *viable* configuration. This thesis provides empirical evidence that it is feasible to test all viable combinations of feature in FACET. By reducing thorough testing to an automatic, relatively efficent process, it is likely that software developers will perform testing routinely and frequently, thus shortening development time and increasing the reliability of the delivered middleware.

An additional issue that arises when enabling testing over all combinations of features is when one feature changes the expected behavior of another feature's unit tests. For example, FACET has a feature that allows for the specification of the maximum number of event channels that an event can pass through before being dropped. Tests that are written without knowledge of this feature, do not initialize it, and therefore have their events dropped when it is enabled. This issue actually arises frequently, typically because an enbabled feature requres that some additional work be performed at initialization, or before uses of some core functionality. Traditionally, testing techniques for middleware overlook testing in this area due to the added complexity for allowing for these cases. The complexity of feature interaction is often sufficiently daunting that developers tend to bundle sets of interactive features without testing their interactions. However, by taking advantage of AOP techniques, FACET can automatically update unit tests to handle modifications to core functionality by other unrelated features. Chapter 5 describes both the test framework behind FACET, and this use of AOP to update unit tests.

During the development and refinement of FACET, several new patterns were identified, involving the use of AOP to develop customizable middleware. Since aspect-oriented software development is relatively new, these patterns will likely be useful to many projects that use AOP. Chapter 6 describes these design patterns, the mistakes that were made before using them, and their use throughout the FACET implementation.

Many performance and footprint improvements were achieved by having the ability to configure the FACET event channel to the exact desired set of features. Chapter 7 provides measurements of FACET under various configurations and analyzes the impact that individual features have on the middleware.

Finally, Chapter 8 summarizes our work on FACET and describes future work applying aspects to flexible middleware components and services.

Chapter 2

Background

This chapter provides background information on the development and evolution of Distributed Object Computing (DOC) middleware. Based on current practices and experiences from the development of this middleware in the DOC Group, the problems associated with evolving and maintaining the software are presented. New programming techniques such as Advanced Separation of Concerns (ASoC) and, in particular, aspects [27] are then described as a mechanism for addressing these shortcomings. Finally, this chapter provides an overview of the specific type of middleware, Event Services, that were studied to develop the Framework for Aspect Composition for an EvenT channel (FACET).

2.1 Middleware

Developing large software projects is notoriously difficult [9]. Programming platforms vary widely, outdated and unwieldy programming interfaces abound, and frameworks for addressing communication issues either may not be available or may not be interoperable. It is for these types of problems where middleware has proven to be very useful in practice [45].

DOC middleware is a specific category of middleware that addresses the many accidental and inherent complexities [10] of network and distributed programming. Accidental complexity refers to the programming issues with using tools, languages, interfaces, and frameworks that are difficult to use and prone to errors. Network programming has historically been difficult due to the lack of availability of anything besides low level socket interfaces. On the other hand, inherent complexities arise out of inherent difficulties with developing any program in the domain regardless

of language, tools, or libraries. For networking, these include issues such as fault tolerance, security, concurrency, and program distribution.

the ADAPTIVE Communication Environment (ACE) [44] and The ACE Object Request Broker (ORB) (TAO) [14] are two of many examples of DOC middleware frameworks that address the difficulties of distributed network programming. Both of these frameworks have matured over many years of use for both research and industry applications [24, 18]. Issues identified during their development and evolution, though, have led to current research and this work.

The development of ACE and TAO parallels many of the issues in the development of large software systems. In both cases, the first releases may not have all of the necessary features, but the overall design was elegant and areas for future extension had infrastructure to support that extension. As time continued, more and more features were piled into the original code. Some of the original code was also refactored to allow for extensibility where there was none before.

This evolution continues, but middleware also has to deal with forces not present in the context of single applications. These include being used in varying applications, platforms, and environments where each environment may have different functionality requirements. Additionally, as middleware becomes more popular, it begins to be put into environments that were not even conceived of early in development. In the case of ACE and TAO, both frameworks are being used in more resource constrained environments than initially expected. At the same time, though, their popularity has encouraged the introduction of more and more features that take up more memory and processing. This has led to a cycle of adding functionality, followed by further subsetting to maintain and shrink the overall resource usage. This process is time consuming and not ideal for the evolution of complex middleware. Many times, the end result may not even be small or fast enough. The following section describes the practices and issues that have arisen during this process.

2.2 Current Practices to Subset Middleware

The need to subset (or extend) middleware selectively has existed for some time [21], and many design patterns have been identified that document successful strategies. These include patterns such as Strategy [22], Interceptors, Extension Interface, Component Configurator and others [46]. Although such patterns have been used extensively in middleware such as ACE and TAO, the patterns carry several disadvantages:

1. They require additional infrastructure within the framework to support their presence. For example, the Strategy and Interceptor patterns require method-invocation hooks to be placed at key locations throughout the code. From a programmer standpoint, these hooks and the additional infrastructure lessen the readability and maintainability of the code.

2. If the locations where subsetting should have occurred are not preconceived, time-consuming refactoring may be needed to partition functionality into separate libraries.

3. The hooks and infrastructure themselves can lead to degraded performance and increased footprint size.

2.2.1 Subsetting Techniques

Currently, subsetting a middleware feature generally involves the following steps:

1. Transforming the code to decouple feature-specific logic, classes, and data from the core library.

2. Informing the base or core implementation that the feature is present when it has been included.

3. Fitting the feature into the base implementation's loading and configuration services, if any.

The first item is the most time consuming and for the most part, it involves many simple code transformations to ease the extraction of optional features to external libraries. A byproduct of these transformations is that the core code becomes more flexible and extensible. Well-known techniques such as those documented in the refactoring literature [21] address these issues and are heavily used in the subsetting process. A common example of such refactoring is to decompose a method into multiple, smaller methods so that it is possible to apply the Strategy or Template Method patterns. Code specific to a feature can then be removed from the core class to a separate library. Once in the separate library, that feature's code can be reregistered with the core middleware using the techniques associated with the Strategy or Template Method patterns. A limitation of this refactoring is that additional registration infrastructure is needed in the core middleware for each extended class.

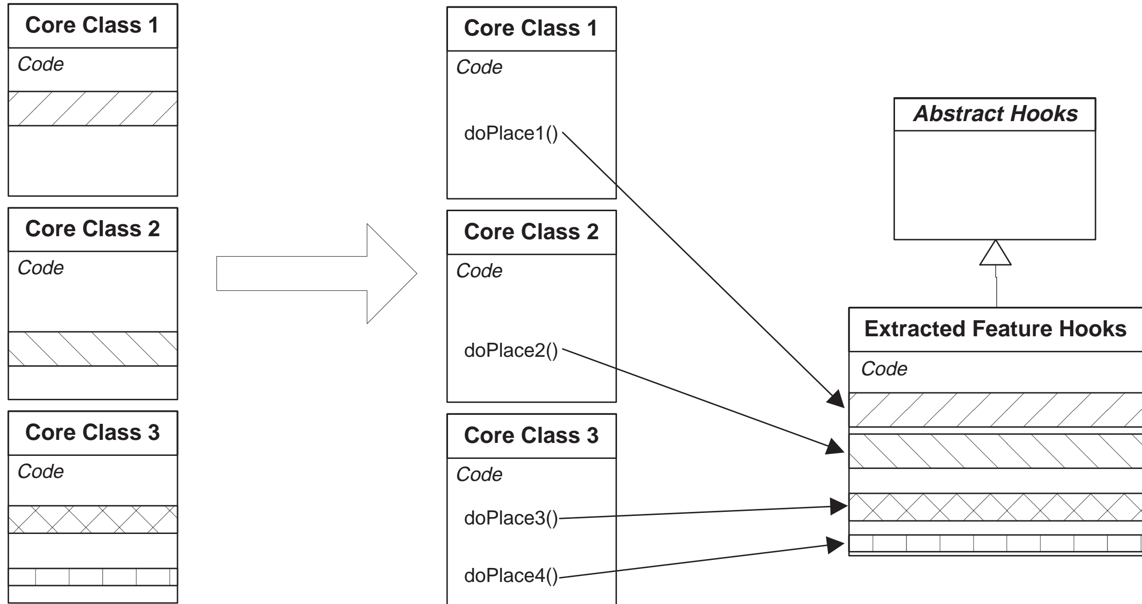

Figure 2.1: Subsetting code fragments from multiple classes to a separate feature.

A variation on this style of refactoring—also in common use—is to combine feature-specific code from multiple core classes into a common *hooks* class. A Singleton [22] is then used throughout the core to access the currently operating hooks class. When a feature is configured, it registers its implementation of the hooks class with the singleton. Figure 2.1 depicts this subsetting operation: on the left is the original code that has pieces of the feature spread across several classes. The subsetting operation involves creating an interface, Abstract Hooks that has methods for each section of code that will be removed from the core. A concrete implementation of this interface is then created that contains all of the feature specific code. When the middleware is executed, the core makes invocations into this concrete implementation, so that the feature can provide its functionality. Another way to describe the concrete implementation, Extracted Feature Hooks, is to say that it encapsulates many of the crosscutting concerns of the feature.

Much of the infrastructure needed to support subsetting a piece of middleware can be shown using this same example. Figure 2.2 is a UML diagram of the static structure of this infrastructure and the feature extension code. On the left is the Component Configurator design pattern [46] which provides the infrastructure to load and configure a feature statically or dynamically. This consists of the Component

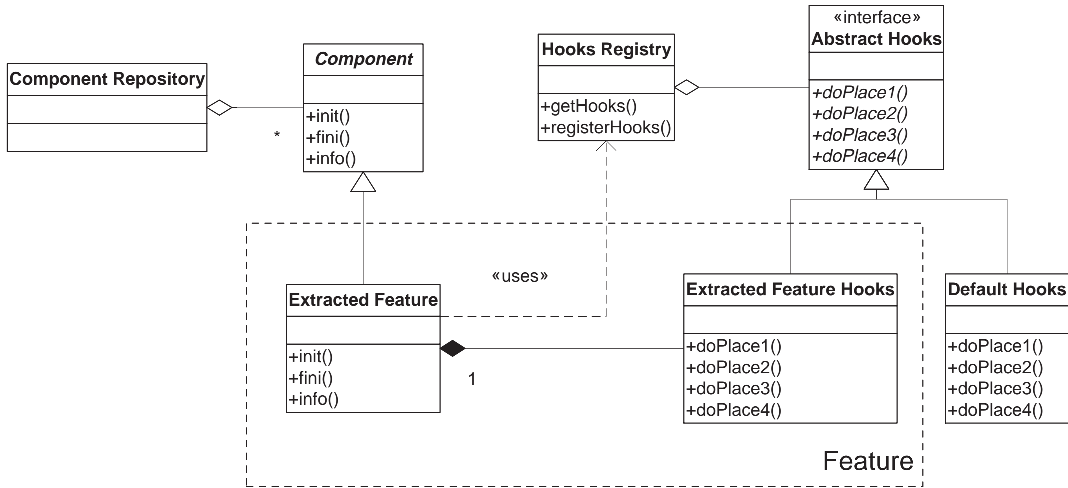

Figure 2.2: Resulting structure of code after subsetting.

Repository that keeps track of the components or features that have been loaded, a Component that defines a common interface to extensions, the Extracted Feature that contains the logic to initialize and register the subsetted feature with the core. The Abstract Hooks and Extracted Feature Hooks fill the same roles as described before. A Default Hooks is usually created to provide some default (possibly null) behavior, so that the core can work when the subsetted feature is disabled. Finally, a Hooks Registry is needed so that the core code can retrieve the current *hooks* code and so that the subsetted feature can register its own hooks.

Figure 2.2 also reflects many of the limitations of current subsetting techniques, because so much infrastructure is required in the core library. Of the classes in the diagram, only Extracted Feature and Extracted Feature Hooks are part of the subsetted feature's library code. Although this does not have to be the case, it is often true that the Abstract Hooks and Default Hooks classes are specific to one feature. Thus, the core middleware must have these classes for each subsetted feature. This is clearly not ideal from a theoretical or a practical standpoint, since feature code (albeit less) is still present in the core, and significant infrastructure needs to be created and maintained once the feature has been extracted.

2.2.2 Case Study: Subsetting TAO

TAO is a full-featured the Common Object Request Broker Architecture (CORBA) ORB developed by the DOC Group at Washington University and University of California at Irvine. Although much of it was designed to be configurable from the beginning, it has nonetheless grown to the point that its footprint size has become too large for many embedded systems (as well as some desktop systems). As a result, functionality has been subsetted from its core many times. In fact, this is an ongoing effort, as TAO is increasingly considered for use in environments with tighter memory constraints.

One feature recently subsetted from TAO is support for the Real-Time CORBA 1.0 (RTCORBA) specification [35]. RTCORBA defines standard mechanisms that allow applications to control the priorities at which CORBA requests are processed and how threads are allocated internally in an ORB. Many applications do not need the features of RTCORBA, and developers of such applications find that the overhead in footprint and processing of those features is burdensome.

The time and effort required to subset RTCORBA features from TAO was significant: it took two expert ORB developers nearly five months' time to refactor code throughout the ORB and its associated libraries, and to verify its operation on all supported platforms. This work consisted of the following tasks:

1. Writing service code to support dynamic loading and initialization of the library,

2. Moving those files and classes having to do with RTCORBA and already decoupled from the core code to separate directories,

3. Creating an RTCORBA-specific *hooks* class that can be used to register callbacks from the core ORB to the RTCORBA library,

4. Identifying RTCORBA code-fragments that can be refactored into calls to the *hooks* class,

5. Restructuring code, using the Strategy pattern, to support code that has completely different behavior with RTCORBA enabled,

6. Refactoring switch statements that have RTCORBA specific cases into registries where appropriate, and

7. Removing RTCORBA fields from core data structures and provide an extension mechanism to the data structures to attach the RTCORBA specific data at runtime.

After RTCORBA was removed, the size of the core library was reduced by about 10%, and many method calls were removed from the critical path of the ORB resulting in a small but noticable performance improvement.

A benefit of this process is that the core TAO code became more extendable for future features. However, in addition to having to go through the tedious subsetting process, the resulting TAO code is now:

1. More complicated due to the additional strategy classes and RTCORBA interception points,

2. Not as fast as possible due to the overhead of maintaining the hooks for RTCORBA even when RTCORBA is not being used,

3. Suffering from additional overhead from new calls between the core code and the RTCORBA library when RTCORBA is in use.

In this thesis we describe how Aspect-Oriented Programming (AOP) techniques can alleviate the problems encountered with subsetting TAO.

2.3 Advanced Separation of Concerns

Separation of concerns [17] is the general term given to the process of identifying and encapsulating related ideas and concepts together. Separation of concerns for Object-Oriented Programming (OOP) involves identifying the structure of classes and interfaces that define an application. However, separating concerns based on structural elements is only one of many dimension where separation can occur. The inability of OOP to separate other concerns has led to significant research in identifying new approaches [19, 16, 27, 42, 15]. These approaches are collectively termed Advanced Separation of Concerns (ASoC) due to their ability to enable more flexible separations. A premise of this thesis is that the difficult subsetting practices described in Section 2.2.1 occur when concerns are not properly separated. By using languages and tools that possess ASoC expressiveness, composable middleware can be constructed more readily.

Before describing the languages and paradigms used to encapsulate nonstructural concerns, it is useful to describe other types (or dimensions) of concerns. These can be broadly categorized as *systemic* and *functional* concerns [41].

- Systemic concerns include synchronization, realtime, scheduling, transaction semantics, caching and prefetching strategies and memory management concerns.
- Functional concerns comprise application logic and features. These differ from systemic concerns in their scope and intention. For example, a application logic such as a new business rule may effect several computations and decisions in separate classes, but a systemic concern such as synchronization affects many classes systemwide.

Both of these types of concerns crosscut many classes, and by encapsulating them into separately compilable units, one can selectively enable or disable their behavior. A key observation is that software requirement tracability is much more apparent for languages that support separation of nonstructural concerns [42]. The following sections provide an overview of the types of languages that are useful to distill functional and systemic concerns from middleware.

2.3.1 AOP and AspectJ

AOP [27] is a software development paradigm that enables one to separate concerns that crosscut sets of classes and encapsulate those concerns in self-contained modules called *aspects*. The AspectJ [48] programming language adds AOP constructs to Java [4] and uses the following terminology. Within an aspect, the locations at which *advice* should be applied are defined using *pointcuts*. Each pointcut is made up of one or more *joinpoints*, which are well-defined locations in the execution of a program. The code applied at a pointcut is called *advice*. In addition to applying advice, languages supporting AOP often allow new methods or other language features to be *introduced* into existing classes. Of all of the separation of concerns languages suitable for developing middleware, AspectJ is currently the most mature and was thus selected for the experiments documented in this thesis.

As described in Section 2.2.1, reducing the coupling between classes in a library can reduce the footprint of applications that use selected parts of that library. AOP provides a novel mechanism to reduce footprint size even further by enabling

crosscutting concerns between modules to be encapsulated into user-selectable aspects. Following chapters will describe how we can use AOP to identify the core functionality of a middleware framework and then to codify all additional functionality into separate aspects. The advantage of using AOP is that the hooks and callbacks required for subsetting (using standard, object-oriented techniques) are no longer required. This removes the need to preconceive where points of variation are needed in the code and also removes the need to refactor large amounts of existing code to insert these hooks after the fact. The patterns described in Chapter 6 make achieving these advantages in AspectJ easier.

Desirable combinations of these aspects are then selected by middleware users so to include the minimum functionality needed to support a given application. By performing a fine-grain decomposition of the functionality, a middleware framework could add very little bloat to an application, and thereby free the embedded developer from concerns about excessive overhead. Unfortunately, fine-grain decompositions can also add complexity for both the middleware user and developer. We address this issue in Chapter 4 by providing a framework to manage features, their relationships, and by integrating this knowledge into the build environment.

Previous work in subsetting applications using AOP has been done for the GNU sort utility [12]. The authors identified 60 fine-grain separate concerns and decomposed many of them using AOP programming techniques. That work has many similarities with the initial event channel decomposition efforts in this thesis.

2.3.2 Multi-Dimensional Separation of Concerns

Multi-Dimensional Separation of Concerns (MDSOC) using Hyperspaces [40] provides another mechanism for encapsuling crosscutting functionality. In the MDSOC model, all possible concerns are *located* at points throughout hyperspace. For example, when writing an application, a programmer chooses a particular way (out of many possible ways) to separate concerns. Likewise, hyperspace can be *sliced* in many ways to form units called *hyperslices*. Additionally, since concerns can be separated using different points of view, hyperspace is considered multi-dimensional. In order to create a program using MDSOC, many hyperslices are needed. These are composed together to form a *hypermodule*. An important contribution of this model is that hyperslices need not be orthogonal to be combined in a hypermodule. When overlap occurs (for example, two hyperslices augment a class with different methods that have the

same signature), the hypermodule can specify the resolution. The resolution can be something as simple as running all overlapping methods and sending all results to a summarization function. AOP does not provide this capability, and therefore some ingenuity is required to address non-orthogonal aspects.

When constructing middleware, hyperslices can be used in a similar way as aspects in AOP. That is, optional functionality can be relegated to independently selectable hyperslices. The main disadvantage of using hyperslices is that knowledge of how to compose hyperslices is specified in the description of the hypermodule. This benefits normal applications that reuse hyperslices, since their incorporation into the final application can be precisely specified. For middleware, though, it is the user who chooses the desired functionality. Since the number of hyperslices may be large, it is impractical to create a hypermodule for every possible configuration. Likewise, relegating hypermodule creation to the user adds significant complexity. Techniques to automate this process may make it more practical.

2.3.3 Composition Filters

Composition Filters [5, 1] provide yet another model through which crosscutting concerns can be encapsulated into separate software modules. Here, all method invocations are treated as if they were passing messages. Filters can then be set up to monitor and modify messages depending on such things as the sender of the message, the recipient, or the contents of the message. Additionally, filters can be attached to classes as enhancements.

Developing composable middleware using this model can be achieved in a similar manner as with AOP. In both cases, a base implementation is developed that has the essential functionality and structure of the middleware framework, and then *filters* are enabled to supply additional functionality. Unlike hyperslices, filters must be orthogonal to each other, so special care is required for some feature combinations.

Although composition filters were not used in this work, other work [6] has investigated their use to construct middleware. This thesis is different in that functionality is separated in a more fine-grained manner, and aspect management and testing are given much higher priority than solely encapsulating crosscutting features.

2.4 Compositional Middleware

Previous research in compositional middleware has also addressed building configurable middleware. One of these approaches was used in the Coyote system [7] and the more recent Cactus system [32]. Both systems address network services and protocol middleware by providing fine grain decompositions of their functionality that can later be composed. The way that they accomplish this, though, is very different from that provided in this thesis. In the Cactus system, functional components are separated into modules called *micro-protocols*. Each micro-protocol contains a set of event handlers that are bound to events in the Cactus runtime. By composing these micro-protocols in various ways, the required functionality can be attained. To contrast, the work presented here uses AOP techniques to compose features with the base implementation. This frees the base from needing to preconceive all of the possible interception points ahead of time, and provides more flexibility for the feature writer.

Another approach to decomposing middleware functionality into features and allowing arbitrary combinations to be enabled is in Feature-Oriented Programming (FOP) [43]. This approach adds language support to make *features* first class entities, and has many similarities to the Separation of Concerns languages mentioned previously. The difference is that it has been designed specifically for composing features in middleware. In FOP, a feature is like an abstract subclass, but it also has the ability to override methods in other classes in addition to the parent. For the most part, though, FOP has not been used to build full systems like has been done for this thesis. Moreover, AOP appears to provide a much more generic and powerful mechanism for attaching features to core middleware.

2.5 Event Channels

FACET is an implementation of a CORBA [35] *event channel* that uses AOP to achieve a high level of customizability. Its functionality is based on features found in the Object Management Group (OMG) Event Service [38], the OMG Notification Service [34, 23], and the TAO Real-time Event Service [39] [25].

At a high level, an event channel is a common middleware framework that decouples event suppliers and consumers. The event channel acts as a *mediator* through

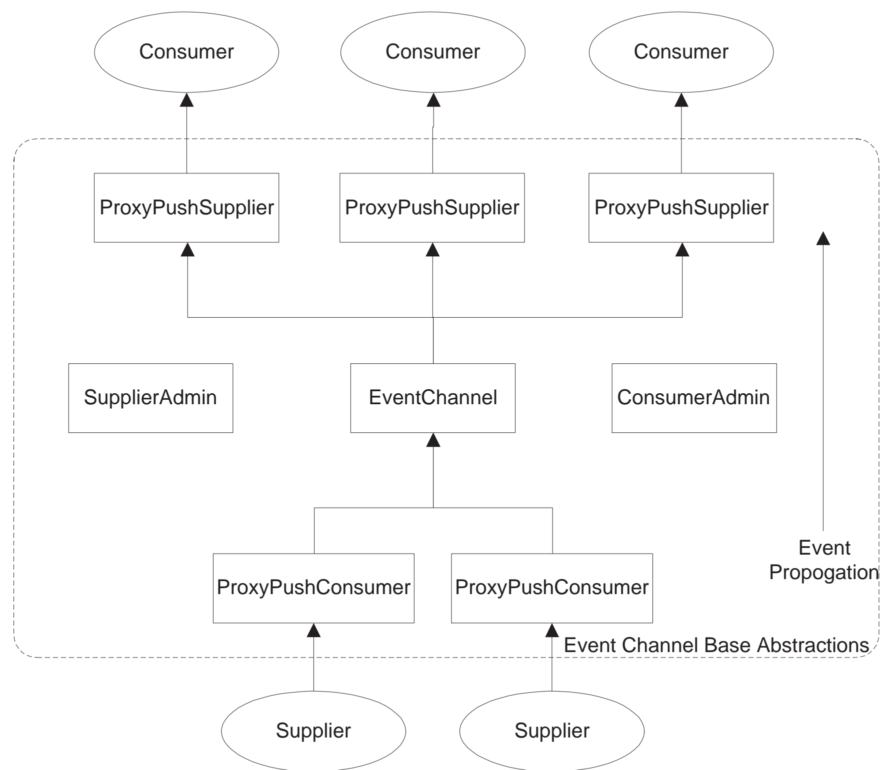

Figure 2.3: Main participants in an event channel.

which all events are transported. Figure 2.3 shows the main participants in the framework. In the simplest case, suppliers push events to the event channel, and then the event channel pushes those events to consumers. Event channel implementations differ in the types of events that they handle and in the processing and forwarding that occurs within the channel.

The following sections describe various CORBA event services and their features.

2.5.1 OMG Event Service

The OMG Event Service is the simplest standardized CORBA event service. It provides the basic interfaces for consumers and suppliers to subscribe to an event channel and then to receive and send events. With the exception of event services that support typed events[1], this event service only transports CORBA Any data types. A CORBA Any is a self-describing data type that can hold any CORBA Interface Definition Language (IDL) describable data (e.g., basic types, structures, unions, and arrays.) The contents of the events are hidden from the event channel, and as such, the event channel can perform no specialized treatment for events. It merely forwards all of the events to all of the consumers that are registered. To limit the propogation of events to uninterested consumers, applications typically instantiate multiple event channels for each event category. Additionally, event delivery is purely on a best effort basis.

To support the various event transport models, the OMG Event Service supports both *push*- and *pull*- style interfaces. Suppliers and consumers can arbitrarily mix their usage of either interface style. For example, a *push* supplier can send events through the event channel to a *pull* style consumer.

Additionally, all event transfers occur synchronously. That is, unless the application specifically uses the CORBA Asynchronous Method Invocation (AMI) facility, calls that send events will block until the event is received by the channel. This can affect the performance of both the supplier and the event channel if network propogation delays are significant. For realtime applications, this can also increase the likelihood of priority inversions [47]. As such, the standard CORBA event service is not commonly used in realtime environments.

[1] Typed event service implementations are much less common than what is described here.

For embedded applications, the footprint size of an event service is important and may limit the ability to use an event channel if it is too large. One data point for the size of a high-quality implementation of the OMG Event Service is the implementation in TAO. Under Linux, the amount of code and initialized data for the Event Service is currently 8,590,080 bytes.

2.5.2 OMG Notification Service

The OMG Notification Service is an extension to the OMG Event Service with the goals of providing mechanisms to filter events and of allowing some quality of service (QoS) metrics to be communicated to the event channel. It provides interfaces that support the transport of *structured events* that contain fields visible to the event channel. A structured event consists of an event header that contains fields such as the event type, name, and a variable section that has options for handling the event. The event body contains a variable number of user-defined, filterable fields and a CORBA Any payload.

Event filtering is specified using Extended Trader Constraint Language (ETCL). By using the filtering capabilities, an application need not create a separate event channel for each category of event. Additionally, ETCL provides the Notification Service with one of the most flexible filtering specification languages of any CORBA event service. This flexibility, however, adds significantly to the footprint and run-time costs of the event channel. The TAO implementation of the OMG Notification Service addresses some of these performance issues [23].

QoS parameters such as event priority, delivery time, and persistence can also be specified using Notification Service interfaces. The consumer can configure its associated delivery queues to reorder events based on priority or the earliest deadline. However, this feature may depend on the quality of the implementation of the service. For high reliability uses of the Notification Service, the suppliers and consumers can also create *persistent* connections and avoid missing any events due to transient failures and unintended disconnects.

The main downside to using the Notification Service is its shear volume of code. The TAO implementation is 24,000,311 bytes, and this does not include the additional overhead of the ETCL support code. As a result, the use of the Notification Service in embedded environments is severely limited.

2.5.3 TAO Real-time Event Channel

The TAO Real-Time Event Channel (RTEC) adds event delivery guarantees to the standard Event Service model so that it is suitable for use in realtime environments. Like the Notification Service, it also uses well-defined structured events. These events have been further optimized, for example, by using a fixed header and by using a CORBA Octet Sequence type instead of a CORBA `Any` to transport the event payload.[2]

In addition to supporting event filtering, the TAO RTEC also supports event correllation. This allows consumers to register with the event channel that they should not be notified until a specified sequence of events arrives. By using this mechanism, consumers can reduce network communication and limit processing events until all data is available.

Since event reception drives consumer processing, the TAO RTEC also supports scheduling event delivery and hence scheduling the consumers. Scheduling is performed offline using Rate Monotonic Scheduling (RMS) [29], and then the computed schedule is configured at runtime.

Although the RTEC provides much more functionality than the OMG Event Service, its implementation for TAO has only a slightly larger footprint of 9,010,532 bytes.

2.5.4 Feature Summary

Each of the previously described event services provides a fixed set of features that are summarized in Figure 2.4. Ideally, when designing a system that needs an event service, an application developer selects the channel that has a feature set close enough to the requirements of the application. The result of the selection has the following outcomes:

1. The feature set exactly matches the application's requirements.
2. The feature set lacks some functionality.
3. The feature set has some functionality that is not needed or used.
4. The feature set both lacks some functionality and provides unneeded functionality.

[2] Results from this thesis indicate that these changes can significantly increase the performance of an event channel at the expense of providing a more complex interface to the user.

Only the first of these outcomes is desirable, but this outcome is also the lease likely. If the chosen event service lacks some functionality, the application developer needs to develop mechanisms to support it, and if there is unused functionality, the footprint size and possibly the performance may be degraded. This thesis addresses both of these issues by presenting practical methods for developing and using composable middleware using AOP techniques.

Feature	OMG Event Service	OMG Notification Service	TAO Real-time Event Service
Basic Event Service Structure	yes	yes	yes
CORBA Anys used for events	yes	yes	no
Push interfaces	yes	yes	yes
Pull interfaces	yes	yes	no
Structured events	no	yes	yes
Event filtering with boolean expressions	no	yes	yes
Event filtering with ETCL	no	yes	no
Consumer registration introspection	no	yes	no
Supplier registration introspection	no	yes	no
Event sets (sequences of events)	no	yes	yes
Event message translation	no	yes	no
Event domain specified in header	no	yes	no
Event type specified in header	no	yes	yes
Event name in header	no	yes	no
Variable length header	no	yes	no
Filterable values in event bodyes	no	yes	no
Persistent events	no	yes†	no
Prioritized events	no	yes†	yes
Event delivery timeouts	no	yes†	no
Persistent event channel connections	no	yes†	no
Bounded consumer event queues	no	yes†	no
Scheduled event delivery	no	yes†	no
Event pacing	no	yes†	no
Event correlation	no	no	yes
Event channel timers	no	no	yes
Offline event scheduling	no	no	yes

† indicates optional feature.

Figure 2.4: High level event channel feature summary.

Chapter 3

FACET Architecture

This chapter describes the architecture of the the Framework for Aspect Composition for an EvenT channel (FACET) event channel. First, a high level overview of the components that make up FACET is presented, and then subsequent sections describe the architecture of these components. Of particular interest are the important design tradeoffs and decisions.

3.1 High Level Overview

Figure 3.1 depicts the five major components that are fundemental to the FACET middleware. Each of these components interacts in some way with each of the other components, and without such interaction, some major functionality would be lost.

The implementation of the event channel is first separated into a base and a set of selectable *features*. The base represents an essential level of functionality. Each feature adds a structural and/or functional enhancement to the base or to other features, and Aspect-Oriented Programming (AOP) language constructs integrate or *weave* feature code into the appropriate places in the base as well as the features.

In FACET, the base consists of a simple implementation of interfaces similar to those found in the the Common Object Request Broker Architecture (CORBA) Event Service with a few caveats:

- The *pull* interfaces and their implementation are not included, since they are much less frequently used.

- Since the event payload type varies with each application, it too has been designated to a feature.

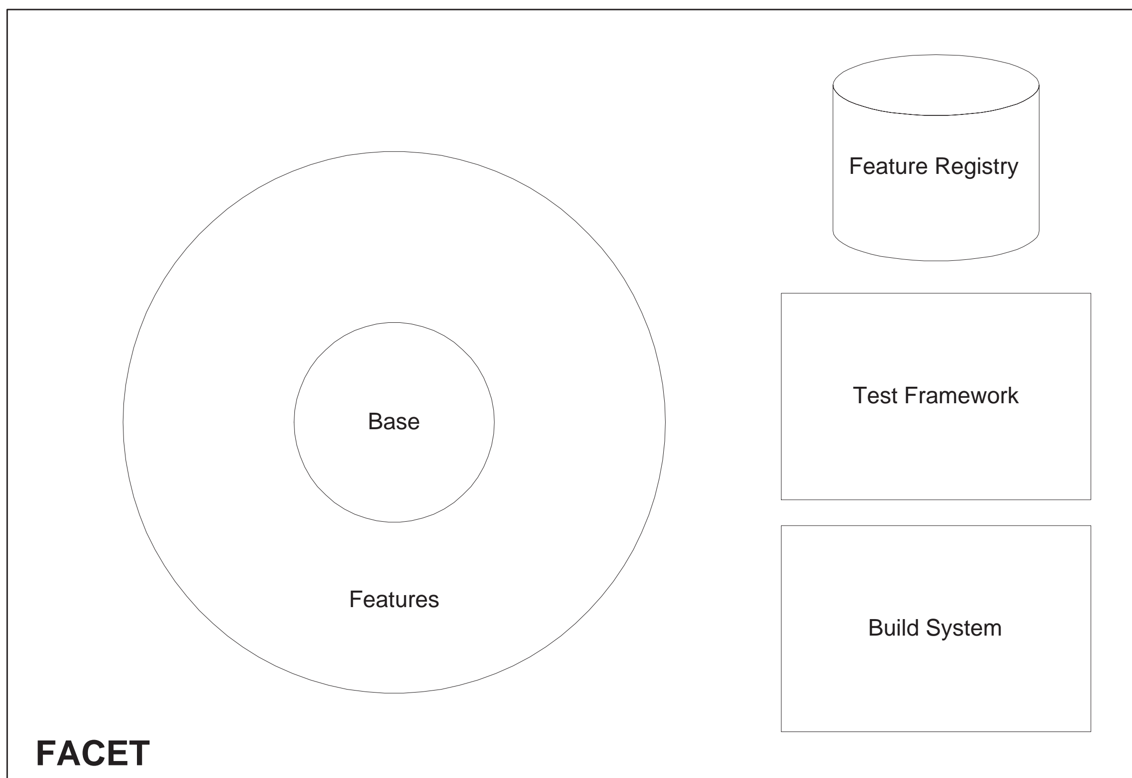

Figure 3.1: The main components in FACET.

Our version of the base can thus be equated more accurately to an interrupt service, where consumers are only notified that *something* happened without any details of *what* happened.

To support functionality not found in the base implementation, FACET provides a set of features that can be enabled and combined, subject to some dependence constraints. These features include:

1. Interfaces and implementation to support *pulling* events through the event channel.
2. Various event-payload types such as CORBA Anys, CORBA octet sequences and strings.
3. Event structures such as headers that are made visible to the event channel and used by other features. These include event type-labels for dispatch and filtering, a time to live (TTL) field to support federated event channels, and timestamp fields for profiling.
4. Dispatch strategies that trade off channel performance and memory usage.
5. Event-correlation support that allows consumers to specify sequences of events that should be received by a channel before notification.
6. Event-channel profiling and statistics generation.
7. Tracing hooks to aid application debugging.

In addition to the base and features, Figure 3.1 illustrates three other major components in FACET. The Feature Registry maintains all of the relationships and metadata concerning every feature. It has the responsibility for validating event-channel configurations and providing dependence relation information to the other components. The Build System is then responsible for selecting and compiling the appropriate source files that correspond to the desired feature configuration. Both of these components are described in detail in Chapter 4.

Finally, the Test Framework has the responsibility of verifying that each feature and its compositions perform actually as intended. It is used to gain a high level of confidence that changes to the base or to other features do not have unintended consequences in any configuration. Chapter 5 provides a description of this component.

3.2 Defining the Base

One of the most important decisions to make when constructing highly subsettable middleware using AOP techniques is to define the functional boundaries of the base implementation. Furthermore, since every feature implicitly or explicitly references the base, it is one of the first software units that must be designed. It is important that the base stabilize, since changes to the base may involve changing every feature. Additionally, the use of AOP—in particular, AspectJ—affects the design of the base due to restrictions on the kind of manipulations that can occur in features. The following design forces are, therefore, important to address when designing the base:

1. *The base should not contain functionality that is disabled by features.* This design force is based on the additive nature of AOP techniques. As stated before, AOP allows one to add code at pointcuts and introduce new methods and class variables but not remove code. By using *around* advice, this restriction can be mitigated somewhat, but the result can become confusing. For example, if one feature disables a method call and another feature adds some specialized advice to it before each invocation, should the later feature's advice be run? AspectJ provides language constructs for specifying the choice, but each feature needs to know about the other. This coupling is undesirable; furthermore, features added in the future may have to be modified to run or not run the appropriate code.

   A simpler solution is to exclude the optional method in the base. A feature can thus be included only if it is needed. If another feature adds advice to that method, and the method is absent, then that advice is not applied. If the features needs its advice to always be applied, then a different pointcut should be found (i.e. in the base implementation.)

2. *The base should contain a sufficient number of joinpoints to enable the process of writing feature aspects.* AspectJ and other AOP languages generally limit where advice can be applied. For example, AspectJ limits advice application to method calls and variable accesses. Thus, the base must have enough of these to make it easy (or at least possible) to write the necessary aspects in the features.

   Luckily, good programming practices such as writing small methods and creating a rich type hierarchy are helpful in this regard. Also, using a consistent

class structure for the base and all features helps ensure that attaching to the joinpoints in the base results in the same semantics as when other features are enabled. For example, in FACET, the base class structure very closely resembles the class structure of the CORBA Event Service even though only a subset of the functionality is present. Adding features to FACET only augments this structure rather than changing the procedure for sending events through the channel. In retrospect, the main changes to the base were to decompose existing methods to add more joinpoints.

3. *The base should perform some functionality that represents the processing of the middleware framework.* This design force is the result of the practical necessity for the base to be simple to understand. The design forces considered so far tend to reduce the base in scope, since that provides the most flexibility when adding new features. In fact, a theoretically pleasing base may be devoid of any functionality, since this gives the most flexibility by far to the feature set.

   A limitation of aspects in separating concerns, however, is that when used too heavily, it becomes difficult to determine what any method actually does. Programming tools such as the *emacs* extensions that come with AspectJ help ameliorate this limitation, but currently, it is still necessary to view several files to determine everything that a method does when it has been modified by an aspect. Since this is cumbersome, the base of FACET has *some* functionality that is representative of an event channel. Through the process of building it, though, that functionality has been reduced to the point of a *interrupt* service, as noted in Section 3.2.

3.3 Features

Nearly every usable configuration of FACET contains at least one feature. This section describes the important pieces found in every FACET feature as shown in Figure 3.2. Note that a detailed description of the currently available features can be found in Appendix B. The main implementation of each feature is comprised of aspects, Java classes and interfaces, and Interface Definition Language (IDL) *introductions*. Unit tests are associated with each feature, which can be excluded for *release* builds. Finally, every feature contains metadata describing its dependendence on other features.

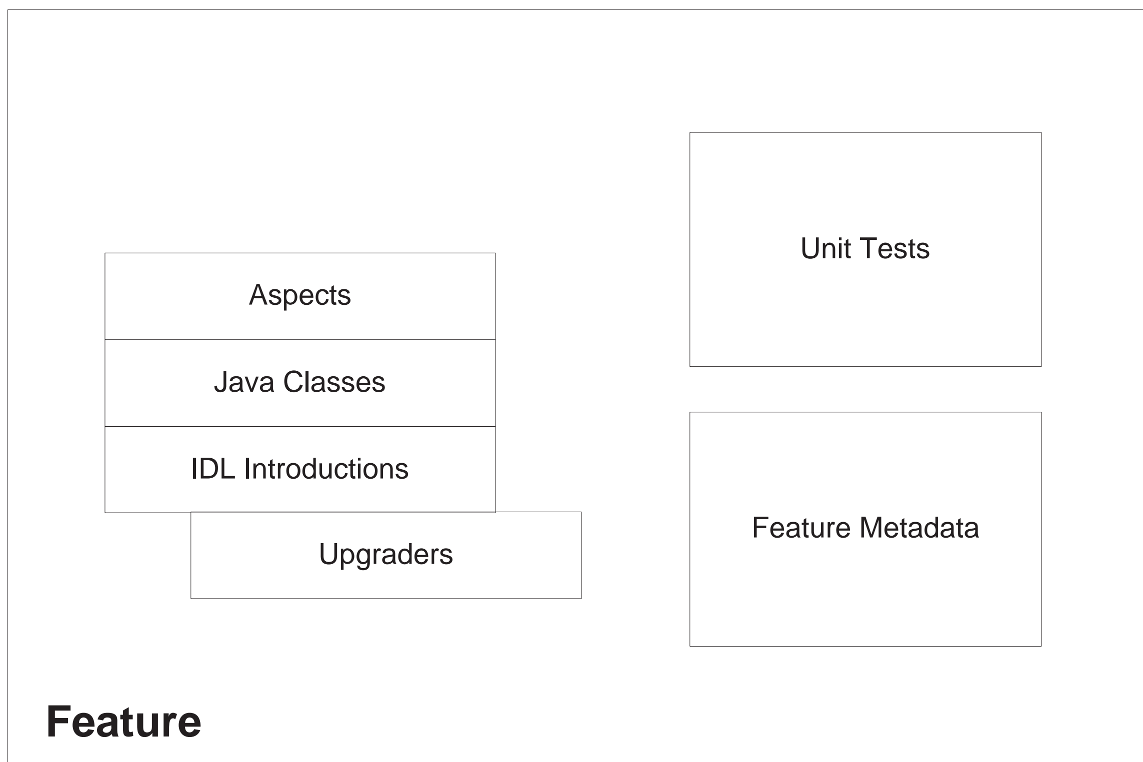

Figure 3.2: Components of FACET features.

In the implementation, aspects enable features to insert *hooks* (in the form of advice) at convenient locations in the base and to introduce fields into existing classes or data structures. The ability to insert hooks at key pointcuts is crucial for a feature to affect the operation of the base event-channel.

For example, one feature adds *filtering* to the event channel. It does this by wrapping a call to *push* events to consumers in the default dispatching mechanism. If an event does not survive the filter, it is simply dropped. Otherwise, events are forwarded to their appropriate consumer by invoking the existing, *wrapped*, method in the base to pass the events onward.

The statistics-collection feature also uses aspects to advise key points in the reception and delivery of events to adjust counters appropriately. Again, the main benefit of using aspects is that the advised code need not be aware that it is receiving advice, and as such, the standard hooks and strategies do not need to be designed upfront.

Aspect *introduction* or the ability to introduce new methods and variables to existing classes is useful when features need to store extra information about options in event and parameter structures. This is used to store quality of service parameters for easier access during the critical event delivery path and also to event fields such as the event type and TTL.

Java classes and interfaces are also key components of each feature. These are needed when the code that would be in an aspect is complex enough to warrant supporting classes. A prime example of this is the *correlation feature*, which has many support classes for the construction and evaluation of event-matching grammars. It should be noted that during the initial stages of constructing FACET, it was thought that features would be implemented solely in terms of aspects. From a code maintanence perspective, this quickly became impractical. The result for many features is that aspects are used to group together all of the interception points and high-level feature-logic, and auxiliary classes are used for the lower level implementation.

Since FACET is a CORBA event service, it supplies IDL specifications for its external interfaces. The base defines interfaces for the standard event-channel adminstration and registration components. Most features provide some introductions to the base IDL to export entities such as event types, payloads, additional adminstration methods, the pull interfaces and more. Of course, once a feature adds definitions to the IDL, it must also use aspects and classes to implement those interfaces. The

pull feature is a prime example of IDL introductions. It not only adds new interfaces to support pull suppliers and consumers, but also adds the factory methods to the event channel adminstration interfaces to instantiate those interfaces. The pull supplier and pull consumer interfaces are implemented using Java classes, and the factory methods are implemented using aspect introduction to the event channel adminstration implementation in the base.

The next main component of a FACET feature is called an Upgrader. It is usually one aspect that adds advice to existing applications and test code that was written without knowledge of the feature. The advice handles any new initialization and registration that is needed for this feature to work in an expected, default way. A prime example of the necessity of an Upgrader is for the TTL feature. This feature simply adds a TTL field to every event structure and decrements that field at every event channel hop. Since events get dropped when their TTL is zero, this adds the precondition that before sending any event, the TTL field needs to be greater than zero. The problem occurs when an application was written without knowledge of the TTL feature and never sets it. Then, as soon as the TTL feature is enabled, all events are dropped since the field default in Java is zero. The TTL feature's upgrader adds code to do this in every class that creates an event but does not know to set the field. The Upgrader concept is crucial for the proper operation of the test framework and is documented there.

Feature metadata will be described in detail in Chapter 4; here we describe its two main pieces. The first is a feature interface that is named after the feature and *extends* the feature interfaces of all other features on which it depends. Note that although not all dependence relationships are the same, the fact that feature interfaces inherit from their dependences will be very useful for feature management and resolving composition issues in testing. The other part of the feature method is an aspect that registers the feature with the Feature Registry. This is needed to construct the feature dependence graph that is used by the build system and the test framework.

Finally, the last component of any feature is a set of unit tests. During the development of FACET, verifying all of the combinations (currently, over 9,000 valid combinations) of event channels became very tedious. The use of automated unit tests made this process significantly easier. The use of unit tests is also a good development practice for any software project.

3.4 Adding New Features to FACET

The process for adding new features to FACET is surprisingly simple. One of the main advantages of using aspects is that very few irrelevent details or hook methods are present in the base or feature code that is augmented. Overall, this procedure consists of the following steps:

1. *Decide what existing FACET features are required.* To allow for the broadest possible use of the new feature, it is important that this set of features be as small as possible. A side effect of this is that debugging is simpler since less code is involved.

2. *Create a feature interface class that extends the feature interfaces of dependent features.*

3. *Within the feature interface class, create an aspect to register this feature with the* FeatureRegistry. If the feature is an *abstract* feature, it is necessary to implement a dependent concrete feature so that a valid event channel configuration can be actually be compiled and tested.

   Also, if the feature *contains* another feature, it must notify the FeatureRegistry of the relationship here. If that contains relationship is with an abstract feature, then the feature should not assume any functionality that makes that abstract feature concrete. See Section 4.1.2 for more information on this issue.

4. *Write the code using ordinary* Java *classes and aspects to implement the feature.*

5. *Write a testcase to validate the feature.* Just as the feature code only assumes the minimum number of dependent features, so should the test case. In FACET, test classes always begin with the word *Test* so that the build system can easily remove them when not needed by the application.

6. *Write an aspect to mark the test case classes and any other relevent classes as Upgradeable.* This aspect should modify these classes to implement the empty Upgradeable interface and implement the feature interfaces that this feature depends upon. This is necessary to allow features that change how registration occurs, add parameters to common requests, or change event registration to fix up code that was written without their knowledge. This is described in detail in Chapter 5.

7. *Write an Upgrader aspect to modify code in other features to work when combined with this feature.* Advice should be applied to only those classes that are Upgradeable and do not implement this feature's feature interface class.

Chapter 4

Feature Management

As the number of features supported by the Framework for Aspect Composition for an EvenT channel (FACET) grew, managing the different combinations and their dependences quickly became tedious and error prone. It was apparent that traditional software configuration techniques would not address the concerns of highly configurable software and that it would be necessary to build a feature-management infrastructure for FACET. This chapter describes this infrastructure and the issues it addresses in managing large numbers of features.

4.1 Feature Registry

The Feature Registry maintains all of the relationships between features and provides interfaces to query those relationships. All of the functionality provided by the registry is completely generic and not tied to the FACET event channel. Yet, nearly every part of FACET takes advantage of the Feature Registry, including the build environment, test environment, and statistics-collection framework. Additionally, every feature must interact with the Feature Registry to inform it of its requirements. In this sense, FACET provides a higher level Aspect-Oriented Programming (AOP) meta-programming framework for middleware. The following sections describe how features are modeled internally and what information the Feature Registry maintains.

4.1.1 Types of Features

Features in FACET can relate to each other and to the base in several different ways. These relationships are important, since they essentially determine valid configurations of the middleware. Fundamentally, each feature can be assigned to one of the following categories based on the usage requirements of the feature:

1. *Concrete Features:*These features can be included in any configuration, given the stipulation that any feature on which they depend are also included.

2. *Abstract Features:* These features provide a structural or functional enhancement that is *incomplete* and cannot exist on its own. A concrete feature must augment this feature for the configuration to be valid.

   As an example, consider the introduction of a header to a structured event. Initially, the header is empty. Until the header contains at least one data field from another feature, it is not useful and does not even compile with the the Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL) compiler.

3. *Mutual Exclusion Features:* Such features are *mutually exclusive* in the sense that at most one can appear in a valid configuration. For example, the type of event passed through FACET may be either a structured event or a CORBA Any, but not both at the same time.

4. *Inferred Features:* These features exist only within FACET and are created when one feature refers to a nonexistent feature. That nonexistent feature is *inferred*, and if it is never loaded, it signals a configuration error. This would occur if the user forgets to specify a dependent feature in their configuration. The FeatureRegistry would create an inferred feature as a place holder for the missing feature, and when validating the dependence graph, it would detect the error. Note that the FACET build system actually protects the user from this type of mistake by automatically including all dependent features, but they may be created temporarily while the dependence graph is built since features need not register in topological order.

The base is modeled as a concrete feature with no dependences. Every other feature depends on another feature or on the base, and it is always possible to reach the base from any feature by following dependence relationships.

4.1.2 Relationships between Features

There are two types of dependence relationships:

1. *Depends:* Most relationships between features are of this type. A feature that depends on another cannot exist in a valid configuration unless all of its dependences are also part of the configuration. Furthermore, this relationship serves to satisfy the requirements of the feature types described above. For example, a feature that depends on an abstract feature also indicates that it supplies the necessary code and data to be able to use that abstract feature. The event source-field feature is an example of a feature that depends on the abstract event-header feature. Its inclusion satisfies the requirement that the abstract event header is completed by at least one conrete feature.

2. *Contains:* Some features create or use data structures that *contain* data structures introduced by some other feature. These features still depend on the presence of the other feature but cannot be used to fulfill dependence requirements of that other feature. An example of this is the *pull* feature that allows users to be able to pull events through the event channel. This feature does not care what kind of event is used, but it does care that an event type feature has been enabled.[1] Since enabling the pull feature does not *complete* the event type, the depends relationship cannot be used, and therefore, it is said that the pull feature contains the event feature.

After the dependence graph has been constructed, it can be validated in time linear to the number of features. This is performed by inspecting the in-degree of each feature node. For example, mutual exclusion features require an in-degree of exactly 1 *depends* relationship, abstract features require an in-degree of 1 or more *depends* relationships, concrete features have no requirement, and inferred features require an in-degree of 0 of any dependence relationship. Note that this model is easily extended to include other conceivable types of features that differ in their in-degree requirements.

[1]Theoretically, the pull feature could be implemented to handle the case where no event type has been selected. However, this case has very limited usefulness compared to the amount of complexity added to the feature code.

4.1.3 Feature Cycles

In the general case, it is possible for a cycle to be created in the feature graph. At its simplest, this occurs if each of two features depends on the presence of the other. For example, in large software projects, this can happen if two development teams are tasked to develop a relatively large feature. Although these two features would ideally be represented by only one feature, the practical organization of the project coerces the division. As this division is unnecessary, cycles such as these (and all other cycles) are not supported or considered in the Feature Registry. Fortunately, cycles in the feature graph are infrequent (and difficult to create by accident) in practice.

4.1.4 Feature Registration

In order for the Feature Registry to manage feature dependences, every feature must register its dependences at initialization. This is accomplished using the Template Advice pattern (see Section 6.2) so that the registration is performed in the feature as opposed to a centralized location. This has the advantage that the feature metadata (a feature concern) is kept with the feature implementation.

Every feature defines an empty interface that serves to identify itself uniquely. This interface, called the *feature interface*, is used by the Feature Registry internally, by other features when they register their dependences, by the build system, and by the test environment. The feature interface extends all of the immediate feature interfaces on which it depends. Figure 4.1 shows a feature interface for the Event Pull feature and its associated registration aspect. Since the Event Pull feature cannot work without the Event Struct feature, its interface extends the Event Struct's associated feature interface.

Figure 4.1 also shows the aspect that is used to register this feature with the Feature Registry. The Register aspect extends the AutoRegisterAspect abstract aspect as part of the Template Advice pattern. Abstract aspects are similar to abstract classes in Object-Oriented Programming (OOP) languages. In the Template Advice pattern, the pointcut and the advice location are defined in the abstract aspect. Derived aspects, such as Register, fill in the processing that should occur. In this case, such processing serves to register the feature interface with the Feature Registry. The registerFeature method accomplishes just this.

Other methods register abstract and mutual exclusion features. The second method call in Figure 4.1 marks the dependence relation between the Event Pull

```
public interface CorbaEventPullFeature extends CorbaEventStructFeature {

    static aspect Register extends AutoRegisterAspect {
        protected void register(FeatureRegistry fr) {
            fr.registerFeature(CorbaEventPullFeature.class);

            fr.markContainsRelationship(CorbaEventPullFeature.class,
                                        CorbaEventStructFeature.class);
        }
    }
}
```

Figure 4.1: A feature interface and registration aspect.

feature and the Event Struct feature as a *contains* relationship. The Feature Registry uses the Java reflection mechanism to determine all of the dependences of a feature interface by looking at all of its parent interfaces. Since the most common dependence relationship is the *depends* relationship, it assumes this relation unless told otherwise as in the above example.

4.1.5 FACET Feature Dependence Graph

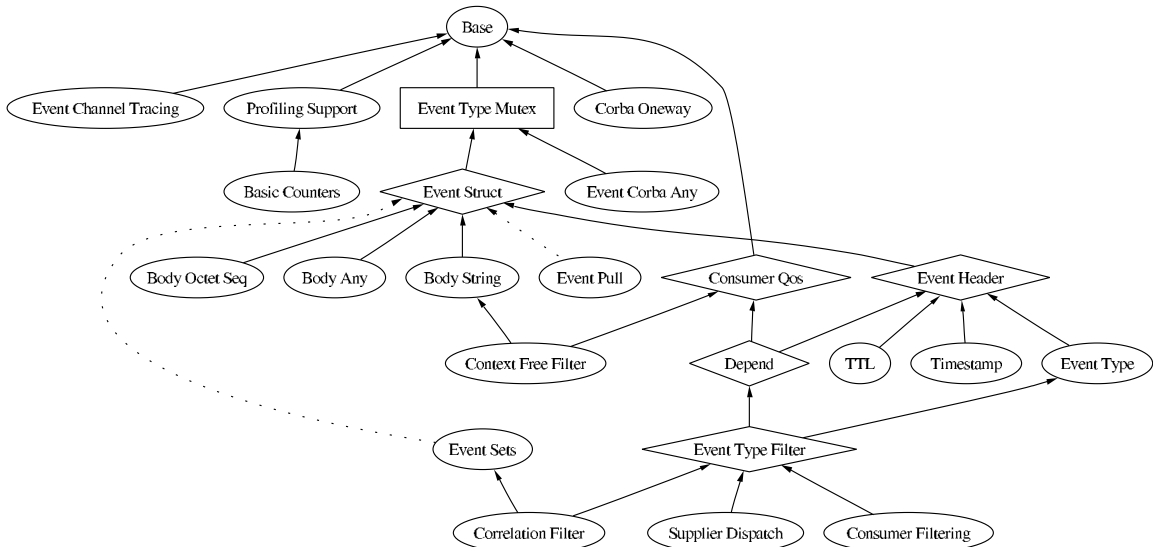

Figure 4.2: Feature Dependence Graph: Oval nodes are concrete features, diamond nodes are abstract features, and rectangular nodes are mutual exclusion features.

Figure 4.2 is a *feature dependence graph*, which shows the relationships between the base and the current features implemented in FACET. In this graph, oval nodes are concrete features, diamond nodes are abstract features, and rectangular nodes are mutual exclusion features. Nodes that are related by the depends relationship are shown with a solid arrow, and those related by the contains relationship are shown with a dotted arrow. The features themselves are described in detail in Appendex B.

4.1.6 Combining Features

As evaluated in this thesis, FACET has 21 different features. Ideally, we would like to verify every possible combination of features and measure the resulting effects on performance and footprint. A naive approach would try all 2^{21} or 2M combinations.[2] However, due to the dependence relationships between features, the actual number of valid configurations is *much less* in practice. It is possible to enumerate all possible combinations by traversing the dependence graph. Performing this with the current feature set yields only 4,596 distinct, valid configurations.

For users of FACET, it is important to validate that the chosen feature set actually does satisfy all dependence constraints. The Feature Registry supports this by iterating over the features that have been registered. Note that the build environment ensures that only the code from the selected features gets compiled, and because of these, only the selected features register with the Feature Registry. At a high level, the Feature Registry checks for the following conditions:

1. The dependence graph contains no inferred features. Equivalently, the targets of all feature dependences have been registered.

2. All abstract features have at least one feature that *depends* on them.

3. All mutual exclusion features have only one feature that *depends* on them.

Currently, since feature meta-data is kept in Java interfaces, it cannot be checked until runtime. With FACET, this checking is performed automatically at build-time after the FACET library has been built.

Moreover, it is important to note that not all feature miscombinations result in compile- or build-time errors. For example, the AspectJ compiler does not issue a

[2] Although the order in which features are applied is important for the compiler, for any one set of features, the resulting event channel configuration is the same. Hence, the number of potential feature combinations is not influenced by the ordering of those features.

warning or error it a specific pointcut is missing. This could be the case if a dependent feature was not enabled. Another example of a error not caught is the inclusion of dead code from a feature that should have been invoked by a missing feature.

To further verify that every configuration is truly viable and defect-free, each feature provides one or more unit tests. Chapter 5 describes the verification process and the use of the Feature Registry to automate testing. When statistic collection is performed on each configuration, all relevant unit tests are also run. Chapter 7 documents these results.

4.2 Build Environment

In most systems, a project's build environment is a secondary concern. When developing customizable software, however, the build environment is usually charged with determining what features are included in the delivered library or executable. For FACET, the environment has an even greater importance, due to the large number of features that are available. If the build environment does not provide a logical, simple configuration mechanism, it will likely frustrate potential users.

The FACET build environment was designed with several goals in mind:

1. Feature identification should be as automated as possible. For example, adding new features to FACET require as few changes as possible to the build environment.

2. The environment should be portable.

3. The resulting code should be validated against unit tests for all selected and dependent features.

4. The user should be able to specify only those features that are direclty needed. Dependent features should be added automatically.

Of these, portability is achieved by using the ant build tool [2] which is essentially a *make* utility designed specifically for building Java applications. Feature-set testing and verification uses information found in the Feature Registry and is described in detail in Chapter 5. The following sections address the remaining goals.

4.2.1 Feature Organization

In the early versions of FACET, feature addition involved registering the feature with the Feature Registry and with the build system, and then registering the added feature's source files with the build script. This process was tedious and error prone, but still easier than the default means of selecting features in AspectJ by using source-file lists. A fundamental problem with using the Feature Registry is that its information is supplied by the features as they register themselves at run-time; unfortunately, the build system requires similar information at compile-time. This section describes the evolution of the feature organization and how it is used to automate the build process so that as much information about each feature can be reused as possible.

To separate feature code from the base FACET code, every feature is implemented in a separate directory and separate Java package. As per common Java practice, the package name of a class and the location of the source files for that class within the directory hierarchy are closely related.[3] The build system takes advantage of this and uses the package name of a feature to refer to it. When a feature is enabled, the build system uses wildcard expression-matches to find and then build all source files in the feature's directory.

The build system cannot know which directories contain features without input from the Feature Registry. To provide this input, a special ant build target can be invoked in the build files to scan the FACET subdirectories for feature interfaces and compile them with the Feature Registry. By compiling all feature interfaces and the Feature Registry code, the Feature Registry will know all of features available for use in FACET and their dependence relationships. A simple Java program is then run that distills this information into a build file. In many ways, this process is analogous to the *makedepend* utility except that it informs the build system of the *locations* of features as well as their dependences.

Following are the simple devices used to automate feature inclusion in the build system:

1. Including the word *Feature* in the name of all feature interfaces so that the build system can easily find them for the Feature Registery.

2. Identifying features by their package name. This allows the Feature Registry to find the features' names by introspection, and it allows the build system to find

[3] For example, the package edu.wustl.doc.facet.corba_ttl has the directory offset of edu/wustl/doc/facet/corba_ttl.

the feature code due to the Java convention of naming directories to correspond to packages.

4.2.2 Feature Selection

Features are enabled and disabled by using a configuration file. Figure 4.3 shows one such configuration file. Each listed directive corresponds to a desired feature that is identified by "use_" and the identifying part of the package for that feature.

```
use_event_pull=yes
use_tracing=yes
use_eventbody_any=yes
use_corba_eventtype=yes
```

Figure 4.3: An example configuration file.

As in the previous section, the Feature Registry is used to add support for these directives in the build system. Additionally, the build system automatically includes all dependent features for those that are listed. This greatly reduces the chance of specifying invalid FACET configurations. It does not completely eliminate it, though, due to the *contains* relationship. For example, the Event Pull feature *contains* the abstract Event Struct feature. Its interfaces contain only references to Event structures, so some other feature is needed to make the Event Struct concrete. In Figure 4.3, the features that do this are the Body Any (use_eventbody_any) and the Event Type (use_corba_eventtype) features. If a user specifies only the Event Pull feature, then neither the Feature Registry nor the build system could know which features to include to make the Event Struct feature concrete. In this case, this error is caught by invoking the verify operation on the Feature Registry.

4.3 Aspect Support for Multi-Languages Environments

A final issue encountered in managing features in FACET was handling multiple languages. Since FACET uses CORBA, it must specify its external interfaces using IDL. Many features need to introduce new methods, new classes, and new structures to the IDL interfaces. This cannot be accomplished with AspectJ. The approach

taken with FACET is to provide Python [30] scripts that are integrated with the build system to introduce IDL definitions into the appropriate files. This procedure is error prone and will eventually become difficult to scale. Ideally, it should be possible to encapsulate concerns that cross language boundaries such as these. This, however, is not addressed by this thesis and is an area for future research.

Chapter 5

Testing

Software verification is necessary and important for any application. Proper testing is even more important for the Framework for Aspect Composition for an EvenT channel (FACET) than for many other software projects for two main reasons:

1. FACET supports a large number of different configurations of features that interact with each other in numerous ways. Validating a subset of legitimate configurations does not guarantee that every configuration will work or even compile.

2. It is difficult to verify that a change made to the base or a feature does not remove or change the semantics of a joinpoint used in another feature.

Because of these reasons, FACET provides a test framework that automates the test process.

5.1 Test Framework

jUnit[20] is a commonly used framework to automate the regression-testing process for Java applications. Its Application Programming Interface (API) provides various methods to validate code and report errors. Additionally, it comes with GUI and text based tools that can run one or more tests, create testing reports, and quickly summarize test-run results. FACET uses jUnit as the basis for its test framework, and by default, the build system invokes a jUnit test runner to execute all relevant tests for a configuration after every build.

Although jUnit is very useful, it does not address several issues that arise when developing highly reconfigurable middleware. These issues include:

1. Support for automatically running tests that correspond to the set of features that were enabled.

2. Support for upgrading tests written using one configuration to work under another configuration that includes other features.

Both of these issues are addressed in FACET by using Aspect-Oriented Programming (AOP) techniques. The latter issue, in particular, would have been very difficult to support with standard object-oriented techniques, but with AOP, it is relatively simple. The following sections describe how the FACET test framework manages both of these issues.

5.1.1 Running the Appropriate Subset of Tests

The main requirement for this issue is that when validating an event channel configuration, every test associated with every feature within that configuration must be run. Several options are possible for achieving this:

1. Create a jUnit test suite to call each appropriate test. This is the standard jUnit method for running more than one test. It has the major disadvantage that FACET has thousands of configurations that can be selected. Writing test suites would have to be automated to be practical.

2. Modify the build system to search the directory hierarchy for Java source files that begin with the word *Test* and invoke a jUnit test runner on each of them. Unfortunately, this option adds complexity to the build system, is slow since it has to launch a new JVM for each test, and does not allow jUnit to summarize the results.

3. Use an aspect to automatically register each task with a well-known test suite.

By using standard object-oriented techniques, it is possible to come up with another option that sounds promising but actually does not work in Java. This is to create a static registration method to add test cases to the main test suite and then to add a *static* block to each of the test cases that has a call to this method. This does not work, since Java does not run *static* blocks until class load time, and if no other code references the class, the *static* blocks will never be run. Thus, the third option above is by far the most desirable one.

In FACET, aspects are used to add unit tests automatically by using the Template Advice Pattern (described in Section 6.2.) Figure 5.1 shows the code for the TestSuiteAdder abstract aspect. This aspect encapsulates the knowledge of where and when unit-test registration should occur. Unit tests should "subclass" this aspect and implement the addTestSuites method to register their test or tests with the jUnit TestSuite.

```
import junit.framework.TestSuite;

public abstract aspect TestSuiteAdder {
    abstract protected void addTestSuites(TestSuite suite);

    private pointcut addTestSuitesCut(TestSuite suite) :
        call(void AllTests.addTestSuites(TestSuite)) && args(suite);

    before (TestSuite suite) : addTestSuitesCut(suite) {
        this.addTestSuites(suite);
    }
}
```

Figure 5.1: TestSuiteAdder aspect.

Figure 5.2 shows a common use of the TestSuiteAdder. For almost all unit tests, the test suite registration code is implemented as a static inner aspect to the unit test class. In this case, the time to live (TTL) feature has only one unit test that it needs to register, so the implementation of addTestSuites is very simple.

```
public class TestEventTtl extends EventChannelTestCase {

    /* Test case implementation */

    static aspect AddTests extends TestSuiteAdder {
        protected void addTestSuites(TestSuite suite) {
            suite.addTestSuite(TestEventTtl.class);
        }
    }
}
```

Figure 5.2: Typical use of the TestSuiteAdder.

The use of the Template Advice Pattern also illustrates a technical limitation with AspectJ. For example, it seems it should be possible to write an aspect that

automatically registers every unit test.[1] This is not the case, since aspects can only add advice or make introductions to their target classes. Since both advice and introduction require that the class be loaded at the very least, both have the same problem as the use of a static block in the approach described above. Finally, this limitation in practice is not much of an issue, since defining unit test classes is much more difficult. For example, FACET provides various subclasses of the top-level, unit-test class to simplify the process of starting up the event channel and supplier and consumer threads. Also, during development, it is useful to disable certain complicated unit-tests to focus on fixing bugs. Both of these examples illustrate how automatically creating unit tests using a centralized aspect is difficult and hence motivate why the Template Advice Pattern was used.

5.1.2 Automatically Upgrading Tests

Combinations of features can easily break unit tests. Yet, to validate the operation of FACET, we would like to run every unit test for every feature successfully. An example of this problem is to consider the interaction between the event type feature and the TTL feature.

The event type feature simply adds an event type field to the EventHeader structure. A unit test for the event type feature may send events between suppliers and consumers and test whether the values stored in the field arrive unchanged at the consumers. Such a unit test would have no knowledge of the TTL feature (nor should it).

The TTL feature adds a TTL field to the EventHeader structure and adds code to decrement and check it as events pass through the event channel. A unit test for the TTL feature may check that events with a TTL of zero get dropped and events with other TTL values arrive at the consumer with their TTL decremented. Note that the TTL field must be set by the supplier or it will receive the default value of zero. Since the TTL field is part of the the Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL) specification for FACET, and CORBA IDL does not provide a way to specify default values, this setting cannot be overriden using normal mechanisms.

In configurations that have only one of these two features, their unit tests will work. The problem arises when both features are combined in one event channel.

[1] It is possible to identify a unit test class either based on its name or its parent classes and interfaces.

This causes the event type feature unit tests to break, since they do not set the TTL field in the EventHeader to a positive value. As might be expected, all of its events would then be dropped.

One solution is to write an aspect to intercept executions of the EventHeader constructor and set the TTL field to a non-zero default. This solution has two main problems. The first is that not all feature conflicts can be resolved by modifying a default value. The second is that this change in processing can be seen by user applications. Since the Java IDL mapping specifications specify that the default field value is zero, this change makes the FACET API appear inconsistant with expectations associated with CORBA programming. Also, when the tests are not compiled with the FACET library, the code to automatically set the TTL field may not be included which may then break user code that relies on this behavior.

The solution to this problem is to use the Interface Tag Pattern (described in Section 6.3) to selectively mark the features that a test case supports, and use *upgrader* aspects to update unit test code to support new features. The Interface Tag Pattern is actually used twice:

- To mark those classes that should be upgraded by implementing the Upgradeable interface.
- To mark the features that are known to the unit test by implementing their feature interfaces.

The feature interfaces are the same interfaces that contain an inner aspect to register a feature with the FeatureRegistry as shown in Chapter 4. Figure 5.3 shows the feature interface for the event type feature. Since this feature depends on the event header feature, it extends the CorbaEventHeaderFeature interface. The CorbaEventHeaderFeature interface also extends all of the feature interfaces that it depends upon and so on. Note that the inner aspect is included in this interface only as a convenience, so that all feature management related code can be together. Other than this, this inner aspect is irrelevent to this part of the FACET test framework.

The actual upgrader aspect specifies that any code that it modifies must be inside classes that implement Upgradeable and do not implement its associated feature interface. Figure 5.4 shows the code for the TTL feature's upgrader. Other feature upgraders are similar. The upgradeLocations pointcut limits the applicability of the aspect to appropriately tagged classes. It is separated from the advice for clarity and since more than one advice block may need to use it. Following the pointcut, the

```
public interface CorbaEventTypeFeature extends CorbaEventHeaderFeature {

    static aspect Register extends AutoRegisterAspect {
        public void register(FeatureRegistry ar) {
            ar.registerAspect(CorbaEventTypeFeature.class);
        }
    }
}
```

Figure 5.3: The feature interface for the event type feature.

advice in the CorbaTtlUpgrader intercepts all calls to the EventHeader constructor and initializes the TTL field to a sufficiently large number for any test.

```
aspect CorbaTtlUpgrader {

    pointcut upgradeLocations() :
        this(Upgradable) &&
        !this(CorbaTtlFeature);

    after () returning (EventHeader header) :
        call(EventHeader.new()) &&
        upgradeLocations() {
        header.ttl = 255;
    }
}
```

Figure 5.4: The upgrader aspect for the TTL feature.

As general practice, all test cases should be marked as Upgradeable. When a feature unit test contains many classes, though, the upgradability of the tests is itself a crosscutting concern and can be encapsulated in an aspect. Figure 5.5 shows one way of writing an aspect to capture this concern for the event type feature's unit tests.

In addition to test cases, the upgraders themselves may need to be upgraded. This is the case for the event header feature upgrader. This upgrader creates new EventHeader instances for Event class instances used by test cases that do not know about the event header feature. When, for example, the TTL feature is enabled, the EventHeader instance created by this upgrader now needs to set the TTL field. This is accomplished by marking the event header upgrader aspect as Upgradeable itself as in Figure 5.6.

```
aspect TestUpgradeAspect {

    declare parents:
        ((edu.wustl.doc.facet.corba_eventtype.Test* ||
          edu.wustl.doc.facet.corba_eventtype.Test*.*) &&
         !TestUpgradeAspect)
        implements Upgradable, CorbaEventTypeFeature;

}
```

Figure 5.5: Encapsulating the upgradable concern within an aspect.

```
aspect EventHeaderUpgrader implements Upgradable, CorbaEventHeaderFeature {

    pointcut upgradeLocations() :
        this(Upgradable) &&
        !this(CorbaEventHeaderFeature);

    after () returning (Event ev) :
        call(Event.new()) &&
        upgradeLocations() {
        ev.header = new EventHeader();
    }
}
```

Figure 5.6: Upgrading an upgrader.

5.2 Verifying All Combinations

As discussed earlier, it is important to verify that any valid FACET configuration will indeed work. Testing a change using a few combinations can add confidence that the change does not break other configurations. Based on experience with FACET, testing the minimal possible configuration that uses the change and a configuration with almost all features enabled tends to find most problems.

However, this approach does not yield 100% confidence that any possible configuration will work. For this level of confidence, all configurations need to be tested. This is done by enumerating all possible configurations using the feature dependence graph and then using a script to compile and test each one. Using a 933Mhz Pentium III, each compile and test cycle takes between 30 to 45 seconds. This allows for about 2,000 configurations to be tested per day, so the current 4,596 configurations can be completely verified in just over 2 days. What is remarkable about this is not that it can be done, but that it is actually practical to do so.

Since the number of configurations can grow quickly, especially when features are added with few dependences, testing all of them may become time-consuming if computing resources are lacking. A number of options exist to speed up test process:

1. Run multiple test scripts simultaneously to take advantage of the inherent parallelism in the test process.

2. Incrementally retest only those combinations that that include features that have been modified since the last test run.

3. Mark features to indicate that they should not be tested. The trace feature is an example where this may be desirable. The trace feature only adds functionally useful for debugging, but doubles the number of possible combinations, so it has a big impact on the amount of time full testing takes.

4. Randomly select the combination test order. This does not reduce the amount of time compiling and testing, but it enables a wider variety of combinations to be tested early in the process. From experiences with FACET, random testing enabled most bugs to be discovered very early in the process so that not much time was spent waiting on the test only to have to restart it after a bug.

5. Mark concrete features as abstract when they are very rarely used alone. The main example of this is the profiling support feature. It provides the framework

for accessing various event counters, but does not actually provide any counters. It is concrete but of limited use without a feature like the basic counters feature. Marking it abstract currently reduces the total number of combinations by a third.

6. Randomly sample combinations and determine the resulting confidence level based on the number of samples run.

5.3 Common Mistakes when Writing Feature Unit-Tests

Over the course of developing FACET, several mistakes were made that were not detected until all combinations were tested. Learning from these mistakes is certainly of interest to future feature writers. Application developers may also be interested so that they can avoid errors when enabling new features in FACET. These mistakes include:

1. Using convenience methods generated by the CORBA IDL compiler. Of the convenience methods, the ones that cause the most trouble are the non-default constructors for IDL structures. Figure 5.7 shows the convenience constructor for the Event structure using a configuration with a Any payload and an EventHeader. Under other configurations, the Event structure may have more or less fields, and the IDL generated constructor will be different. The argument ordering is not even guaranteed, so non-default constructors should always be avoided, by initializing public fields after construction.

2. Using functionality in test cases that is not available under every valid configuration. This is very easy to do if not testing with the minimal configuration as has been recommended. If the feature has any *contains* relationships with an abstract feature, this is even easier to do and may not be detected if the abstract feature only has one feature that can make it concrete.

3. Not marking a class as Upgradeable. This particular error usually is not detected until a new feature is introduced into FACET. Since it may only affect a certain combination of features, it may not be detected until all configurations are tested. The procedure for reducing this error is to always mark unit tests

as Upgradeable even if it appears that they will not need it. Using an aspect to mark all unit tests for a feature makes this easy to do. Additionally, the overhead of doing this is minor: it adds a minor increase to the size of the resulting Java class files and causes a few more interfaces to be loaded at runtime.

```
public final class Event
        implements org.omg.CORBA.portable.IDLEntity
{
    public Event(org.omg.CORBA.Any payload,
                 edu.wustl.doc.facet.EventComm.EventHeader header)
    {
        this.payload = payload;
        this.header = header;
    }
}
```

Figure 5.7: IDL generated convenience constructor.

5.4 Genericity of the Testing Framework

Although the description of the testing framework has focused on its use for the FACET event channel, this is not a requirement. Any piece of reconfigurable middleware has to deal with testing issues, and it can reuse the provided framework. The extraction of the feature-management and test framework code and aspects from FACET into a seperate feature management framework would be useful and beneficial to the development of future subsettable middleware.

The ability of the test framework to upgrade test cases automatically to support new features can also offer big advantages for an application. By marking unit tests as Upgradeable and with the appropriate features' interfaces, an application programmer can quickly experiment with new capabilities offered by a middleware framework. This can also be advantagous when integrating two applications that use FACET. If the two applications make use of different feature sets, they can be automatically upgraded to a configuration supporting the union of the feature sets.

Chapter 6

Aspect Oriented Design Patterns

Design patterns are solutions to recurring problems [22]. Patterns are usually identified by reflecting on experiences from previous programming projects where a common problem has repeatedly arisen. Since Aspect-Oriented Programming (AOP) has only found its way into programming projects recently, there is a lack of significant experience to draw upon yet. Indeed, one of the main obstacles to adopting AOP technology in new projects and in the Framework for Aspect Composition for an EvenT channel (FACET) is the lack of knowledge of how to successfully use it.

Over the course of developing FACET, many of the core interfaces and aspects had to be significantly refactored to surmount difficulties when implementing new features. The ability to add new features to FACET using AOP is critical to its ability to be precisely customizable for its users. As a result, much attention was focused on ways of using AOP mechanisms to enhance feature flexibility, scalability, efficiency, and maintainability. The patterns identified in this chapter represent the most common and useful of those found in developing FACET. Additionally, although these patterns are useful in the development of aspect-oriented middleware, they are applicable to the development of any software that uses AOP, as they fundamentally address problems that arise from developing new functionality that crosscuts a base system.

The ensuing sections document the useful patterns in FACET using a standard format for pattern presentation [22]. This format serves to clarify a pattern's role in software development as well as to convince members of the patterns community that the pattern is "real" in the sense that it has application beyond its present usage in FACET.

6.1 Encapsulated Parameter Pattern

Intent

Allow new features to add parameters to Application Programming Interface (API) calls while keeping the programmer's interface simple.

Motivation

It should be possible to add parameters to API method calls to support new functionality added by using aspects. For example, consider an API call to print a formatted message to the terminal. A useful capability to add to this print routine may be to be able to redirect its output else where. By using AOP, one can conveniently create a piece of advice to intercept the call that actually prints the text to the terminal and add logic to support redirection. However, the redirection code still needs to know where to redirect the output. This information needs to be acquired from the programmer somehow. The following are possible approaches to solving this problem:

1. *Introduce a new method to the API to set the parameter.* In this approach, the API user would call the introduced method before calls to the existing method for configuration. In the example, this would involve introducing a method such as `setPrintLocation` and then having the user call it before calls to print. Although such an approach may be acceptable for parameters that rarely change, it is tedious for the programmer to remember to use `setPrintLocation` before each call. Additionally, it is error prone, since the print location may be changed within method calls.

2. *Introduce a new method with the extra parameter and add advice to the old method to call the new one with a default value.* This approach solves the tedium associated with adding another method as above. Additionally, in the example, the user may choose whether to set the output location when calling the print method, and both versions will work. However, this approach has at least the following limitations:

   - Most importantly, it can be used only to add parameters for one feature. This is because each feature needs to introduce a method to override the

base method. If two features need to add different parameters, the result will be two methods with different parameters, but no method with parameters for both features.

- Moreover, this approach requires that code be duplicated in the introduced method since the original method needs to be completely overridden to call the new method with a default.
- Finally, since the contents of the original method are no longer called, the approach tends to create code bloat as methods are overridden.

3. *Use a* cflow *aspect [48] to determine parameters from the context of the caller.* A cflow or control flow construct allows one to attach code to two different contexts that are related based on one calling into the other (possibly through several intermediate method calls). In AspectJ, it is possible to use cflow to extract parameters from a context within the calling application and then use them later, deep within the middleware.

   Applying this to the print method, one would add an aspect that would look at the user's code to determine where the output of the print method should be directed. A clue to where the output should go could be based on the hypothetical observation that it is always desirable to output to the device that is passed as an argument to any of the user's methods. The obvious downside to this approach is that we cannot control how the user writes his or her code, so it is inevitable that we will misinterpret the user's intentions at some point. Additionally, cflow has a performance impact due to its need to maintain a stack internally to save state.

4. *Hard code all possible method parameters to the base methods.* This approach simply adds all of the parameters ever needed by any possible feature to base methods. The base code ignores these parameters, but their presence enables feature aspects to apply functionality to them. Additionally, the API is stable, in that it does not change from the perspective of the user. It just gains functionality as features are enabled. However, this approach greatly reduces the ease with which new features can be added since every feature must modify the base if it adds a parameter. In fact, this approach ends up adding pieces of individual features throughout the base and reduces the advantages of using AOP in the first place.

The Encapsulated Parameter pattern avoids the liabilities of the above approaches by merely passing a structure (or Java class with public member variables) to API methods. Additional parameters can then be added to those methods by introducing new member variables to the passed structure. If a parameter has a default value, it can be initialized in the constructor for the parameter class, so that the user does not need to set it.

Applicability

Use the Encapsulated Parameter pattern when

1. A method call will need to be passed additional parameters to support functionality added using aspects.

2. The new parameters to the method are most logically set when that method is called.

3. The parameters cannot be determined from the calling context or the calling context is unknown.

Structure

Figure 6.1 shows the structure for the Encapsulated Parameter pattern. Since Unified Modeling Language (UML) [36] does not currently support the depiction of AOP interactions, these interactions are shown using stereotypes for advising and introducing functionality to existing classes.

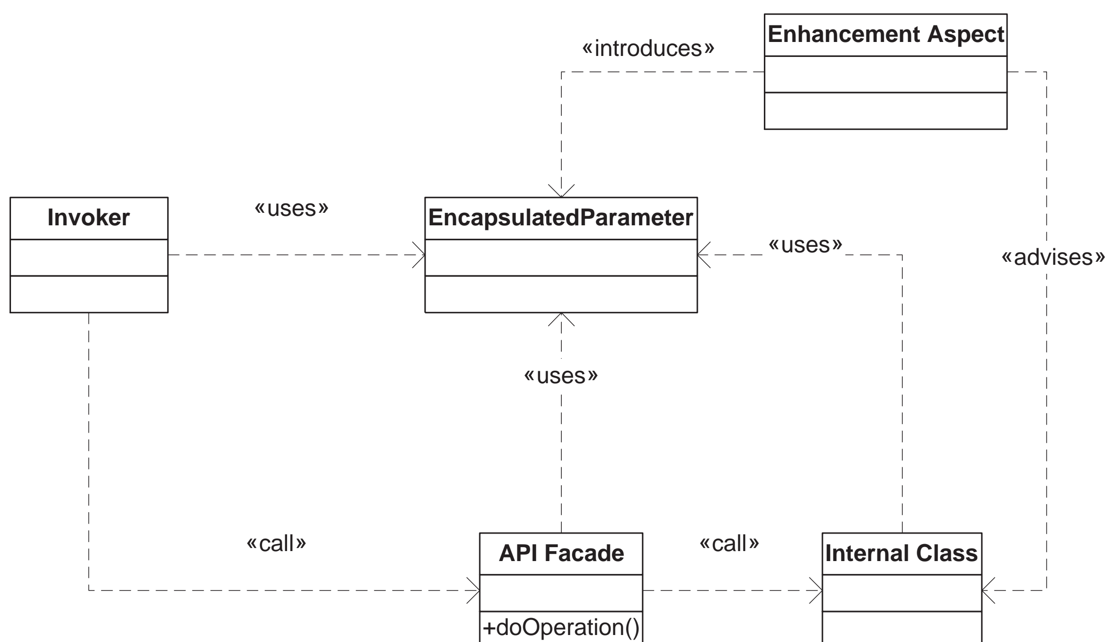

Figure 6.1: Encapsulated Parameter pattern structure.

Participants

Encapsulated Parameter: holds parameters introduced by the Enhancement Aspect.

Invoker: initializes an Encapsulated Parameter instance and passes it to the appropriate API method.

Enhancement Aspect: introduces parameters to the Encapsulated Parameter and adds advice to use the parameters in the Internal Class.

Internal Class: one or more classes that are not directly accessable through the API Facade.

API Facade: defines an interface to a part of the framework to the user.

Collaborations

The Invoker creates an Encapsulated Parameter object and passes it to a method in the API Facade. The API Facade then may pass the Encapsulated Parameter to other Internal Classes. One or more Enhancement Aspect instances can add new parameters to the Encapsulated Parameter class and then use those parameters internally.

Consequences

The Encapsulated Parameter pattern has the following consequences:

1. Makes it possible to extend the parameters of API calls.

2. Simplifies the procedure of writing enhancement aspects, since parameters are easy to access.

3. Allows default parameters to be specified in the Encapsulated Parameter class to simplify the introduction of new enhancements.

4. Complicates the API somewhat, since parameters are not stored in the definition of the Encapsulated Parameter class.

Implementation

Consider the following issues when implementing the Encapsulated Paramater pattern:

1. *Passing the Encapsulated Parameter internally.* Once the Encapsulated Parameter is part of the API, a decision needs to be made as to how far it should propagate through nested calls beyond the API. Often, it cannot be determined ahead of time where the Encapsulated Parameter class will be needed. By using a *cflow* joinpoint, an enhancement aspect can always access these parameters in internal classes that are in the control flow of the API call. Unfortunately, using *cflow* can be computationally expensive, so it may be necessary to pass the Encapsulated Parameter manually.

2. *Including the Encapsulated Parameter in the base.* When the Encapsulated Parameter contains no parameters as is often the case in the base, it is tempting to remove the class entirely from the base. By introducing the Encapsulated Parameter class in a feature, it is necessary to introduce new API functions with the parameter. Since aspects are only additive, the old API functions that do not have the parameter cannot be removed and are still exposed to the user. This confuses the API and if it is normally the case that the Encapsulated Parameter is used, then the simplification of the base is useless.

Known Uses

The Encapsulated Parameter pattern is used in several places in FACET.

1. *Event passing.* The Event class in FACET is itself an instance of the Encapsulated Parameter pattern. It is passed into a ProxyPushConsumer instance by the user to send the event, then passed through the event channel and finally through a ProxyPushSupplier instance to another user. The base code of FACET sends empty events, and features introduce fields into the Event class to hold a payload, headers, source and destination fields, etc.

2. *Consumer registration parameters.* The connect_push_consumer method to register consumers with the event channel takes a ConsumerQOS class to pass in quality of service (QoS), filtering and correlation parameters. Initially, this class is empty, and, for example, when filtering is enabled, parameters are added to this structure to specify what events are desired.

Related Patterns

The Encapsulated Parameter pattern has some similarity to the Command pattern [22], since both patterns encapsulate data in a class that is passed like a parameter. The patterns differ in context, and also since the Command pattern passes code in the class, and it is not meant to be extended through the use of aspects.

At the API interface, this pattern is similar to using named parameters in languages that support this. Named parameters can be specified in any order and can take on default values in the same way as the encapsulated parameter. If Java had support for named parameters, they could be used as an alternative to the

Encapsulated Parameter pattern. However, passing parameters in structures as done in this pattern is convenient and is easy and efficient to pass around internally.

6.2 Template Advice Pattern

Intent

Export key interception points to API users and extension developers and decouple advice from hard coded pointcuts.

Motivation

One of the benefits of using AOP is that it provides a mechanism for extending existing code without explicit modifiation by using an aspect compiler to weave new code at the desirable joinpoints. This ability to break through layers of encapsulation to add cross-cutting functionality is what makes AOP useful. However, specification of joinpoints can be very tricky, especially when the joinpoint applies to unfamiliar code. Additionally, if the base code is undergoing actively developed, that joinpoint may not exist in the next release. Even worse, the joinpoint may be reached in a completely different way in a subsequent release, causing the advice to behave unexpectedly.

AspectJ provides a potential solution to this problem by allowing pointcuts to be specified in abstract aspects and then *concretized* by sub-aspects. By using this mechanism, a core-code developer can specify an interception point by creating an abstract aspect with the appropriate pointcut. A user of that interception point can then create an aspect and inherit the pointcut. This approach still requires that the user know whether before, after, or around advice should be used and if any preprocessing needs to be done to convert parameters from the pointcut to an appropriate *external* form.

Applicability

Use the Template Advice pattern when

1. An interception point may be used by many aspects.

2. It is desirable to decouple the join point and advice location from the actual advice implementation. This may be the case if the base and feature code are developed independently.

3. Common processing is needed to adapt internal variables, parameters and the join point to an exportable form.

4. It is desirable to expose interception points as part of an aspect oriented API.

Structure

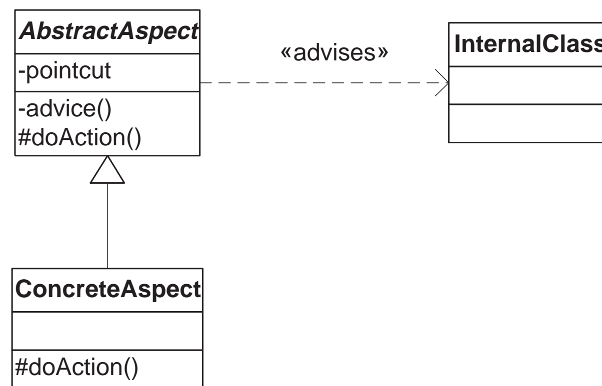

Figure 6.2: Template Advice pattern structure.

Participants

AbstractAspect: defines a pointcut and a skeleton advice implementation that calls abstract methods (doAction) to be filled in by the ConcreteAspect.

ConcreteAspect: implements the logic that should be applied at the interception locations defined by the AbstractAspect.

InternalClass: contains the pointcuts defined by the AbstractAspect and receives the advice from the ConcreteAspect.

Collaborations

The ConcreteAspect relies on the AbstractAspect to execute its methods at the appropriate interception points.

Consequences

The Template Advice pattern has the following consequences:

1. Decouples the pointcut and advice location from the actual implementation of the advice. This adds the flexibility to change internal code without worrying about breaking important pointcuts and retains the advantages of being able to use aspects.

2. Simplifies the extension of a framework by exposing common interception pointcuts.

3. Can limit the parameters accessable to the ConcreteAspect's implementation. This can be advantageous since it reduces the number of variables that need to be considered when extending a framework.

Implementation

Consider the following issues when implementing the Template Advice pattern:

1. *Use access control.* Like the Template Method pattern [22], access control can prevent unintended uses of pointcuts and advice. For example, the pointcut should be declared as private, and the abstract methods in the AbstractAspect should be protected.

2. *One aspect per pointcut.* To reduce the complexity of using the Template Advice pattern, define one abstract aspect per interesting pointcut. Most likely, only one abstract method will be needed for the implementation of the advice.

3. *Provide access to enough parameters to advice implementation.* In order for the ConcreteAspect to implement its advice, it will need some parameters from the interception point. An implementation must decide how many internal details it should reveal to the ConcreteAspect.

Known Uses

The Template Advice pattern is used in FACET to register features with the FeatureRegistry. The FeatureRegistry has an empty method that it calls whenever it needs to build a list of the features in the system. As shown in Figure 6.3, the AutoRegisterAspect abstract aspect contains the pointcut for this empty method. Individual

features derive concrete aspects from AutoRegisterAspect and implement the appropriate registration code. An example of this is shown in Figure 6.4. Note that the actual feature registration encompasses many more details that have been left out here for simplicity.

```
public abstract aspect AutoRegisterAspect {

    abstract protected void register(FeatureRegistry fr);

    private pointcut registry(FeatureRegistry fr) :
        execution(void FeatureRegistry.buildGraph()) && target(fr);

    after(FeatureRegistry fr) : registry(fr) {
        register(fr);
    }
}
```

Figure 6.3: AutoRegisterAspect abstract aspect.

```
aspect RegisterTtlFeature extends AutoRegisterAspect {
    protected void register(FeatureRegistry fr) {
        fr.registerFeature(CorbaTtlFeature.class);
    }
}
```

Figure 6.4: RegisterTtlFeature registration implementation.

Related Patterns

The Template Advice pattern has many similarities to the Template Method pattern. Both patterns define a general skeleton that defers an operation definition to derived types. They differ in their mechanism (use of aspects) and in intent. The intent of the Template Advice pattern is to decouple the knowledge of a pointcut and where advice should be placed from the actual implementation of that advice.

The Template Advice pattern is also related to the Interceptor pattern [46] in its use. Both patterns provide mechanisms to add logic at predefined interception points. The Template Advice pattern, though, takes advantage of AOP techniques to avoid requiring a registry to manage interceptors or the need to add interceptor callbacks throughout the code. Consequently, since Template Advice is applied at

compile-time, it has a higher performance than an equivalent implementation that uses interceptors. Finally, many of the high level design techniques for the Interceptor pattern are also useful for the Template Advice pattern.

6.3 Interface Tag Pattern

Intent

Tag a set of arbitrary classes and aspects as possible recipients of advice.

Motivation

AspectJ enables one to specify the places at which advice is applied in aspects by defining joinpoints. When joinpoints are not precisely known by an aspect, a standard technique is to create an abstract pointcut and then to specialize the pointcut in derived aspects. In many cases, though, the specialization serves simply to identify those classes into which an aspect should be woven. For example, a trace aspect that logs a message whenever a method is entered or exited may specify before and after advice around method calls, but the classes to which it is applied may vary. If the classes to which the advice is applied are arbitrary, each particular class's name needs to be hardcoded in the aspect's pointcut. As indicated above, aspect inheritance could be used to help decouple the pointcut locations by creating a derived aspect each time a new location is discovered. This solution can become cumbersome when the number of aspects that need to be created becomes large.

An alternative solution used in the Interface Tag pattern is to create an empty interface class or a *tag* and use the tag to mark every class that should be affected by the aspect's advice. For example, in the trace aspect example, a Traceable interface could be created and any class that would like to be traced need only implement the Traceable interface. The trace aspect itself would only need to specify that its advice apply to all classes that implement Traceable to work.

In this example, the tracing concern is still encapsulated in the tracing aspect. However, the pointcut to which the tracing concern applies has been decoupled from the aspect. As classes are added to the system, the decision whether or not the tracing aspect should be applied can be made. Note that it is also possible that the tracing aspect is not enabled or has a stricter pointcut that may not apply to all classes that request it. In cases where the aspect is not applied, the presence of the interface tag results does not add any runtime overhead.

Applicability

Use the Interface Tag pattern when

1. An aspect's advice applies to arbitrary classes that are difficult or impossible to categorize in a central location.
2. The class intends to have advice applied to it that is consistent with the tag.
3. Knowledge of classes to which a pointcut applies breaks the encapsulation boundaries of what an aspect should know.

Structure

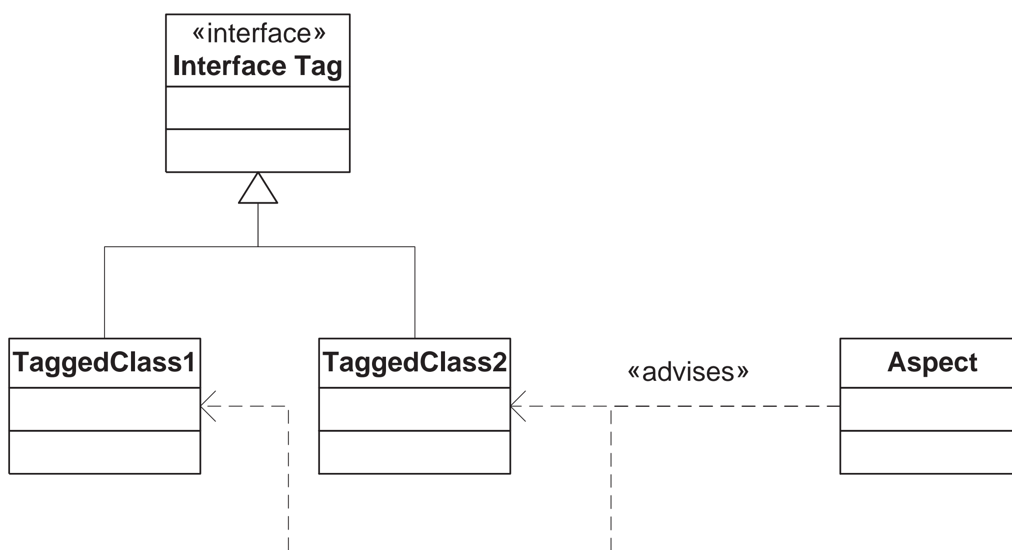

Figure 6.5: Interface Tag pattern structure.

Participants

Interface Tag: defines an empty interface that is implemented by a class to *tag* it.

TaggedClass1 and TaggedClass2: implement the Interface Tag interface.

Aspect: implements advice that affects classes that implement the Interface Tag interface.

Collaborations

The Aspect advises joinpoints within the tagged classes.

Consequences

The Interface Tag pattern has the following consequences:

1. *The Pointcut is more general.* By using the Interface Tag pattern, the pointcut in the aspect can be specified without specifically referencing target classes. This allows new classes to be added without revising the pointcut.

2. *The Aspect is more reusable.* Since the aspect does not contain references to specific classes, it is no longer coupled tightly with the code that it advises.

3. *Standard aspect semantics are inverted.* Normally, aspects apply their advice to target pointcuts that they specify. By using the Interface Tag pattern, classes request that aspects be applied to them.

Implementation

Consider the following issues when implementing the Interface Tag pattern:

1. *Tag all application classes.* It is easy to miss tagging some classes such as inner classes in Java. To be sure that classes are not missed, it may be possible to write a *tagging* aspect that tags all affected classes in a particular set of source files. The motivation for this pattern precludes tagging all classes in one aspect, but it may be possible to localize the tagging.

2. *Take advantage of classes that act like tags.* In some cases, classes may already exist that *tag* other classes. This obviates the need to define a specific Interface Tag.

Known Uses

The Interface Tag pattern is used identify classes and aspects in FACET that should be upgraded when unrelated features are included in the system. For this purpose, it is actually used twice. The first usage is to mark a class as upgradable by implementing the Upgradeable interface, and the second is to mark which features that class knows about. This latter marking is done by implementing an interface defined by each feature. Additionally, if a feature depends upon another feature, its interface will, in turn, extend that feature. Chapter 5 describes the Upgradeable interface in detail and provides more information on the automated upgrading of unit tests.

Related Patterns

The intent behind the Interface Tag pattern is similar to that used in Java to mark classes that can have their state *serialized* to or from a stream. Such classes are identified to the JVM by implementing the java.io.Serializable empty interface. Another such interface in Java is java.lang.Cloneable.

Chapter 7

Experimental Results

By enabling the user to select only those features that are necessary, the Framework for Aspect Composition for an EvenT channel (FACET) enables both code footprint and performance advantages over traditional middleware implementations. This chapter quantifies those benefits by using statistics generated during the compilation and testing of all FACET combinations.

7.1 Footprint

One method for measuring the footprint size of a Java application is to sum the size of all of the .class files that are loaded. Embedded systems that use Java interpreters or just-in-time compilers could use this metric to size an application ROM and to a lesser extent, the amount of RAM needed. By default, the AspectJ compiler includes debugging metadata in each .class file. The Jopt [31] .class file optimizer was used to strip the .class files of this information, remove unused constant pool entries, and perform minor optimizations on the generated bytecodes. After these optimizations, the .class file sizes are believed to be very close to the lower bound of the amount of information needed to use the Java code and data that the class files contain.

Another method consists of generating native code, as with the GNU Java compiler GCJ [49]. The size of the resulting executable image can then be measured, and since it contains only native code, it is more suitable for comparisons with C and C++ code. Moreover, embedded realtime applications are likely to precompile to native code for execution predictability. An overall observation is that the GCJ

produced object[1] files were generally larger than their corresponding .class files. This is commensurate with the design of .class files to be small—to speed transmission over networks.

Without any features enabled, the base FACET event channel code consists of only 110,125 bytes of .class files or 162,840 bytes of code and initialized data in GCJ-produced object files. Note that this measurement and the others that follow do not include the unit tests that are associated with FACET and its features. These are compiled together so that correct operation can be verified, but would not be present in real applications. At the other extreme, one of the heaviest FACET configurations consists of 470,133 bytes of .class files and 572,646 bytes of GCJ produced object files.

7.1.1 Quantifying the Footprint Increase of Individual Features

Studies that quantify the effect of features on footprint are typically difficult to find. The ability of FACET to test and gather statistics automatically on all viable combinations of features, however, makes this information easily obtainable. Figure 7.1 shows the average number of bytes added to the total footprint of the FACET middleware library for individual features or indivisible sets of features. Note that it may not be possible to determine the size of every feature since some are abstract. Figure 7.2 provides the short names for each set of features displayed in the chart. Appendix B describes each of these features in further detail.

To calculate the footprint overhead imposed by a feature or indivisible set of features, pairs of FACET configurations were compared that differed only by the feature set of interest. Figure 7.1 shows the results of averaging over all possible pairs for each feature set. Depending on where the feature set appears in the feature dependence graph, there may have been over a thousand combinations that were compared to determine the average. Besides the tracing feature, most features show little variation between combinations. This indicates that most features augment the base consistently. The main exception to this is the tracing feature, and its effect will be discussed in the following sections.

[1]Footprint size measurements were taken using the Unix *size* command. This command displays the actual text, data and bss sizes without including debug and other irrelevant information.

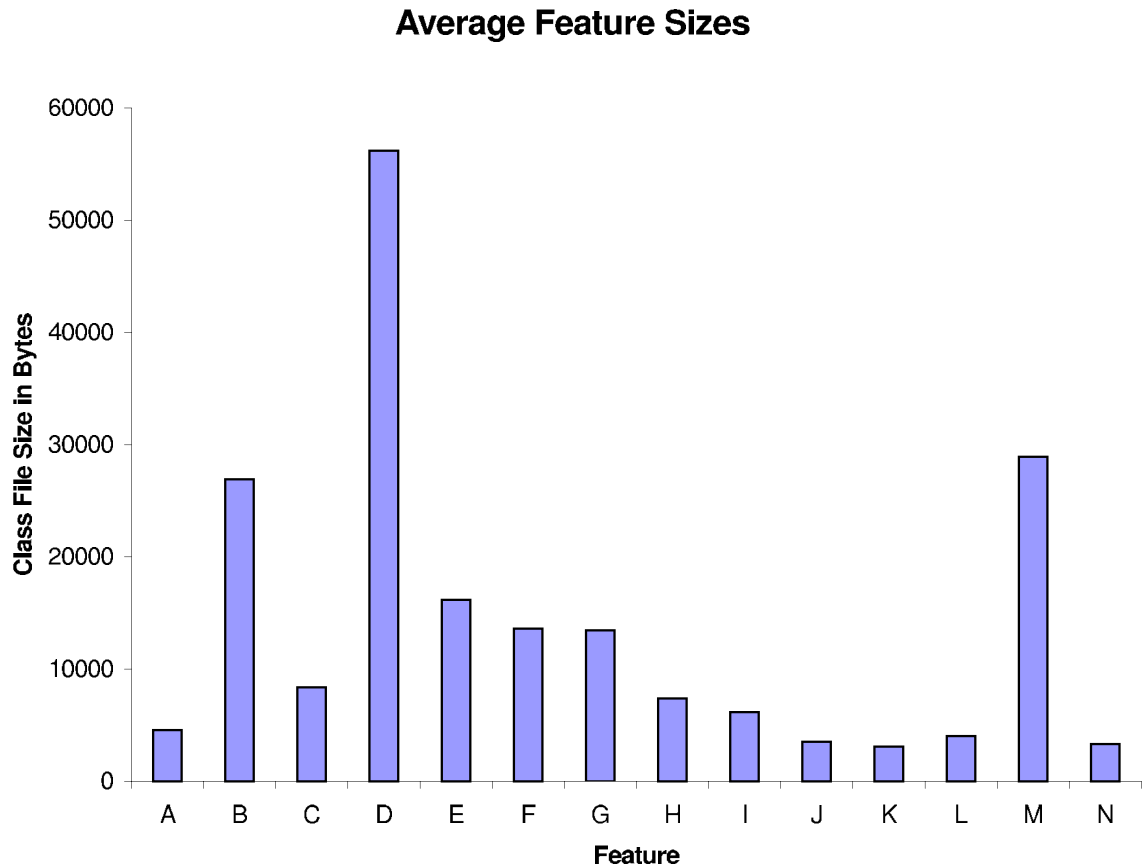

Figure 7.1: Feature set sizes.

Feature Set	Description
A	Supplier Dispatch
B	Correlation Filter
C	Event Sets
D	Event Pull
E	Event Struct and Body Octet Seq
F	Event Struct and Body Any
G	Event Struct and Body String
H	Event the Common Object Request Broker Architecture (CORBA) Any
I	Event Header
J	Event Type
K	Timestamp
L	Time-To-Live
M	Profiling Support
N	Basic Counters

Figure 7.2: Feature sets.

From Figure 7.1, it can be seen that the event pull feature and the event correlation feature contribute the most to the size of the footprint. This is not surprising since both features add significant functionality throughout the event channel. Specifically, the event pull feature adds several new Interface Definition Language (IDL) interfaces, event-buffering code for consumers, and event-polling code to obtain new events from suppliers. The new IDL interfaces tend to produce significant amounts of code in the generated stubs, skeletons, and helper classes. The event correlation feature makes few additions to the IDL interfaces, but it contains many classes that represent and check sequences of events.

When comparing footprint contributions of features, it is important to include code from dependent features. For example, based on the feature dependence graph, the event correlation feature cannot be included in isolation: one must therefore include, at a minimum, support for structured events, an event header, an event type field, and event set support.

Figure 7.3 and Figure 7.4 show the measurements for the additional footprint added when features or feature sets (containing no more than one concrete feature) are enabled. All measurements are in bytes. The total number of measurements used to determine the results in each row is denoted in the last column. The total possible is the number of pairs of FACET combinations that differ by only the specified feature for all tested combinations of the event channel. Again, the tracing feature has been left out of these measurements, since it significantly skews the results and would only be used for debugging in practice.

7.1.2 Footprint Sizes for Common Configurations

In the end, the combined footprint of the desired feature set is what is important to an embedded middleware user. Indeed, FACET event channels can vary in size by a factor of four depending on the selected features. Based on feedback from several developers in the The the ADAPTIVE Communication Environment (ACE) Object Request Broker (ORB) (TAO) user community who are using event channels in their applications, the following were identified as interesting configurations:

1. *Configuration 0 (Base):* Although the applications requested by developers all required more functionality than the base, it is useful in that it is a lower bound on the footprint. Note that all subsequent tests use the full functionality provided by the base.

Feature Set	Minimum	Maximum	Mean	StdDev	Samples (Used/Total)
Supplier Dispatch	4559	4559	4559	0.0	384/384
Event Type Filter Correlation Filter Depend Consumer Qos	47140	48739	47968	615.8	192/192
Supplier Dispatch Event Type Filter Depend Consumer Qos	25253	25994	25623	274.6	384/384
Profiling Support	28927	28944	28931	7.3	766/766
Correlation Filter	26446	27304	26904	351.3	384/384
Event Sets	7952	8572	8370	110.4	762/762
Event Pull	55377	57058	56192	518.1	1146/1146
Event Struct Event Type Mutex Body String	13463	13469	13465	2.8	3/3
Event Struct Event Type Mutex Body Any	13610	13616	13612	2.8	3/3
Event Struct Event Type Mutex Body Octet Seq	16170	16176	16172	2.8	3/3
Event CORBA Any Event Type Mutex	7392	7398	7394	2.8	3/3
Event Type	3398	3689	3532	85.4	288/288
Body String	2729	3060	2910	98.3	1140/1140
Body Octet Seq	5245	5967	5436	112.5	1140/1140
Body Any	2721	3213	2861	86.8	1140/1140
Time-To-Live	3934	4079	4034	43.0	1056/1056
Time-To-Live Event Header	9961	10584	10148	141.9	84/84
Event Type Event Header	9425	10224	9694	188.9	84/84
Event Header Timestamp	8940	9570	9130	144.0	84/84
Timestamp	2979	3152	3093	59.9	1056/1056
Consumer Filtering	2881	2881	2881	0.0	768/768
Basic Counters	3220	3395	3336	82.5	766/766

Figure 7.3: Class file measurements for FACET feature sets.

Feature Set	Minimum	Maximum	Mean	StdDev	Samples (Used/Total)
Event Channel Tracing	0	0	0	0.0	0/0
Supplier Dispatch	6172	6188	6178	6.1	384/384
Event Type Filter Correlation Filter Depend Consumer Qos	60812	63046	61918	628.6	192/192
Supplier Dispatch Event Type Filter Depend Consumer Qos	32040	34186	33104	618.9	384/384
Profiling Support	31936	34692	34679	99.4	766/766
Correlation Filter	34884	35044	34967	43.2	384/384
Event Sets	12080	14028	13175	400.0	762/762
Event Pull	69052	69840	69546	287.1	1146/1146
Event Struct Event Type Mutex Body String	13752	16560	15605	1310.7	3/3
Event Struct Event Type Mutex Body Any	13812	16620	15665	1310.7	3/3
Event Struct Event Type Mutex Body Octet Seq	17160	19968	19013	1310.7	3/3
Event CORBA Any Event Type Mutex	5960	8784	7820	1315.5	3/3
Event Type	4746	5362	4996	196.6	288/288
Body String	3650	4134	3895	131.9	1140/1140
Body Octet Seq	7062	7704	7434	209.0	1140/1140
Body Any	3724	4206	3966	135.5	1140/1140
Time-To-Live	5194	6484	6042	410.6	1056/1056
Time-To-Live Event Header	12898	13616	13259	306.9	84/84
Event Type Event Header	12394	13228	12809	354.6	84/84
Event Header Timestamp	11562	12286	11926	309.9	84/84
Timestamp	3858	5144	4698	415.3	1056/1056
Consumer Filtering	3308	3376	3339	26.0	768/768
Basic Counters	4392	4480	4445	34.3	766/766

Figure 7.4: GCJ object file measurements for FACET feature sets.

2. *Configuration 1:* Several developers only needed configurations similar to the standard CORBA COS Event Service specification. This configuration has CORBA Any payloads and does not support filtering. For these developers, the pull interfaces were not used and were not included.

3. *Configuration 2:* This configuration is the same as the previous except with the tracing feature enabled.

4. *Configuration 3:* Structured events and event sets are enabled. This configuration also adds the time to live (TTL) field processing to eliminate loops created by federating event channels. This configuration is still minimal, however, and does not support any kind of event filtering.

5. *Configuration 4:* This configuration has support for dispatching events based on event type. It uses a CORBA octet sequence as the payload type and is a common optimization over using a CORBA Any. This configuration is similar to that used in the TAO Real-Time Event Channel (RTEC).

6. *Configuration 5:* This configuration adds support for the event pull interfaces to configuration 4 and uses a CORBA Any as the payload.

7. *Configuration 6:* This configuration enhances configuration 4 by replacing the simple event type dispatch feature with the event correlation feature. In the corresponding application, event timestamping information was also needed, but the event pull feature was not.

8. *Configuration 7:* This configuration represents one of the largest realistic configurations of FACET. It supports the pull interfaces, uses event correlation, and adds support for statistics collection and reporting. It uses structured events carrying CORBA Anypayloads and headers with all possible fields enabled.

9. *Configuration 8:* This configuration adds the tracing feature to configuration 7.

Figure 7.5 shows which specific features are enabled for each above configuration. (See Appendix B for more information on each feature.) Figure 7.6 compares the footprint size of the library for these important configurations.

Configuration	Event Channel Tracing	Profiling Support	Basic Counters	Event Type Mutex	Event Corba Any	Event Struct	Event Sets	Body Octet Seq	Body Any	Body String	Event Pull	Event Header	TTL	Timestamp	Event Type	Consumer QoS	Depend	Event Type Filter	Correlation Filter	Supplier Dispatch	Consumer Filtering
0																					
1				X	X																
2	X			X	X																
3				X		X	X		X			X	X								
4				X		X		X				X			X	X	X	X		X	
5				X		X	X		X		X	X	X		X	X	X	X		X	
6				X		X	X	X	X			X		X	X	X	X	X	X		
7		X	X	X		X	X		X		X		X	X	X	X	X	X	X	X	
8	X	X	X	X		X	X		X		X		X	X	X	X	X	X	X	X	

Figure 7.5: Enabled features under various configurations.

7.1.3 Impact of External Libraries

Features can have an even more substantial impact on the overall footprint when they depend on auxiliary libraries to provide some functionality. An example of this is the tracing feature, since it pulls in the *log4j* logging libraries [3] that require an additional 290 kilobytes of .class files. In non-embedded Java applications, the Java class loader can limit the amount of code and data in memory by dynamically loading only what is needed. On the other hand, embedded applications often require that all possibly executed code be linked or packaged together prior to runtime, so that such code can be deployed in ROM or some other local memory device. Therefore, a middleware user also needs to consider external libraries that are referenced as byproducts of enabling features.

In addition to referencing other libraries, features may make more or less use of libraries required in the base. This becomes apparent in FACET's use of the JacORB [11] CORBA ORB. For example, CORBA Anys require support from the ORB and additional code to be generated from IDL files to marshal and demarshal Any variables. Since FACET can be configured to avoid using CORBA Anys, it would be desirable to remove all Any support from the ORB to reduce code footprint. As the ORB libraries contribute a substantial amount of code to the end application, it is desirable to trim other functionality as well. This is not possible in the JacORB

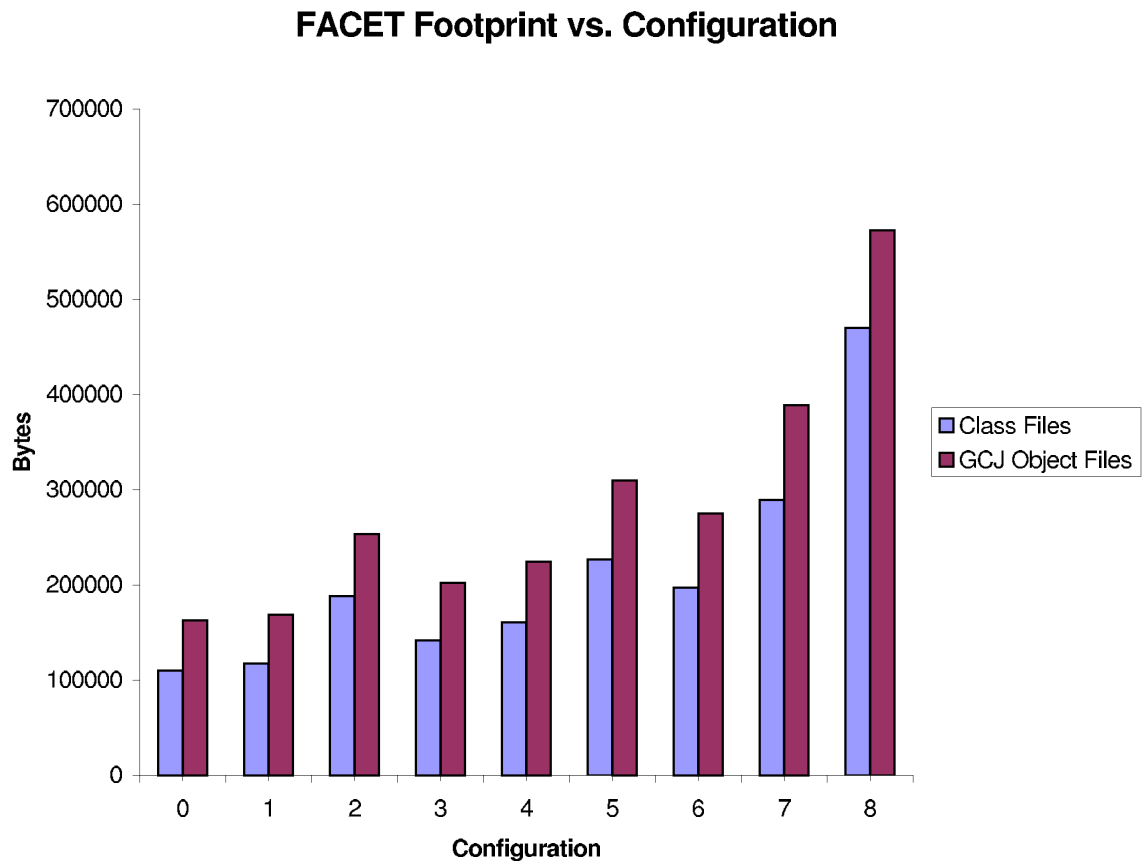

Figure 7.6: FACET Library sizes under different configurations.

implementation, however, since the degree to which these concerns can be separated from the ORB is not as significant as what can be accomplished using Aspect-Oriented Programming (AOP) techniques—as in FACET.

Increases in external library usage from various feature combinations can be determined by the following:

$$m = s - t - f$$

where m is the footprint increase attributed to additional code from external libraries, s is the size of a simple executable linked against a FACET configuration, t is the size of a trivial executable, and f is the size attributed to code in the FACET library. For the following measurements, the trivial executable is what is created by GCJ using an empty main function and linking against JacORB. Linking against JacORB was performed, since it is a large library that is sufficiently entangled so that the most casual reference (say, to initialize the library) causes nearly the whole library to be linked. Even though JacORB is an external library, including it as such distorts variations in the usage of other external libraries that can be partially linked. The design of the ZEN ORB [13] appears likely to be able to mitigate many of these issues with JacORB.

Figure 7.7 shows external library usage based on the configurations defined in Section 7.1.2. As described earlier, the most substantial increases in external library code are seen when the tracing feature is enabled. Other increases result from adding reference to more classes from Doug Lea's util.concurrency library [28].

7.2 Performance

Performance measurements were attained by running an event throughput test. By marking the test as *upgradeable*, feature upgraders could automatically augment the test to take advantage of their features. In the test, no data is passed from the suppliers to the consumers so that the base configuration could be supported. In the cases where the upgraders added payload fields, those fields would be initialized to empty or zero length values. For example, when strings are used as payloads, they are initialized to the empty string.

All measurements were performed on a dual 933Mhz Pentium III workstation with 512MB of RAM running an SMP version of the Linux 2.4.9 kernel. The Sun JDK 1.3.1_01 virtual machine ran the throughput tests using native threads, and the

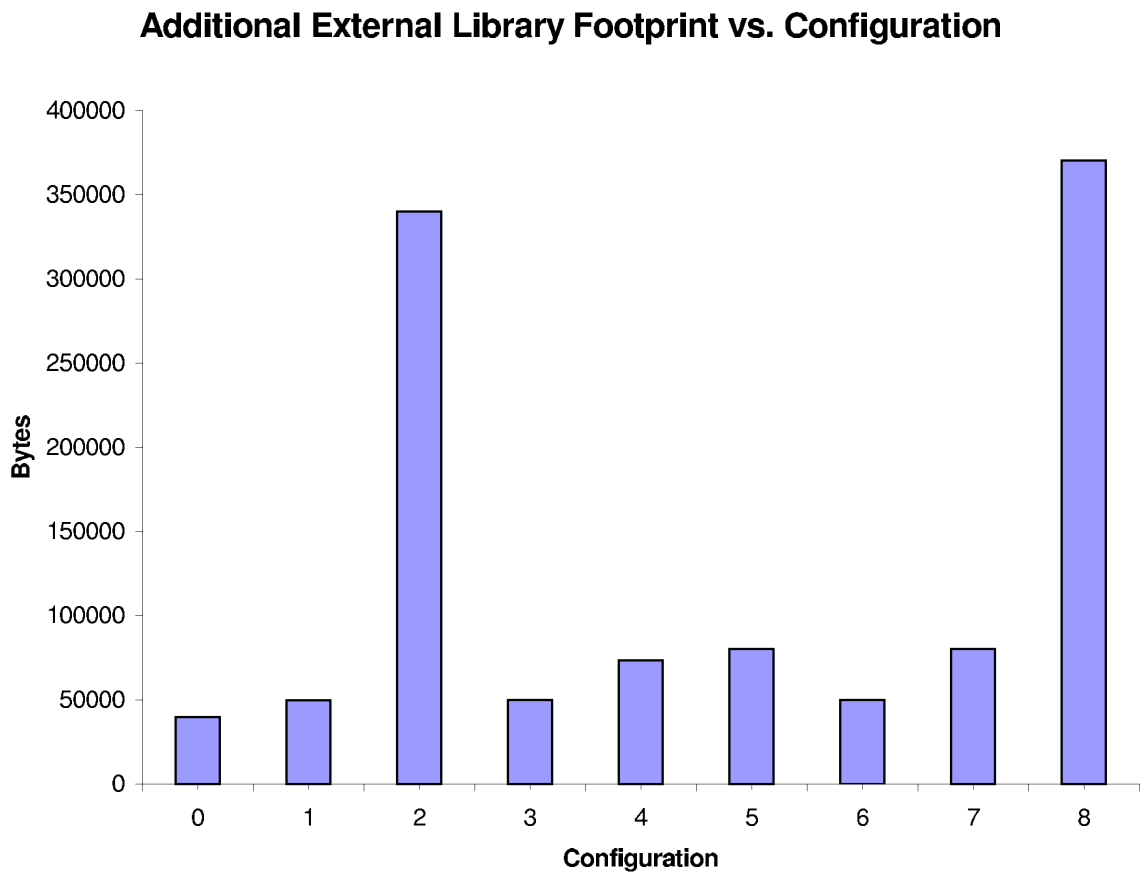

Figure 7.7: Increase in external library under various configurations.

CORBA ORB used was JacORB 1.3.30. If the tracing feature was enabled, its output was configured to go to a file on the local disk rather than to write to the terminal.

Overall, the base FACET configuration performed the fastest with an average throughput of 1330 events/second. Ignoring the tracing feature, one of the most fully featured configurations was over 20% slower at 1041 events/second. The tracing feature had a significant impact on the performance of all configurations. In particular, it reduced the performance of the base configuration to 555 events/second and the fully featured configuration to 268 events/second. The following sections present the throughput results in detail.

7.2.1 Performance Effect of Enabling Features

Just as for the footprint measurements, the throughput test was run on every configuration so that the effect of each feature on the performance could be determined. The tracing feature has been ignored as its effect is so severe that it distorts the results. Figure 7.8 shows the results on many of the more important features. As before, Figure 7.2 provides a short description of each set of features displayed in the chart.

From the figure, most of the features degrade the throughput only slightly, if at all. Of these, event pull support (D), event set support (C), and statistics infrastructure (M) do not add any code to the critical path of the throughput test. The filtering and correlation features (A and B) both degrade performance by less that 1% on average. Interestingly, accessing the current time to mark a timestamp reduces the performance by almost 2%. But by far the worst effect on performance is seen when enabling event payloads. Of these, using strings (G) is slightly faster than using octet sequences (E) – possibly due to internal Java support for strings. Using CORBA Any types is the worst and reduces performance by around 15%.

7.2.2 Performance of Common Configurations

Figure 7.10 shows the performance results for the common configurations identified in Section 7.1.2. The throughput of each configuration is scaled to the performance of the base configuration (Configuration 0). The results shown here support what would be found by combining each feature's performance degradation from the previous section to arrive at the indicated configuration. Note that the tracing feature has a

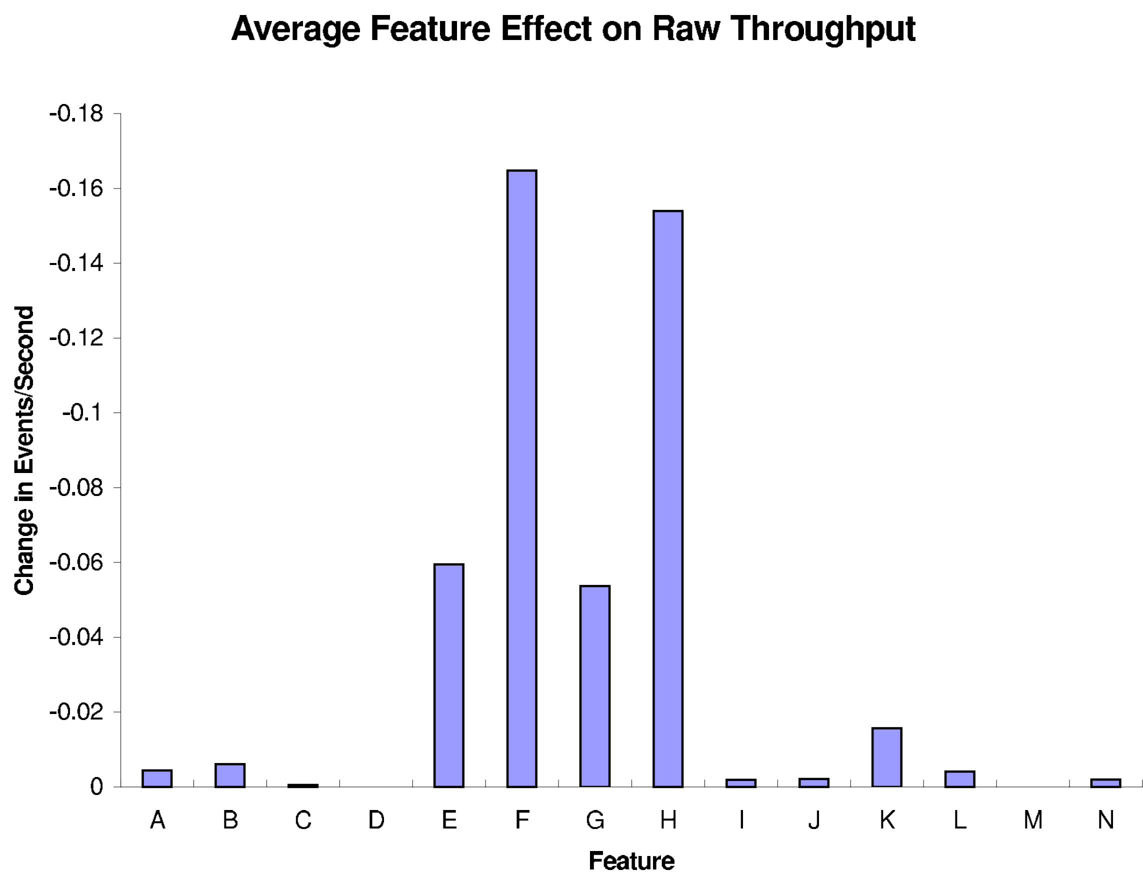

Figure 7.8: Feature impact on throughput.

Feature Set	Minimum	Maximum	Mean	StdDev	Samples (Used/Total)
Supplier Dispatch	-0.073	0.054	-0.004	0.020	384/384
Event Type Filter Correlation Filter Depend Consumer Qos	-0.064	0.063	-0.010	0.020	192/192
Supplier Dispatch Event Type Filter Depend Consumer Qos	-0.071	0.042	-0.008	0.019	384/384
Profiling Support	-0.070	0.065	0.001	0.020	766/766
Correlation Filter	-0.067	0.061	-0.006	0.020	384/384
Event Sets	-0.057	0.057	-0.001	0.019	762/762
Event Pull	-0.068	0.070	0.000	0.020	1146/1146
Event Struct Event Type Mutex Body String	-0.079	-0.031	-0.054	0.019	3/3
Event Struct Event Type Mutex Body Any	-0.189	-0.146	-0.165	0.018	3/3
Event Struct Event Type Mutex Body Octet Seq	-0.068	-0.050	-0.059	0.007	3/3
Event CORBA Any Event Type Mutex	-0.178	-0.123	-0.154	0.023	3/3
Event Type	-0.051	0.050	-0.002	0.018	288/288
Body String	-0.071	0.069	-0.005	0.020	1140/1140
Body Octet Seq	-0.061	0.068	-0.006	0.019	1140/1140
Body Any	-0.185	-0.069	-0.132	0.017	1140/1140
Time-To-Live	-0.078	0.069	-0.004	0.020	1056/1056
Time-To-Live Event Header	-0.052	0.070	-0.006	0.023	84/84
Event Type Event Header	-0.051	0.039	-0.004	0.021	84/84
Event Header Timestamp	-0.058	0.053	-0.015	0.020	84/84
Timestamp	-0.084	0.056	-0.016	0.019	1056/1056
Consumer Filtering	-0.071	0.076	-0.002	0.020	768/768
Basic Counters	-0.072	0.064	-0.002	0.020	766/766

Figure 7.9: Throughput degradation measurements for FACET feature sets.

significant impact on configurations 2 and 8. Figure 7.11 shows the actual throughput measurements for each of the configurations.

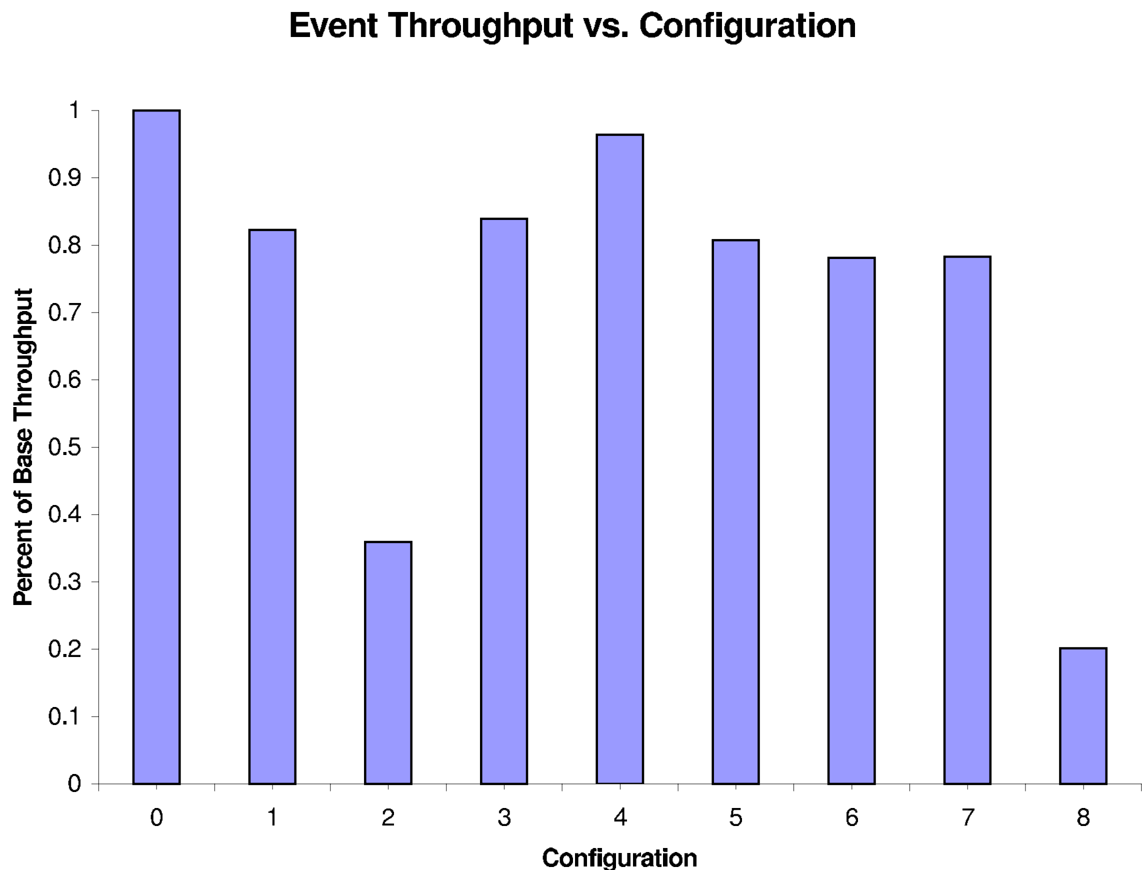

Figure 7.10: Throughput results normalized to the base throughput.

7.3 Savings from Using Aspects

By using aspects to weave features together, FACET does not require the programming infrastructure to support varying functionality that traditional middleware needs. This includes `if` statements to choose alternate paths, virtual function calls to strategized methods, and abstract factories to select functionality at runtime. This entire infrastructure impacts the performance and code size of the middleware both when extended features are enabled *and* when they are not included.

Determining the overhead saved by using AOP instead of traditional techniques in FACET is not straightforward. At a minimum, several features introduce fields to

Configuration	Throughput (Events/Second)
0	1330
1	1094
2	478
3	1116
4	1282
5	1074
6	1039
7	1041
8	268

Figure 7.11: Measured throughput of common configurations.

existing data structures such as the Event structure. Any Java-only implementation of FACET could not allow this flexibility.[2] Other features have mutually exclusive relationships that cannot be easily rewritten for the same functionality just using object-oriented techniques. Additionally, the flexibility of aspects to augment code directly at the appropriate interception points serves to minimize the commonality of interception points between aspects. If converted directly to an object-oriented program with similar flexibility, this would result in a high number of hooks to call extensions. An object-oriented designer would probably try to reduce the number of these hooks and find more commonalities between features to reduce the complexity of the resulting code.

In spite of these issues, some information about the overhead saved using aspects can be determined simply. In most Java programs that output trace messages, the code that generates those messages is surrounded by `if` tests to check if tracing has been enabled. For performance reasons, these `if` tests are in the client code to avoid any unnecessary method calls. By using aspects, all of these `if` tests can be eliminated along with all of the tracing code if tracing is not desired. To measure the overhead of having `if` tests, the standard FACET tracing feature was duplicated, and `if` blocks were added around the calls to the tracing advice by using the AspectJ `if` pointcut designator.

Figure 7.12 shows the results of adding `if` tests around calls to tracing advice and how it compares to the standard FACET Tracing feature. One of the heaviest

[2]Languages such as C and C++ that support preprocessor macros could allow for this flexibility at a major cost to code readability.

FACET configurations (Configuration 7 from Section 7.1.2) was used for all the measurements, since it has the most method calls that get traced. The main disadvantage of using the `if` statements is that the tracing code is included even when it isn't used. Additionally, the `if` statements slightly impact (1-2%) the throughput of the channel both when logging is enabled and disabled.

Configuration	Sum of Class File Sizes (Bytes)	Throughput (Events/Second)
FACET Tracing feature disabled	289,524	1055
FACET Tracing feature enabled	470,133	267
Tracing with *if* guards disabled	476,727	1029
Tracing with *if* guards enabled	476,727	264

Figure 7.12: Overhead of using *if* statements for tracing rather than aspects.

Chapter 8

Conclusions and Future Work

As embedded software becomes more complex, it becomes increasingly desirable to use middleware and, in particular, Distributed Object Computing (DOC) middleware in distributed embedded applications. Two major impediments to using middleware frameworks such as the ADAPTIVE Communication Environment (ACE) and The ACE Object Request Broker (ORB) (TAO) are their footprint size and the inability to subset them enough to fit on platforms with limited program storage. Unfortunately, existing object-oriented techniques to subset middleware are time consuming and can make existing code more complex.

In this thesis, we have developed a novel approach to constructing middleware by using Aspect-Oriented Programming (AOP) techniques. By designing an essential base implementation and using *aspects* to encapsulate optional features, the middleware user now has the ability to select only those features that are truly needed. This has distinct advantages in that the resulting middleware contains very little code bloat for unused features—they are simply not compiled. Additionally, by using aspects, the hooks, strategies, registries, and other infrastructure needed to support subsetting object oriented middleware are no longer needed. This simplifies the readability of both the feature and the base code.

To research the feasability of developing middleware using AOP, we built the Framework for Aspect Composition for an EvenT channel (FACET), a the Common Object Request Broker Architecture (CORBA) event channel modelled after the Object Management Group (OMG) Event and Notification Services and the TAO Real-time Event Channel. The base FACET implementation is essentially an interrupt service that can notify consumers when events happen but not pass any information.

Features are then used to send payloads, provide correllation and filtering, support statistics collection, provide *pull* interfaces, and more.

Managing the many features in FACET is itself an issue, since dependences between features make some event channel combinations invalid. A framework was developed to describe the characteristics of features within the system and their relationships to other features. In FACET, every feature registers its characteristics and dependences with the FeatureRegistry. With this information, the FeatureRegistry can be used to validate configurations and automatically select missing dependent features.

Managing features alone, however, is not enough for developing highly customizable middleware. Especially for high reliability environments, verification that selected configurations actually do work is also needed. In FACET, this is provided by including a test framework that is used by every feature. For any particular combination of features, the build system can run all relevant unit tests. Additionally, FACET can itself verify that every possible combination of features and associated unit tests compiles and runs by using information from the FeatureRegistry.

To aid the development of future middleware that uses AOP, several design patterns were identified and documented that proved very useful in the development of FACET. These include lessons learned when extending Application Programming Interface (API) calls to contain new parameters, encapsulating and exposing joinpoints and advice, and using aspects to augment arbitrary code.

Lastly, footprint and performance measurements were taken that quantify the advantages of using AOP to selectively enable and disable middleware functionality. These measurements show that disabling complex unneeded middleware features can significantly improve middleware's applicability to more constrained environments. Also, performance and footprint results were shown for several real configurations of event channels described by members of the TAO user community. By modeling the dependences between features in FACET, we have built, tested and gathered measurements for all viable configurations. In this paper, we have presented these measurements for event service configurations presently in use by members of the TAO user community. Of note, a variation of over four times was seen between the most and least feature-rich configurations.

Many areas exist for future work. First, for highly embedded environments, using CORBA is unnecessary and contributes too much overhead. It would be desirable to encapsulate the CORBA aspects of FACET into one or more features. Many

impediments make this difficult, though, since the CORBA Interface Definition Language (IDL) compiler generates stub and skeleton code that have different creation, use, and destruction semantics from standard Java classes. Once this is acheived, it is desirable to support other types of distributed middleware such as Java RMI [50]. An additional advantage of having this ability is that the event channel filtering, dispatching, and statistics collection code can be reused in widely different environments. Such a capability is currently not possible, since applications tend to be tightly coupled with their choice of distributed middleware.

Another area of research is the integration the FACET event dispatching mechanisms with the Real-Time Specification for Java^{TM} (RTSJ) [8]. This would allow FACET to provide realtime guarantees to suppliers and consumers. By using FACET features to select the degree to which realtime assurances are important to an application, programmers could trade off the performance advantages of soft realtime systems with the absolute guarantees needed in hard realtime systems.

One of the issues found when using FACET is that currently a general shared library cannot be created that supports all possible configurations. For example, it would be ideal if an application could specify its required features, and an aspect-aware linker (or dynamic library loader) could weave in the features to create the desired event channel. Currently, a separate library needs to be created for every desired configuration, and an application needs to link against the library that supplies the right features. In FACET, since every configuration is in the same package, only one configuration can be in use at a time. Simply creating a different package for every configuration is impractical due to the large number of configurations and the space required.

Also of interest is applying the experience of developing FACET using AspectJ and Java to C++ by using AspectC++ [37]. This process will likely identify many other challenges to developing highly subsettable middleware using AOP, since more low-level language details will have to be addressed when writing features. It also broadens the appeal of FACET, due to the number of embedded environments that already using C++.

Finally, related research within the DOC Group will use many of the lessons learned from using AOP in FACET to build a highly subsettable ORB. The project investigates combining AOP techniques with Generic Programming techniques to separate feature concerns further and more cleanly than using either technique alone. An example of where this might be useful in FACET is to decompose the filtering

features. Currently FACET requires a feature to add a filterable field (i.e. the event type field) and a filter that checks that field. A Generic Programming technique to avoid creating separate filters would be to make the filtering feature parameterizable.

Appendix A

Glossary

advice: Code contained in an aspect that is executed at the locations of its associated joinpoints.

aspect: An *aspect* is a specification of a cross-cutting concern.

base: As used in this thesis, the *base* refers to the core set of code that supports a fundemental level of functionality. This functionality is indivisible, and features are used to extend and enhance it.

cflow: A *cflow* or control flow specification describes an execution path joinpoint. Variables and data available at both the beginning and end of the execution path can be used in advice.

feature: A *feature* is a cohesive set of code (classes and aspects) that provides a specific functional or structural enhancement to the base.

introduction: An *introduction* statically adds member variables or methods to existing classes and interfaces.

joinpoint: A *joinpoint* is a well-defined point in a program such as a invoking a method or accessing a class member variable.

pointcut: A *pointcut* is an expression containing joinpoints that can identify a set of well-defined points.

Appendix B

FACET Features

Figure Figure B.1 shows the dependence relationships between the features implement in FACET. This appendix provides a brief description of each of these features.

Unless noted otherwise, all classes and interfaces are specified relative to the edu.wustl.doc.facet package.

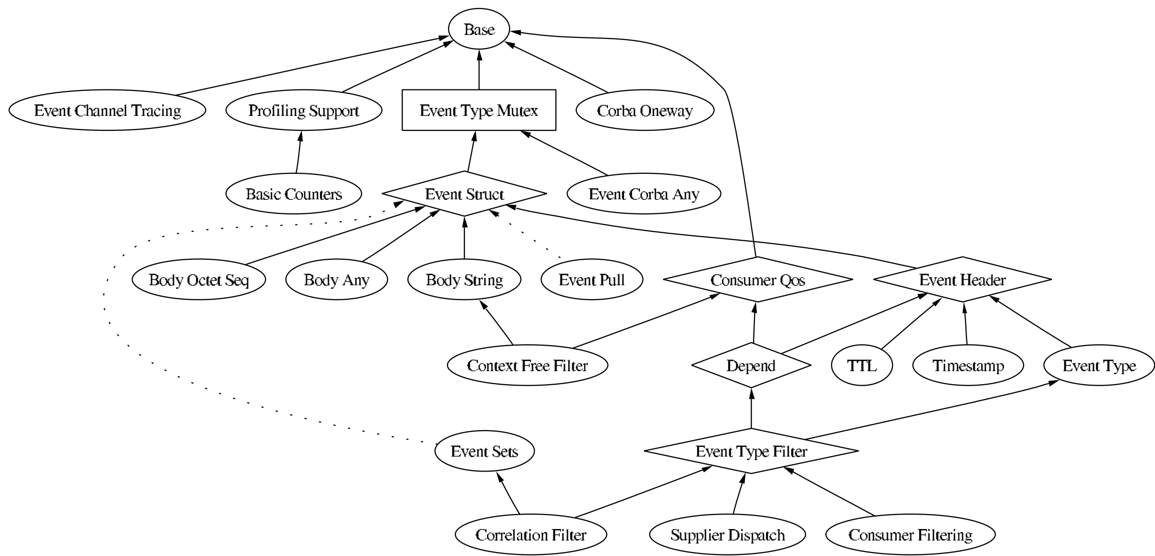

Figure B.1: Feature Dependence Graph.

B.1 Basic Counters

Feature interface: basic_counters.BasicCountersFeature

Type: Concrete

Depends: Profiling Support

Contains: None

Description: The Basic Counters feature registers event counters with the Profiling Support feature to measure statistics such as the number of events that pass through the event channel. It uses aspects to intercept internal event channel calls and increment counter.

B.2 Body Any

Feature interface: eventbody_any.CorbaEventBodyAnyFeature

Type: Concrete

Depends: Event Struct

Contains: None

Description: This feature introduces a CORBA Any payload to the Event structure. Any payloads automatically carry type information, but are generally not as fast as other payload types to marshal and demarshal.

B.3 Body Octet Seq

Feature interface: eventbody_octetseq.CorbaEventBodyOctetSeqFeature

Type: Concrete

Depends: Event Struct

Contains: None

Description: This feature introduces a CORBA Octet Sequence payload to the Event structure. This is a common optimization for event channels to pass large amounts of data.

B.4 Body String

Feature interface: eventbody_string.CorbaEventBodyStringFeature

Type: Concrete

Depends: Event Struct

Contains: None

Description: This feature introduces a CORBA String payload to the Eventstructure and is convenient for event channel use cases where all information can be contained in a string. The Context Free Filter feature can be used to filter strings based on particular patterns.

B.5 Consumer Filtering

Feature interface: consumer_dispatch.CorbaConsumerDispatchFeature

Type: Concrete

Depends: Event Type Filter

Contains: None

Description: This feature adds support for filtering events for consumers. The filtering occurs after event dispatching has occurred. It is useful for environments where the additional overhead to maintain dispatch tables is large or almost all consumers receive all events. In most cases, the Supplier Dispatch feature will be preferable to this one.

B.6 Consumer Qos

Feature interface: consumer_qos.CorbaConsumerQosFeature

Type: Abstract

Depends: Base

Contains: None

Description: This feature provides the IDL interfaces and basic infrastructure in FACET to allow consumers to register quality of service requirements with the event channel.

B.7 Context Free Filter

Feature interface: cffilter.ContextFreeFilterFeature

Type: Concrete

Depends: Consumer Qos and Body String

Contains: None

Description: This feature enables consumers to specify grammars that should be run on payloads of incoming events. Events are forward to a consumer only if the pattern matches. The filter uses an efficient parsing technique described in [26] that was extended and implemented in Java by Dan Rosenstein and further enhanced by Martin Linenweber.

B.8 CORBA Oneway

Feature interface: corba_oneway.CorbaOnewayFeature

Type: Concrete

Depends: Base

Contains: None

Description: This feature specifies that the event push methods should be marked as CORBA oneways. This can result in a performance improvement, since the sender no longer needs to wait for replies from the event channel or the consumer. Since CORBA oneways are not guaranteed to reach their destination, events can be dropped.

B.9 Correlation Filter

Feature interface: filter.CorrelationFilterFeature

Type: Concrete

Depends: Event Sets and Event Type Filter

Contains: None

Description: This features supports the creation of correlation filters so that consumers can specify that they should not be notified until a sequence of events is received. This feature may require the event channel to buffer many events, and the processing requirements are greater for it than other filtering features.

B.10 Depend

Feature interface: corba_depend.CorbaDependFeature

Type: Abstract

Depends: Consumer Qos and Event Struct

Contains: None

Description: This feature adds the infrastructure to allow consumers to specify their dependences upon events. It is used by most of the filtering features to specify their grammars.

B.11 Event Channel Tracing

Feature interface: tracing.EventChannelTraceFeature

Type: Concrete

Depends: Base

Contains: None

Description: This feature adds logging code to allow one to trace and debug all calls within the base and enabled features in FACET. The logging facility uses the *log4j* library to support sending log events to a variety of destinations.

B.12 Event CORBA Any

Feature interface: event_any.CorbaEventAnyFeature

Type: Concrete

Depends: Event Type Mutex

Contains: None

Description: This feature specifies that the event push interfaces should use CORBA Any data types to pass events. By enabling this feature, FACET's API is very similar to the API of the CORBA Event Service.

B.13 Event Header

Feature interface: corba_eventheader.CorbaEventHeaderFeature

Type: Abstract

Depends: Event Struct

Contains: None

Description: This feature introduces a header field to the Event structure. The intention is that fields in the header are visible to the event channel. Fields not in the header are considered as payload and are generally opaque.

B.14 Event Pull

Feature interface: event_pull.CorbaEventPullFeature

Type: Concrete

Depends: Base

Contains: Event Struct

Description: This feature adds the *pull* style interfaces to suppliers and consumers. It also adds the implementation to support polling pull suppliers for events and queuing events to pull consumers.

B.15 Event Sets

Feature interface: corba_eventvec.CorbaEventVecFeature

Type: Concrete

Depends: Base

Contains: Event Struct

Description: This feature adds the capability for more than one event to be sent to the channel or to consumers simultaneously. This feature can be used to optimize event transmission by allowing for more events to be sent at a time. The Correlation Feature uses Event Sets to bundle the sequence of events that causes a match together for transport to the consumer.

B.16 Event Struct

Feature interface: corba_struct.CorbaEventStructFeature

Type: Abstract

Depends: Event Type Mutex

Contains: None

Description: This feature adds support for transporting Event structures to the supplier and consumer interfaces of the event channel and internally. This style of sending events is very similar to that used in the TAO Real-time Event Service and cannot be used simultaneously with the Event Corba Any feature.

B.17 Event Type

Feature interface: corba_eventtype.CorbaEventTypeFeature

Type: Concrete

Depends: Event Header

Contains: None

Description: This feature adds an event type field to the Event structure's header.

B.18 Event Type Filter

Feature interface: eventtype_filter.CorbaEventTypeFilterFeature

Type: Abstract

Depends: Event Type

Contains: None

Description: This feature contains common code used by all of the event type filtering features. It does not provide any useful functionality by itself.

B.19 Event Type Mutex

Feature interface: eventtype_mutex.EventTypeMutexFeature

Type: Mutual Exclusion

Depends: Base

Contains: None

Description: This mutual exclusion feature prevents the Event Struct feature and the Event Corba Any feature from being enabled simultaneously. Without it, it would be possible to produce event channels that do not compile.

B.20 Profiling Support

Feature interface: profiling_support.CorbaProfilingSupportFeature

Type: Concrete

Depends: Base

Contains: None

Description: This feature provides the interfaces and registration implementation for adding profiling counters to FACET. Although this feature is concrete, it does not provide any performance counters itself. In most configurations, another feature that provides counters would be enabled as well.

B.21 Supplier Dispatch

Feature interface: supplier_dispatch.CorbaSupplierDispatchFeature

Type: Concrete

Depends: Event Type Filter

Contains: None

Description: This feature adds the ability to dispach events to consumers. It is different from the Consumer Dispatch feature since the dispatching occurs when the event is received by the event channel. If there are many consumers or each consumer only receives a small subset of events, it can enable performance gains over the Consumer Dispatch feature.

B.22 Timestamp

Feature interface: corba_timestamp.CorbaTimestampFeature

Type: Concrete

Depends: Event Header

Contains: None

Description: This feature adds a timestamp field to the event header and adds code to mark this field with the current time when the event is received by the event channel.

B.23 Time-To-Live

Feature interface: corba_ttl.CorbaTtlFeature

Type: Concrete

Depends: Event Header

Contains: None

Description: This feature introduces a time to live (TTL) field to the event header and adds code to decrement the field at each event channel hop. If the TTL field is zero. then the Event is dropped. This feature is necessary if event channel loops are possible in federated configurations.

References

[1] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[2] Apache Software Foundation. Apache Ant. `http://jakarta.apache.org/ant/`.

[3] Apache Software Foundation. log4j. `http://jakarta.apache.org/log4j/`.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[5] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[6] Lodewijk M.J. Bergmans and Mehmet Aksit. Aspects crosscutting in layered middleware systems.

[7] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

[8] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[9] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1975.

[10] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[11] Gerald Brose, Nicolas Noffke, and Sebastian Müller. JacORB 1.4 Programming Guide. http://jacorb.inf.fu-berlin.de/ftp/doc/ProgrammingGuide_1.4.pdf, 2001.

[12] L. Carver and W. Griswold. Sorting out concerns, 1999.

[13] Center for Distributed Object Computing. The ZEN ORB. www.zen.uci.edu, University of California at Irvine.

[14] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[15] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using aop to improve os structure modularity. *Communications of the ACM*, 44(10):79–82, 2001.

[16] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[17] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[18] Douglas C. Schmidt. Successful Project Deployments of ACE and TAO. www.cs.wustl.edu/∼schmidt/TAO-users.html, Washington University.

[19] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[20] Erich Gamma and Kent Beck. JUnit. www.xProgramming.com/software.htm, 1999.

[21] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 1999.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[23] Pradeep Gore, Ron K. Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and Optimizing a Scalable CORBA Notification Service. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, pages 196–204, Snowbird, Utah, June 2001. ACM SIGPLAN.

[24] DOC Group. ACE Success Stories. `http://www.cs.wustl.edu/~schmidt/ACE-users.html`.

[25] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[26] Mahesh Jayaram and Ron Cytron. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, University of Arizona, Tucson, AZ, February 1996.

[27] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[28] Doug Lea. *Concurrent Java: Design Principles and Patterns, Second Edition.* Addison-Wesley, Reading, Massachusetts, 1999.

[29] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.

[30] Mark Lutz. *Programming Python.* O'Reilly, 2nd edition, 2001.

[31] Markus Jansen. Jopt. www-i2.informatik.rwth-aachen.de/∼markusj/jopt/, 2000.

[32] Matti A. Hiltunen and Richard D. Schlichting. The Cactus Approach to Building Configurable Middleware Services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, October 2000.

[33] J. P. Morgenthal. Microsoft COM+ Will Challenge Application Server Market. www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[34] Object Management Group. *Notification Service Specification*, OMG Document telecom/99-07-01 edition, July 1999.

[35] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, October 2000.

[36] Object Management Group. *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 edition, September 2001.

[37] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, February 2002.

[38] OMG. *CORBAServices: Common Object Services Specification, Revised Edition*. Object Management Group, 97-12-02 edition, November 1997.

[39] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2), March 2002.

[40] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, 1999.

[41] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[42] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, May 1999.

[43] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

[44] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[45] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*. Addison-Wesley, Boston, 2002.

[46] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[47] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[48] The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.

[49] The GCC Team. GCJ: The GNU Compiler for Java. `http://gcc.gnu.org/java/`.

[50] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4), November/December 1996.

Vita

Frank Hunleth

Date of Birth October 25, 1974

Place of Birth St. Louis, MO

Degrees B.S. Computer Science, 1997,
from Northwestern University.

Publications F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," in *The OOPSLA 2001 Separation of Concerns Workshop,* (Tampa Bay, FL), October 2001.

F. Hunleth and R. Cytron, "Footprint and Feature Management using Aspect-Oriented Programming Techniques," in *Proceedings of LCTES/SCOPES 2002,* (Berlin, Germany), ACM SIGPLAN, June 2002.

May 2002