Refactoring Middleware with Aspects

Charles Zhang and Hans-Arno Jacobsen, *Member*, *IEEE*

Abstract—Middleware platforms, such as Web services, J2EE, CORBA, and DCOM, have become increasingly popular during the last decade. They have been very successful in solving distributed computing problems for a large family of application domains. The architecture of middleware systems have gone through many significant cycles of evolution, both in terms of the completeness of functionality and the range of adoptions for different types of platforms. However, at the same time, it is getting increasingly difficult to achieve and to maintain a high level of adaptability and configurability because the structure of the middleware architecture is becoming overly complicated and rigid. We attribute that problem to the limitations of traditional software decomposition methods. Aspect-oriented programming, on the contrary, has introduced new design perspectives that permit the superimpositions of different abstraction models on top of one another. This is a very powerful technique for separating and simplifying design concerns. In our effort of applying principles of aspect orientation to the middleware architecture, we first pragmatically analyze the use of aspects in the middleware architecture. We then show that aspects are the correct remedy for the above outlined middleware problems by quantifying crosscutting concerns in the legacy implementations of several prominent middleware systems. Our aspect analysis results strongly indicate that modularity of middleware architecture is greatly hindered by the wide existence of tangled logic. To go one step further, we factor out a number of crosscutting concerns identified in the mining process, reimplement them as aspects, and superimpose them back into the refactored architecture. This allows us to use a set of software engineering metrics to quantify the refactorization in terms of changes in the structural complexity, modularity, and performance of the resulting system. This aspect-oriented refactoring proves that aspect orientation is capable of composing orthogonal design requirements. The final "woven" system is able to correctly provide both the fundamental functionality and the "aspectized" functionality with negligible overhead and an overall leaner architecture. Furthermore, the "aspectized" feature can be configured in and out during compile-time, which greatly enhances the configurability of the architecture.

Index Terms—Aspect-oriented programming, aspects, middleware, aspect analysis, refactoring.

1 Introduction

In recent years, middleware systems, such as Web services, .NET, J2EE, and CORBA, have been widely adopted, not only on traditional enterprise computing platforms, but also in a very large family of emerging application domains, such as control platforms, smart devices, and networking equipment. Many of these domains exhibit a broad spectrum of characteristics, such as specialized runtime requirements, real-time constraints, stringent resource requirements, high availability, and high performance.

The most prominent problem in today's middleware systems is that their architecture constantly struggles between generality and specialization. That is to say, on one hand, vendors want to support many application domains with their middleware products that provide a relatively complete set of features. However, these systems usually require large memory space and abundant computing resources. On the other hand, architects often want to optimize the architecture to support particular domains with specialized runtime requirements, such as real-time, embedding, and high availability. As a result of that, for the same technology, there often exists multiple specifications, various branches of code bases, and different implementations. Each requires a tremendous amount of effort in order to maintain the conformity of services, which middleware is supposed to provide.

Recent approaches, such as OpenCOM [1] and DynamicTAO [2], address these issues by introducing new software engineering techniques like component-based architecture and reflection to enable adaptations of the middleware architecture according to specific platforms and deployment instances. On the module level, these approaches are designed with conventional modularization techniques and incapable of modularizing crosscutting design concerns, which can cause a considerable amount of logic tangling in the code.

We recognize that one of the fundamental causes of these problems is that middleware systems have to support many distinct computational requirements, in addition to addressing distributed computing concerns. We define these additional computational requirements as orthogonal design requirements with respect to the fundamental functionality of middleware systems. Traditional top-down decomposition processes produces one decomposition model. That means the abstractions of orthogonal design requirements need to be permanently imprinted into the model at some stages of the decomposition process. Consequently, the logic of orthogonal implementations are intermingled with each other and with that of the primary functionality. It is impossible, by using traditional methods, to completely decouple them from one another.

We think that the phenomenon of handling multiple orthogonal design requirements are in the category of crosscutting concerns, which are well addressed by aspect-oriented programming techniques (AOP). Hence, we believe that middleware architecture is one of the ideal places where we can apply AOP methods to obtain a

• *The authors are with the Department of Electrical and Computer Engineering and the Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4. E-mail: {czhang, jacobsen}@eecg.toronto.edu.*

Manuscript received 8 Dec. 2002; revised 6 Aug. 2003; accepted 9 Aug. 2003. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118745.

modularity level that we cannot obtain via traditional programming techniques. To follow that theoretical conjecture, it is necessary to identify and to analyze these crosscutting phenomena in existing middleware implementations. Furthermore, by using aspect-oriented languages, we should be able to resolve concern crosscutting and yield a middleware architecture that is more logically coherent. It is then possible to quantify and to closely approximate the benefit of AOP in its applications to the middleware architecture design methodology. This paper contributes to the aspect-oriented analysis and design of middleware architecture in the following ways:

1. We show that middleware architectures inherently suffer from coordinating crosscutting concerns by performing a quantitative aspect analysis, which reports the degree of crosscutting of various aspects identified.
2. Our aspect analysis is based on an aspect mining approach. We develop an aspect mining methodology and a software tool to exercise this methodology.
3. Through rigorous aspect mining, we report several new aspects that are specific to the platforms chosen.
4. We are the first to perform aspect-oriented refactorization of middleware platforms. AOP-based refactorization is a process of separating certain orthogonal features out, composing them in aspect programs, and weaving them back into the platform based on an existing implementation.
5. We quantify the benefits of the AOP-based refactorization by applying a set of software engineering metrics to the original and the refactored implementation. From this evaluation, we show that aspect-oriented technology lowers the complexity of the architecture, increases modularity, and preserves the original design requirements, and maintains the performance of the overall system as compared to the original implementation.

The rest of this paper is organized as follows: Section 2 constitutes a more detailed discussion of middleware problems. Section 3 presents a brief overview of aspect-oriented programming that will help to better understand our approach. Section 4 presents the aspect analysis results. Section 5 presents the refactorization of aspects using AspectJ. Related approaches are discussed in Section 6. Section 7 concludes the paper and outlines future work. This paper is an extended work from our previous work in [26].

2 Problems with Today's Middleware Architectures

Generally speaking, middleware systems can be defined as a set of services that facilitates the development of distributed applications in heterogeneous computing environments. Prominent examples of middleware systems include CORBA, DCOM, the Java suite of protocols, and Java RMI and, most recently, Web services. In this section, we briefly summarize a few observations in the evolution of middleware, motivate the use of aspect-oriented techniques for designing middleware architectures, and outline the choice of our case study platform underlying this work.

2.1 Middleware Architecture Evolution

A striking change in the evolution of middleware systems is that, in recent years, the target platforms are not limited to traditional enterprise systems and desktop machines. New platforms include mobile devices, network devices, control systems, safety critical systems, and many more. The characteristics of these platforms differ from each other in significant ways. These platforms are referred to as the emerging application domains of middleware. For example, middleware systems are used on the Cisco ONS 15454 optical transport platform to deal with hardware customizations and the communications between management software and hardware drivers [3]. Middleware systems are also adopted by the US Navy as the software bus for subunits in the submarine combat control systems [4]. The widening of the application spectrum has dramatically stretched the capability of middleware systems. However, the limitations of the traditional middleware architecture has also been becoming increasingly eminent. These limitations include the following:

1. *One size fits all*: Many middleware implementations are collections of many features in order to support a large diversity of target platforms. However, in practice, for a particular instance of deployment in a particular domain, many of these features are not needed all the time. Some are not needed at all during an application's lifetime. Current middleware architectures lack of methods to tailor the architecture at deployment time or at runtime.
2. *Evolving too fast*: Although the networking communication model is relatively stable (consider TCP/IP stack implementation), models or abstractions in middleware systems rarely stay the same. That is because the middleware architecture needs to constantly accommodate new computational characteristics of target domains. For example, the object adapter is a key component in CORBA [5] platforms. Even though the role of the object adapter has not changed significantly since the dawn of CORBA, its architecture has gone through a series of evolutions from the basic object adapter to object adapters supporting portability, interception, and multiple threads. The evolution of middleware abstraction models is necessary and unavoidable. But, at the same time, it is also essential to maintain backward compatibility with applications developed on top of the older models.
3. *Same technology, multiple flavors*: Another problem is that there has been a proliferation of middleware specifications to accommodate different requirements that stem from many application domains. For instance, the Object Management Group (OMG), in addition to defining CORBA standards, also defines the Real-Time CORBA specification in order to address execution predictability, Fault Tolerant CORBA to address high availability, High Performance CORBA to address data and compute intensive applications, and Minimum CORBA for embedded platforms. Java platform exhibits the same syndrome by having different types of JVMs for different platforms. The former DCOM architecture and the new Microsoft .NET platform is following similar trends.

These tendencies cause challenges to vendors, who must rearchitect the system differently according to a particular specification, as well as to the adopters who, at the same time, find the platform very complex to comprehend and to use.

2.2 Applying AOP to Middleware Architecture Design

Zhang and Jacobsen [6] have established that one of the fundamental causes for the problems in today's middleware architecture is that the software decomposition model obtained using vertical decomposition (i.e., designs based on functional or object oriented decomposition) is incapable of simultaneously modularizing coexisting orthogonal design requirements.

Aspect-oriented programming, on the contrary, allows us to decompose software systems in different dimensions. We can use a vertical decomposition process to establish the primary decomposition model of the architecture. We then use aspect-oriented techniques to "horizontally" compose or to "superimpose" the implementation for orthogonal design requirements onto the primary model, without modifying the existing architecture. We refer to that decomposition process as the horizontal decomposition. For a more detailed discussion, refer to Zhang and Jacobsen [6].

There are many limitations of applying a vertical decomposition process to the design of software architecture, in general, and middleware architecture, in particular. The object-oriented paradigm and design patterns facilitate powerful decomposition of software, such as levels of abstraction and advanced modeling methods, as found in the MDA.[1] However, it is still difficult and sometimes infeasible to apply these if the abstractions at higher levels need to be modified. In the middleware domain where requirements are constantly changing, it is extremely hard to get the initial abstraction correct. Another limitation is due to the presence of orthogonal design requirements. Two design requirements are orthogonal to each other if one can be implemented without coordinating with the other, such as in the case of the requirements for efficiency and the requirements for location transparency. Since each of the orthogonal requirements must have its own most appropriate decomposition model, the vertical decomposition process, which generates one decomposition model, may not be an optimized solution for both requirements. In fact, it generates a model with tangled logic, as indicated by Gregor Kiczales et al. [7].

We think horizontal decomposition is a logical approach based on the assumption that vertical decomposition unavoidably causes concern crosscutting in the decomposed model. To confirm that and to further understand orthogonal design requirements in middleware systems, we need to open up existing legacy implementations to conduct quantitative analysis of the tangling phenomenon. The method we have employed to perform such tasks is called *aspect mining*. We explain our aspect mining methodology in the following section.

2.3 CORBA as Case Study

We have chosen CORBA implementations as case study for the following reasons: CORBA has been addressing middleware concerns for more than a decade. Its architecture reflects distinct evolution cycles in the domain of middleware and can be treated as an excellent case study of the traditional functional decomposition approach. CORBA is an open standard by which we are able to achieve a better understanding of its behavior. There are many open source implementations available for CORBA. All conform to the same OMG standard. That allows us to cross-analyze our results. The core of the CORBA middleware is the object request broker (ORB). The ORB provides a standardized middleware platform to allow transparently locating objects and to invoke methods on these objects. The distributed objects can be specified using the interface definition language (IDL). The IDL compiler converts these definitions to a specific language, such as C++ or Java, according to the standardized IDL language mapping specifications. The ORB uses the portable object adapter, the interface of which is also standardized, to process invocation requests. The ORB uses the interorb protocol (IOP) as the communication mechanism on the network to transfer information about the distributed data and operations with other ORBs. The `omg:CORBA:ORB` interface is a standardized facade [8] for providing abstractions of complicated broker functionalities.

3 Aspect-Oriented Programming

Aspect-oriented programming offers an alternative paradigm for software development [7]. It aims at achieving a high degree of separation of concerns. Examples of aspects include security, reliability, manageability, and further nonfunctional requirements, often simply referred to as a system's "ilities" [9]. The existence of aspects is attributed to handling crosscutting concerns in software development using the traditional "vertical" decomposition paradigm. AOP overcomes the limitations of traditional programming paradigms by providing language level support to modularize these systematic properties as separate development activities. The final system results by merging the aspect modules and the primary functionalities together. This development process is commonly supported by a *component language*, such as Java or C, to implement the primary decomposition of a system, an *aspect language* to modularize crosscutting concerns as `aspects`, and the *aspect weaver* (also known as aspect complier) that instruments the component program with aspect programs to produce the final system. Representative aspect languages are AspectJ[2] and Hyper/J.[3] In addition to conventional Java language features, AspectJ defines a set of new language constructs to model the aspects. A `joinpoint` represents an interception point in the execution flow of the component program. For convenience and elegance, a `pointcut` construct can be used to denote a collection of joinpoints. Actions can be triggered before, after, or in place of the program execution when a joinpoint is reached. These actions are defined using AspectJ specific constructs `before`, `after`, and `around`. These constructs are called `advices`. An aspect module in AspectJ contains `pointcuts` and the associated `advices`. It also contains *intertype declarations* which are reused to declare new members (fields, methods, and constructors) in other types.

1. Model Driven Architecture, http://www.omg.org/mda.

2. AspectJ, http://www.eclipse.org/aspectj.
3. Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj.

TABLE 1
Middleware Architecture Elements Overview

	CORBA	DCOM	Web Services	Java/RMI
Service defnition	IDL	MIDL	WSDL	Java interface
Identity publication	IOR	OBJREF	URL	ObjID
Request dispatching	POA	Service Control Manager	application	JVM
On-wire representation	CDR (binary)	NDR (binary)	SOAP as XML (unicode)	marshalled object
Transport	IIOP	ORPC	HTTP	JRMP

4 Quantifying Aspects in Middleware

In aspect-unaware legacy implementations, it is difficult to reason about aspects due to the code scattering problem. That is because the particular aspect is not localized at one point in the code, but rather, distributed all over the code. However, if the scattering phenomena can be captured in a way to allow observation and quantification, effective means to identify aspects for particular application domains become possible. From the identification and quantification of high degrees of code scattering, we infer the existence of aspects, however, the reverse inference is not always possible. This will become obvious in the analysis below, where architecture crosscutting concerns are identified as aspects that do not give rise to high degrees of scattering.

Aspect analysis consists of the activities that aim at capturing and analyzing the scattering phenomenon in legacy implementations. To correctly perform aspect discovery, it is very important to identify the roles involved in the logical tangling. We consider the concept of the aspect as a relative term with respect to the primary functionality or, equivalently speaking, the primary decomposition model. Therefore, in order to analyze the scattering problems in middleware systems, we need to first of all define a decomposition model, which represents the most fundamental properties of the middleware substrate. We are then able to identify aspects by using that model as the reference point to think about tangling in terms of code and orthogonality in terms of design requirements. We refer to this process as aspect orientation.

4.1 Primary Decomposition of a Middleware Architecture

We think that the fundamental functionality of a middleware system, supporting remote invocations, mainly consists of the following major architectural elements:

1. a standardized programming model or API with the associated skeletons and stubs that allows applications to make abstractions of the distributed objects or services,
2. the mechanism of publishing the representation of an object or a service to peers,
3. the dispatching mechanism that forwards the requests associated with the published representation to its concrete instance, and
4. the commonly agreed representation of data and operations on the network layer with their associated interpretation mechanisms in order to exchange information with its remote counterparts.

To be more specific, Table 1 shows the architectural elements that fall into the categories listed above from the popular middleware platforms, CORBA, DCOM, Java RMI, and .NET.

Having identified the primary architecture of a middleware system, we now define *aspects of middleware systems* as concerns which can be decomposed independently and yet have to be addressed in the lines of the primary architecture. More specifically, middleware aspects are abstractions or implementations that crosscut any of these major architectural components enumerated above.

4.2 Extended Aspect Mining Tool

The aspect mining tool (AMT) [10] is designed to capture the code scattering problem on the source code level. It was developed at the University of British Columbia and based on the AspectJ compiler. AMT emphasizes on visualizing perspective aspects in program code and is therefore well-suited to study aspects in small software projects. AMT allows one to inspect the code scattering phenomenon by performing type and textual analysis in building a collection of all the types used in the program in combination with the entire source code space. We have built the extended aspect mining tool (AMTEX) [11] to overcome the limitation of visualization-based mining and scale the mining process to software systems with thousands of classes. The extended mining tool provides much larger flexibility in terms of composing mining activities, managing mining tasks, and cross analyzing mining results. It is designed to fit the needs of mining very large software systems that consist of thousands of classes with millions of lines of code.[4]

4.3 Mining Methodology

We have taken the following approaches to best utilize the analysis capability of the extended mining tool in order to identify aspects in middleware systems:

1. We first inspect well-known aspects that have been previously identified in other software systems, such as logging, synchronization, and others. We are interested in finding the corresponding coding representations in middleware systems and identifying whether these aspects are also present.
2. We then apply our understanding of the functionality of middleware to analyze if some of the features in middleware systems are orthogonal to the primary functionality. To confirm our analysis, we use AMTEX to capture the corresponding crosscutting structures.

4. The reference to "mining" in AMT and in AMTEX is somewhat misleading, since patterns are not autonomously discovered by the tool. Both tools perform an analysis, which with the user interaction may give rise to aspect discover.

3. We use the ranking feature of AMTEX to make sensible guesses. AMTEX is able to rank the popularities of all class types used in the system. Types that are used relatively widely in the code space are able to provide good hints of potential aspects.

The aspect mining process is guided by a human miner who knows or suspects the existance of aspects in the code and guides the discovery process. The human miner begins this process with the initial description of the crosscutting structure of an aspect. This structure can be described, either in succinct lexical and type patterns, or gradually in several steps. We refer to such a description as the *characterization*.

As described in [10], the use of lexical and type-based aspect mining works effectively with programs that have well-defined naming conventions and follow a consistent coding style. This method is appropriate for discovering crosscutting structures at the granularity of statements inside method bodies. For example, code such as `log` or `trace` and types such as `Logger` or `Tracer` are fairly accurate in describing the crosscutting of the aspects *logging* and *tracing*. We use this method primarily to locate well-known aspects such as logging, tracing, synchronization, and others.

For aspects that have no statement-level crosscutting structures but domain-specific semantics, it is difficult to characterize the aspect without a good understanding of the semantics of the program. We use a feedback-directed approach to find a good characterization of this type of aspects in several steps of refinements. That is, a lexical pattern can be used to describe the miner's intuition of what the crosscutting structure might be. Matching results discovered in this step serve as feedback and assist the miner to refine the characterization. We use this approach to locate middleware specific aspects such as type `Any`, for example. This is assisted by the *type ranking* feature of AMTEX, which allows to rank the usage popularities of all class types in the system. Types that are used relatively widely in the code space provide good hints of potential aspects. For example, AMTEX shows that the class type `Assert` is the most used type in the ORBacus code. We then use this class type and quantify the aspect of pre/postcondition checking as further explained below.

Another type of crosscutting structure is the additional control flow incurred in the code through the coordination with an orthogonal concern. We can inspect the values involved in conditional branches and trace all the code involved in accessing these values through assignments, method parameter passing, and accessor methods. We essentially compute code slices based on the conditional statements. If these slices are not localized, we have identified very good candidates of crosscutting concerns. A tool can use this technique and evaluate all conditional statements in the system to find nonlocalized slices. Currently, we perform this technique manually by browsing through the source code with the help of Eclipse and Feat [12]. Our ongoing PRISM project [13] will support this functionality. With this technique, we were able to find aspects such as oneway invocation, which produces a large number of conditional statements for deciding if a oneway invocation is occurring or not. Later sections present details about such an aspect.

TABLE 2
Sizes of Code Spaces (in Number of Classes)

implementation	*CCS*	*VCS*
JacOrb	1778	579
ORBacus	1777	655
OpenOrb	1521	287

4.4 Mining Results

In this section, we present the mining results for a number of aspects in CORBA implementations, including aspects defined prior to this work, such as logging, synchronization, exception handling, and pre/postcondition checking. We also present aspects that are discovered through mining with respect to the CORBA platforms. These aspects include the dynamic programming interface and support for portable interceptors. For each aspect, we report how it crosscuts the primary model in the following format:

1. *Definition:* Explains the context and the definition of the aspect.
2. *Logic Tangling:* Discusses the cause for the aspect by identifying orthogonal design concerns.
3. *Characterization:* The orthogonality of design requirements causes the scattering of their implementations on the code level. In these situations, one implementation can be characterized by a few class types or special textual expressions. This section lists types and expressions that are used to represent the tangling logic in different implementations.
4. *Mining Results:* If the aspect can be characterized by a set of types and textual expressions, we can use the extended aspect mining tool to quantify the usage of these types and expressions. This is expressed by AMTEX as the *degree of scattering* of the aspect in both the Vendor Code Space (VCS)[5] and the Complete Code Space (CCS). The degree of scattering is a measure of the percentage of the usage of the characterization set throughout the source code. AMTEX counts the total number of classes in an application and the number of classes that contain any class type from the characterization set. The ratio of these two numbers is the degree of scattering. All our mining results are expressed as this ratio.
5. *Result Analysis:* This section provides a brief analysis of the mining data.
6. *AOP Benefit:* This section explains how AOP can theoretically help improve modularity, adaptability, and performance of middleware platforms by composing the feature as aspect programs.

The mining data is collected over three open source CORBA implementations: ORBacus, a commercial ORB from IONA Technologies;[6] JacOrb,[7] an open source ORB, commercially supported by OCI; and OpenOrb,[8] a community open source project. All three implementations comply with the CORBA 2.0 specification defined by the OMG. The sizes of these CORBA implementations, in terms of number of classes in VCS and CCS, are listed in Table 2.

5. Classes defined and implemented by the vendor, other than those specified by OMG.
6. ORBacus, http://www.iona.com.
7. JacOrb, http://www.jacorb.org.
8. http://openorb.sourceforge.net.

TABLE 3
Degree of Scattering for Dynamic Programming Interface

implementation	*CCS*	*VCS*
JacOrb	31.56%	23%
Orbacus	31.2%	26.56%
OpenOrb	32%	23%

TABLE 4
Degree of Scattering for Portable Interceptor Support

implementation	*CCS*	*VCS*
JacOrb	0.51%	3.52%
Orbacus	2.66%	7.09%
OpenORB	2.44%	13%

4.4.1 Dynamic Programming Model

Definition: A dynamic programming model allows an application to be designed without prior knowledge of the interface definitions of the invoked objects. Instead, invocations on an interface can be composed during runtime.

Characterization: In CORBA, the OMG defined objects that handle DSI/DII are the following classes in terms of the Java language mapping: `org.omg.CORBA.Any`, `org.omg.CORBA.NamedValue`, `org.omg.CORBA.NVList`, `org.omg.CORBA.Request`, and `org.omg.CORBA.ServerRequest`.

Logic tangling: The dynamic programming model (DII/DSI) crosscuts the ORB functionality in the following ways:

1. All Helper and holder classes[9] contain operations to allow transformation and manipulation of these server objects in the dynamic invocation context.
2. The ORB interface supports dynamic invocations by providing the functionality of composing operations, their parameters, and their return types.
3. The request processing classes supports dynamic invocations through specialized request objects such as `org.omg.CORBA.Request` and `org.omg.CORBA.ServerRequest`, and also through extra control logic.
4. The encoding and decoding mechanisms, which map the program data and operations to bit streams according to the transport protocol, support the dynamic behavior by using value types such as `Any` and `NamedValue`.

Although the list above is not complete, we have observed that the support for DII and DSI is incorporated into the static invocation model not only as a part of the programming interface, but also in the request processing mechanism and in the encoding/decoding process. Therefore, we refer to the dynamic invocation mechanism as an aspect of the ORB if its primary invocation model is static. Similarly, we can refer to the static invocation model as an aspect of the ORB, if its primary invocation mechanism is dynamic.[10]

Mining Results: The degree of scattering for the dynamic programming interface is reported in Table 3.

Result Analysis: The data presented confirms the above analysis that the code handling the dynamic programming model is not well modularized. More than a quarter of the classes in all three implementations deal with DII or DSI in some way. Also, the degree of scattering of the dynamic programming model increases, with increasing code size.

9. As defined in [14], all user-defined IDL types should be generated with additional helper and holder classes to allow convenient manipulation of these types.

10. An earlier version of the popular C++ MICO CORBA implementation was entirely based on the dynamic invocation interface.

AOP Benefit: Aspect-oriented programming can be applied here to separate the dynamic programming model from the static programming model. That is, we can write an aspect program for the static model to support the dynamic model, or vice versa. The use of AOP techniques has a number of advantages over the conventional ORB architecture. First of all, the separation of concerns liberates the ORB architect from the effort of incorporating the dynamic model into the static model, or vice versa. Both models can be better designed, modularized, and optimized. Second, since, in most cases, only one invocation model is sufficient for a particular domain application, implementing the dynamic programming interface as aspect program gives us the option of either "weaving" the feature in or leaving it out. The ORB architecture, therefore, becomes more configurable, adaptive, and computational efficient.[11]

4.4.2 Portable Interceptors

Definition: Portable Interceptors are hooks into the ORB through which CORBA services can intercept various stages during the object request processing. Interceptors allow a third party to plug in additional ORB functionalities such as transaction support and security.

Characterization: The implementation for supporting interceptors is characterized by the usage of the interceptor class type.

Logic tangling: The specification for interceptors were added to the CORBA specification at a much later time; that is, after the basic functionality of the ORB had been defined and implemented. The support for interceptors is incorporated in the implementation of the POA and other objects that are directly responsible for request processing as follows:

1. The ORB interface contains methods and data members to allow the registration of interceptors. It also provides methods to allow access to these interceptors when it is necessary to notify them upon reaching interception points.
2. During the propagation of the request, the invocation context is checked to see if it is modified by interceptors.
3. The POA needs to bundle requests with information of interceptors if they are registered with the ORB.

The number of objects, such as POA, is proportional to the number of server objects in the domain applications. Therefore, an even higher degree of scattering could happen during runtime.

Mining Results: The degree of scattering for the portable interceptor support is reported in Table 4.

11. The MinimumCORBA profile, for example, leaves all dynamic object management functionality out of the specification, arguing that, for resource constraint embedded applications, this functionality is not required.

TABLE 5
Degree of Scattering for the Type `Any`

Implementation	*CCS*	*VCS*
JacOrb	27.4%	21.8%
Orbacus	30.7%	26.1%
OpenOrb	31.3%	22.6%

TABLE 6
Degree of Scattering for Wchar and Wstring Support

Implementation	*CCS*	*VCS*
JacOrb	1.2%	2.0%
Orbacus	3.3%	4.8%
OpenORB	4.2%	10.1%

Result Analysis: The mining results show that portable interceptor support is a crosscutting phenomenon, mostly in the vendor code space since the spreading in CCS is much lower.

AOP Benefit: Portable interceptor support is typically needed to support enterprise computing features such as security, transaction processing, and fault tolerance. Implementing the support for interceptors in aspect programs is attractive, since they can be configured out if they are not needed.

4.4.3 Type Any

Definition: We categorize type Any as part of the support for the dynamic programming style of CORBA. We provide an independent analysis for this type because it is one of the primitive data types of IDL and, hence, can be used beyond just dynamic programming. For example, `Any` is also used in representing generic information when querying the ORB's execution state or the interface repository.

Characterization: In CORBA, the IDL data type `Any` is mapped to the class `org.omg.CORBA.Any` in terms of the Java language mapping. In addition, we also include the vendor specific implementation class for this language mapping. The ORBacus implementation is `com.ooc.CORBA.Any`, `org.jacorb.orb.Any` for JacORB, and `org.openorb.CORBA.Any` for OpenORB.

Logic tangling: The support for the type `Any` crosscuts the ORB functionality in the following ways:

1. All the helper and holder classes must support the conversions between the statically typed classes and the type `Any`.
2. The ORB interface supports factory methods for user applications to obtain the vendor specific instance of the type `Any`.
3. The encoding and decoding mechanisms, as explained in Section 4.4.1.
4. The use of `Any` to represent generic information about the execution context of the ORB as part of the portable interceptor framework.

Mining Results: The degree of scattering for the dynamic programming interface is reported in Table 5.

Result Analysis: The data shows that the scattering of the type `Any` in all three implementations are consistently high. Therefore, we categorize the type `Any` as an aspect of CORBA.

AOP Benefit: For applications that do not require the support of the IDL type `Any`, it is currently not possible to remove its support in conventional ORB architectures. Therefore, separately decomposing the support for `Any`, as a set of aspects, will lead to a simpler architecture, while preserving the CORBA functionality.

4.4.4 Wchars and Wstrings

Definition: Wchars are characters with an expanded number of bytes. Every wchar has an equal number of bytes and the number is platform dependent. CORBA uses wchar to support additional sets of character encodings such as Unicode. A Wstring is a sequence of wchars. We treat wchar and wstring or, more generally speaking, primitive data types, as aspects, because each data type represents an independent, hence orthogonal way of encapsulating and interpreting data in applications.

Characterization: We use the lexical pattern "wchar" and "wstring" to locate class concern with the support for these two data types.

Logic tangling: The implementation of wchar and wstring crosscuts the CORBA implementations as follows:

1. The initialization process of CORBA needs to include the loading of the encoding (i.e., Code-set) information of wchars.
2. The encoding and decoding mechanism contains special operations related to the reading and writing of wchars in incoming and outgoing octet streams.
3. CORBA supports multiple encoding formats for characters. Each encoding format also needs to support wchars and wstrings.
4. The error handling mechanism contains error codes and error messages related to operations with wchar and wstring.

Mining Results: The degree of scattering for the wchar and wstring support is reported in Table 6.

Result Analysis: The support of `wchar` and `wstring` causes varying degrees of scattering. The degree of scattering is generally low because the crosscutting mainly involves operations regarding characters but not the call processing. However, the implementation of wchar support is still nonlocalized.

AOP Benefit: The support for wide characters and strings, as well as other primitive data types, is not localized and modular. Decomposing IDL data types as aspects rather than in a scattered fashion allows us to support user applications with the right set of data types and reduces the complexity of the architecture, eventually offloading non-required types for specific resource constraint applications, for example.

4.4.5 Oneway Invocation

Definition: The oneway invocation semantic of CORBA supports the best-effort delivery of client requests and is of an asynchronous nature, without any return value sent back to the client. That is, the thread of control is returned to the user application once the invocation is sent to the network layer. We categorize the oneway invocation as an aspect because it has an orthogonal semantic to the synchronous invocation, which is typically the primary communication mechanism in CORBA.

Characterization: As previously mentioned, the characterization for the aspect of oneway invocations is the slicing of the branching condition of whether the invoked interface is

TABLE 7
Degree of Scattering for the Oneway Support

Implementation	*CCS*	*VCS*
JacOrb	0.8%	2.1%
Orbacus	2.4%	4.7%
OpenORB	5.4%	10.7%

tagged oneway or not in its IDL definition. Since AMTEX does not yet evaluate this type of aspect characterization, we use the lexical pattern "response" as an approximation. We have verified that this approximation is fairly accurate, as all three implementations use the condition "`response_expected`" to decide if the current invocation should be treated as oneway or not.

Logic tangling: The implementation of oneway crosscuts the ORB as follows:

1. The oneway invocation is addressed in the stub/skeleton layer to support the usage of the oneway IDL keyword.
2. Depending on whether the request is oneway or not, the client-side takes different processing steps so that the stub will not expect a response from the server.
3. The server issues a reply message to the client's request in the case of synchronous communication. This reply is not needed if the invocation is oneway. Therefore, the server-side call dispatching process needs the "oneway" information of the current request to determine the execution path.
4. The low-level transport and protocol layer addresses the encoding/decoding information and related socket operations regarding the oneway semantic.

Mining Results: The degree of scattering for the oneway support is reported in Table 7.

Result Analysis: The results show that the scattering of oneway support mainly exists in the vendor code space. Although the degree of scattering is relatively small, the runtime ramification of this aspect is high since the scattering of oneway concentrates on the path of call processing. Our metric does not capture this effect.

AOP Benefit: The oneway invocation is commonly implemented as a set of branched executions in both the marshalling and unmarshalling processes, regardless of whether the current invocation is oneway or not. If the application does not use oneway calls, these conditionals become redundant, and yet, are still part of the execution path of the synchronous invocation. This, in turn, degrades the performance of synchronous invocations in CORBA implementation not designed with aspects. Composing oneway as aspects will make the synchronous invocations more efficient without losing the support for oneway invocations.

4.4.6 Common Aspects

In this section, we collectively treat the error handling, the pre/postcondition checking, the logging, and the synchronization functions. These are well-known aspects found in most large software systems.

Definition: Error handling deals with unexpected states of the system during execution. Sophisticated software systems, such as middleware platforms, require a robust, flexible, and manageable error handling mechanism such as offered by exception handling mechanisms found in many programming languages.

Pre/postconditions are used in applications to validate the correct state of the data either before or after the data is processed.

The purpose of logging is to report the running state of the system and to aid in collecting debugging information.

Synchronization primitives, such as mutex, semaphore, and monitors, are used to protect critical regions and the valid state of shared resources.

Characterization: Exception-based error handling can easily be characterized through the use of exception classes and expressions that match "try" and "catch" blocks.

The implementation of pre/postcondition checking is application specific. Typically, it can be characterized by the use of assert-like statements.

The implementation of logging is highly application specific. For example, JacOrb uses the method `void printLog(int mode, String message)`, and ORBacus uses the following methods: `info(String message)` and `error(String message),warning(String message)`.

The use of synchronization can be characterized by the synchronization data types such as mutexes, semaphores and monitors. In addition, the keyword "synchronized" in Java programs identifies this aspect.

Logic tangling: Error handling can be considered as an aspect because, although never being a design goal of any application, error handling is a property that has to be incorporated in the architecture in an ad hoc manner. More importantly, the error handling mechanism provided by a language, such as Java exceptions, can incur execution overhead that cannot be overlooked in certain computing environments. That is particularly important to middleware because it should be designed to fit a wide range of platforms. This clearly indicates that the error handling mechanism, such as exceptions, crosscuts the design goal of performance optimization.

We consider the pre/postcondition checking an aspect because it implements the design requirements of validity. The validation checking by nature scatters throughout the decomposition model.

Logging can be considered as an aspect because it is designed for "debuggability" in sacrifice for system performance and maintainability of the source code.

Synchronization, as a pervasive property, is applied where resource contention exists. Synchronization is a well-known aspect not only for middleware platforms, but also for software systems in general.

Mining Results: The degree of scattering for the analyzed aspects is reported in Table 8. Since logging, pre/postcondition checking, and synchronization, other than error handling, are implementation specific, the mining is conducted over the vendor code space only.

Results Analysis: The mining data indicates that the scattering degree of error handling is almost proportional to the code size. Furthermore, the data confirms that modularity is clearly violated in the implementation. Similarly, for the other functions, which are also widely scattered through the code, the analysis confirms their aspectual nature.

AOP Benefit: We briefly discuss the key benefits of modularizing the functions discussed in this section as aspects.

TABLE 8
Degree of Scattering for Error Handling, Pre/Postchecking, Logging, and Synchronization Aspects

	error handling		pre/post	logging	synchronization
implementation	*CCS*	*VCS*	*VCS*	*VCS*	*VCS*
JacOrb	25.8%	46.5%	13.3%	14.66%	14.11%
Orbacus	39.3%	45.5%	5.47%	18.9%	9.16%
OpenORB	35.3%	44.6%	10.4%	16.3%	9.41%

TABLE 9
Combined Scattering of Aspects

implementation	*total number*	*one aspect*	*ratio*	*three aspects*	*ratio*
Jacorb (CCS)	1778	1273	71.6 %	309	17.4%
OpenOrb (CCS)	1521	760	50.0 %	460	30 %
ORBacus (CCS)	1777	813	45.75 %	559	31.5 %
Jacorb (VCS)	579	319	71.6 %	110	20.0%
OpenOrb (VCS)	287	155	54.0 %	81	28.0%
ORBacus (VCS)	655	340	51.9 %	212	32.4%

1. Although Java exceptions provide a powerful mechanism for error handling, it does come with a performance price, because not only does it increase the code size, for the Java compiler has to generate the exception management code, but it also elongates the normal execution path of the application because of the stack unwinding process. The exception mechanism might not be a big factor for performance in the enterprise or desktop computing environment. However, in the case that the computing resources are limited, it may be desirable not to use exception support, but to use return code-based error handling. That greatly decreases the code size and performance overhead. However, it is very difficult and cumbersome to allow switching between the two error handling schemes using traditional programming paradigms. AOP can provide an elegant solution by allowing us to compose the source code for the basic functionality and the error handling functionality separately. For example, we can write two sets of aspect programs, one using exceptions and the other using return code checking. We can configure the ORB statically or dynamically to use return code checking instead of exceptions if resources are constrained.[12] In addition, any modifications to the error handling logic would not affect the ORB code, thus making changes less error prone.
2. The modularization of pre/postcondition checking with aspects offers the option of enforcing systematic checks and policies, which can also be easily modified. In the case of traditional programming techniques, developers have to manually go through the entire code space to make the corresponding changes.
3. Aspects can be applied to separate the logging code from the core logic of the implementation. In that way, the state reporting capability can be preserve without introducing any performance overhead in the final system. The source code of the ORB can be more concise and clean, while the logging code becomes more modular and more reusable.
4. Synchronization assumes a concurrent execution model. That assumption is not always valid for certain platforms where it is costly to use concurrency for the limitation of computing resource and power efficiency concerns. Aspect implementation of synchronization enables the feature to be configured in or out depending on the target platform. Synchronization implementation can also become more modularized.

12. When the MinimumCORBA specification was first discussed, the need for costly exception support in the target domain, embedded systems, was questioned.

4.4.7 Cross-Comparisons of Mining Results

The sections above have listed the mining results of several individual aspects. We have used our extended aspect mining tool to combine these individual results, which yield statistical information that illustrates the scattering of all these aspects in the CORBA platform from a slightly different angle. Table 9 shows, for each of the three CORBA implementations, the number of classes in VCS that are crosscut by at least one aspect. It also shows the number of classes containing three or more aspects.

These results strongly suggest that, in the investigated platforms, tangled logic, which gives rise to aspects, is a common phenomenon in both implemented classes as well as the IDL-compiler generated classes. We conclude that vertical decomposition yields architecture models that inherently suffer from coordinating orthogonal design requirements. Moreover, these results almost suggest that the scattering of code in implementations of middleware systems is consistent regardless of whether the development model is *cathedral* or *bazaar*.[13] However, further experiments need to be carried out to more fully validate this observation.

5 Aspect Refactoring Middleware Using AOP

Our aspect mining results have shown that concern crosscutting is an inherent property of conventional middleware

13. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.

implementations. However, it is still not clear how AOP techniques could improve the architecture and result in implementations that, at least, preserve the design requirements. Obviously, that question cannot be answered without quantitative comparisons between two instances of implementations, one using traditional decomposition techniques and the other using the aspect-oriented approach. Our long term research goal is an aspect-centric design of a middleware platform. Here, we base our comparative evaluation on a refactored legacy implementation. That is, if we identify a certain functionality, which constitutes an aspect as the result of aspect mining, we should be able to separate that functionality from the original implementation, compose the functionality as an aspect, and weave it back into the architecture. All design requirements should still be satisfied transparently to the user. We believe that "refactoring" closely resembles what happens in a completely aspect-oriented approach. The following sections will present the set of software engineering metrics we have used in order to quantify the architectural differences. We then describe our refactoring approach, the detail of the aspect-oriented implementations, and the results of the evaluations.

5.1 Quantification Metrics

Metrics are measures for the quality of software. In [15], the following software metrics are used based on the static information available in the implementation. These are

1. cyclomatic complexity,
2. size,
3. weight of the class (in number of methods),
4. coupling between classes,
5. DIT (depth of inheritance tree),
6. number of immediate children, and
7. lack of cohesion in methods.

These metrics can be applied to assess the design quality of both component programs as well as aspect programs in general. In our analysis of the benefit of AOP as compared with the traditional programming methodology, not all of the above metrics apply. In addition, while applying these metrics, our reference is the primary program (i.e., implementation of the primary decomposition), with the recognition that the sum of complexities of both the primary program and aspect programs might not necessarily decrease. We can mainly put the primary program under our consideration because aspect programs are to be configured in and out and are maintained separately. We think it is appropriate to use a combination of metrics to address various properties of the aspectized architecture, including both the static properties, which directly tie to the cost of development and maintenance, and the runtime characteristics, which reflect the cost of adopting the technology. The following is a list of metrics that are used to quantify aspectization in our work. For each metric, we also provide analysis of the expected change due to aspectization and the rationale for the change.

1. *Cyclomatic complexity*: Cyclomatic complexity is a measure of alternative execution paths in code segments caused by control flow statements. It is an index obtained through heuristics and is independent of specific languages. The SEI defines the following measures for cyclomatic complexity [16]: A cyclomatic complexity number (CCN) between 1 to 10 refers to simple programs without much risk; a CCN between 11 to 20 refers to more complex programs indicating moderate risk; and a CCN between 21 and 50 identifies complex programs that are highly risky. A lower CCN, that is, fewer alternative execution paths, makes the program easier to understand and to maintain. More importantly, it could also improve the runtime characteristics of the program in significant ways, such as better cache performance and less coverage tests. AOP refactoring should lower the cyclomatic complexity because it takes the handling of crosscutting properties out from the primary decomposition.
2. *Size*: In [15], several methods are mentioned to assess the size of a software system. In this paper, we define size as the total number of executable lines in all measured classes. The size of a software system directly ties to the development and maintenance cost. AOP refactoring decreases the number of comments and blank lines of code, as well as the count of executable statements of the primary program.
3. *Weight of a class*: The weight of a class is the average number of methods per class. It reflects the complexity of classes. AOP refactoring decreases the weight of classes.
4. *Coupling between classes*: Coupling is a measure of how much the types in the system are related to each other. A good architecture is always less coupled due to good modularization. AOP refactoring decreases coupling by separating the primary program from the knowledge of the types, which implement the crosscutting logic. In this paper, we measure the average number of classes that a class has connections to in the call graph.
5. *Response time*: In the context of middleware, response time can be defined as the time taken to respond to an invocation request by the request broker. This is defined as the total time a message traverse through the middleware stack during its round-trip. We divide that into four intervals of time as messages are processed in the middleware stack:

   a. **Interval A:** client-side marshalling;
   b. **Interval B:** server-side unmarshalling and dispatching;
   c. **Interval C:** server-side marshalling; and
   d. **Interval D:** client-side unmarshalling.

   It is necessary for the aspect-oriented refactoring to at least preserve the performance measure. In the case of having crosscutting features factored out, aspectization is expected to decrease the processing time due to the simplification of the program logic.

To ease our discussion, we classify these metrics into two categories, structural and behavioral. Structural metrics include size, weight, and coupling. Behavioral metrics reflect the runtime characteristics of the system, such as, in our case, the response time.

The structural metrics are collected over classes that are involved in the refactorization. We use JavaNCSS,[14] an open source metric collection tool, to collect cyclomatic complexity, average class size, and weight. We use our extended aspect

14. http://www.kclee.com/clemens/java/javancss/.

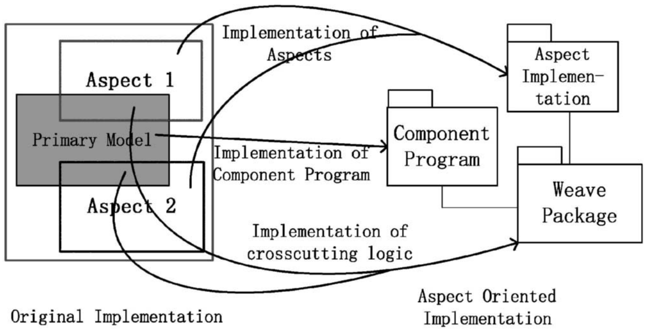

Aspect 1
Primary Model
Aspect 2
Implementation of Aspects
Implementation of Component Program
Implementation of crosscutting logic
Aspect Implementation
Component Program
Weave Package
Original Implementation
Aspect Oriented Implementation

Fig. 1. Code transformation for the aspect-oriented refactorization.

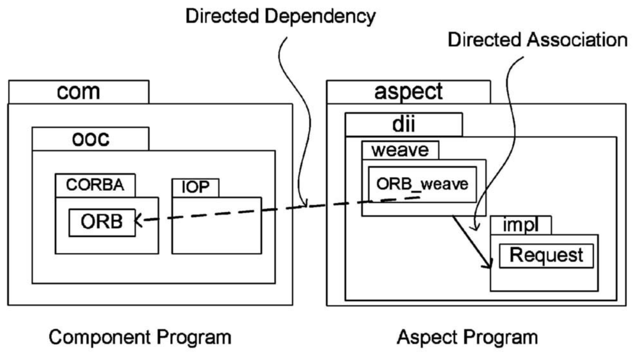


Fig. 2. Package organization for aspect-oriented refactorization.

mining tool to collect the coupling index and the degree of scattering. To measure performance, a simple C-based timing tool is written based on the Java Native Invocation interfaces. It is integrated with AspectJ programs to insert various measurement points into the ORBacus execution stack. The stack traversal intervals are measured in microseconds and computed as the average of 100,000 remote invocations on a Pentium III 1G NT workstation. Each remote invocation involves an integer message sent from the client process to the server. The server also responds with an integer message. We have carefully chosen the measurement points to exclude the socket operation. Therefore, the influence of the size of the message is minimal.

5.2 The Refactoring Approach

There are a few key artifacts in an aspect-oriented system, namely, the component program, the aspect program, and the aspect weaver. We use AspectJ as the aspect language for its maturity and its natural integration with the Java programming language. We pick ORBacus, the CORBA implementation used previously for the aspect mining, as the component program. To verify the correctness of the refactorization, we adopted the demonstration code, which is a part of the standard ORBacus source distribution, to serve as test cases. The test programs invoke CORBA functionality without being aware of the refactorization that is performed. The test programs are also used for performance measurements. As the first step of refactorization of the ORBacus implementation, we need to identify the presence of the crosscutting property in two forms, the implementation structure for the property and the crosscutting points in the primary decomposition model for that property. By using that model, the tangled code is transformed to three types of groupings of classes in the aspect-oriented implementation, namely, primary classes, aspect implementation classes, and the weaving classes. The transformation is illustrated in Fig. 1, where the outside box on the left depicts that the original implementation is one monolithic entity. The primary model and the aspect model coexist in a single structure with parts intersecting each other. The package diagrams on the right presents a clear division of structures. The importance of such division is that it allows all three components to be designed, tested and evolved with unprecedented independence and freedom.

5.3 Aspect-Oriented Refactored Implementation

In this section, we present our refactored implementation of a number of crosscutting features of ORBacus in AspectJ, including the dynamic programming interface, support for portable interceptors, collocated invocation, and logging. For each feature, we first provide the rationale of the construction of the package scheme, which is described in the previous section. We use the package diagrams in Fig. 2 to illustrate the hierarchical structure and the major types of relationship between aspect packages and the component program, using the dynamic programming interface as an example. We then present the evaluation of the metrics as the result of factoring out that specific feature from the ORBacus implementation. A detailed presentation, including code snippets and discussion of the aspects, of our refactoring is presented in Zhang and Jacobsen [17] which, due to space limitations, cannot be included here. At the end of this section, we will look at the overall change of the metrics after we factor out all the "aspectized" features. For clarity purpose, we use the term "client" to denote a role that is requesting a service and the term "server" to denote the role for providing that service, with the observation that, in the middleware context, the classification of client and server is a relative one.

5.3.1 Dynamic Programming Interface

Aspect mining revealed that the dynamic programming interface can be treated as an aspect of the CORBA implementation, which mainly supports stub and skeleton-based static invocation methods in its primary decomposition model. Our AOP-based refactorization of the dynamic programming interface consists of two parts, the client side dynamic invocation interface (DII) and the server side dynamic skeleton interface (DSI).

Dynamic invocation interface (DII):

- *Aspect implementation*: The client-side facility for the dynamic programming model, is supported through the implementations of the interface `org.omg.CORBA.Request` and `MultiRequestSender`. These two class types are taken out of the original implementation and grouped under the aspect implementation package for the DII.
- *Crosscutting points*: We then identify, in the primary decomposition model, the places where operations of classes need to acquire or to exploit the knowledge of these class types identified in step one. These places are the crosscutting points of the DII aspect. In AspectJ, these crosscutting points can be implemented as `"joinpoints"` instead. To be more specific, the crosscutting points of the aspect DII is summarized below in terms of how to change the original model using AspectJ.

TABLE 10
Metric Matrix for DII (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	5.08	1559	40	40.67	105	59	37	125
Aspectized	4.76	1490	37.67	39.33	108	59	35	124

TABLE 11
Metric Matrix for DSI (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	3.64	274	9.33	14	79	8	43	126
Aspectized	3.46	262	9.33	13.5	76	9	41	119

- We use the "`introduction`" construct to factor out a number of methods to handle multiple DII requests in the ORBacus implementation of the `org.omg.CORBA.ORB` interface.
- The "`introduction`" construct is used to factor out the downcall creation logic for dynamically composed downcalls.
- "`introduction`" is used to factor out the code in the object delegates [14], which creates dynamic invocation request objects.
- We group the aspect classes that contain implementations of these "`joinpoints`" in the *weave* package.

- *Evaluation*: The structural metrics are collected on the ORBacus implementation prior to AOP refactoring and after the DII is factored out. The response time is measured using both the original implementation and "woven" implementation. The data indicates that the structural change as the result of factoring out the DII is same as predicted and runtime performance of the aspectized DII is equivalent to that of the original implementation (see Table 10).

Dynamic skeleton interface (DSI):

- *Aspect implementation*: The server side facility of the dynamic programming model is supported through the ORBacus implementations of the interface `ServerRequest` and the user specific implementation of the interface `DynamicImplemenation`. We first remove these two class types and group them under the aspect implementation package.
- *Crosscutting points*: We implement the crosscutting points of the aspect DSI in the primary decomposition model as follows:
  - We first remove the code segments, which dispatches client requests to a dynamic server implementation, from the request dispatching code. Then we use the "`around`" construct to replace the request dispatching call with an alternative implementation, which appropriately handles dynamic server implementations.
  - ORBacus prohibits the direct invocations for DSI server implementations. We move the logic of checking whether an invocation is toward a dynamic implementation or not into the aspect implementation. The "`before`" construct is used to precede the normal invocation process in order to prevent direct invocations.
- *Evaluation*: For this evaluation, we use the static invocation interface on the client-side. The client-side processing times, interval A and interval B, are therefore dramatically decreased as compared with the DSI. However, that change is irrelevant to our AOP refactorization. The data shows that factoring out the DSI has simplified the control flow and decreased the class size. The average weight of classes dose not change because server-side support for the dynamic programming interface is much simpler. No additional methods are used to support the DSI in the original implementation (see Table 11).

5.4 Support for Portable Interceptors

- *Aspect implementation*: In ORBacus, the functionality for portable interceptors is implemented through three categories of classes. They include the classes related to implementing the interceptor interfaces defined by the OMG. They also include the ORBacus specific interceptor initialization classes and request processing classes that support portable interceptors. We separated classes in these three categories from ORBacus, and grouped them under the aspect implementation package.
- *Crosscutting points*: We implement the crosscutting points where the primary ORB model tangles with support for portable interceptors in AspectJ. These crosscutting points correspond to the specified behavior of portable interceptors. That is, an ORB implementation must allow interceptions of the client request process, of the server request process, and the creation process of server objects. The following is a summary of our AOP implementations:
  - The portable interceptor allows the processing of request sending to be intercepted before it is completed. Therefore, in ORBacus, the request sending process, e.g., the downcall creation process, needs to check if any client request interceptors are registered. If yes, a downcall class initialized with the portable interceptor information is created instead of a plain one. However, ORBacus performs the checking regardless of whether portable interceptors are used. We moved the code segments into the aspect program in an "`around`" construct. As

TABLE 12
Metric Matrix of Portable Interceptor Support (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	4.11	3016	26.8	34.75	78	8	42	118
Aspectized	4.0	2909	26.7	32.38	79	9	42	122

the result, the "aspectized" ORBacus only performs necessary checks if portable interceptors are required for a particular application.

- A similar situation occurs in the server side request dispatching process, e.g., the upcall creation process. We move the checking and upcall creation code into the aspect implementation. That makes the server request processing leaner and more precise. That is, it only needs to know and to handle portable interceptors when it is required to do so.
- The portable object adapter (POA) plays a key role in the process of object creation. It needs to notify all the interceptors if there are interceptors registered for intercepting object creation processes. Consequently, the POA code needs to have extra control paths in order to support that requirement. We moved that checking logic into the aspect code and implemented the same logic via the `"after"` construct. That is, following the completion of object creation, the checking code is executed only if the support for portable interceptors is required.
- The ORB also contains the initialization code for loading portable interceptors and registering them with the ORB. We move the corresponding code into the aspect implementation such that, if interceptor support is not needed, it is no longer necessary for the ORB to perform the extra initialization procedures.

- *Evaluation*: This section presents the measurements of structural metrics and the response time. The response time is measured as the time for a message to traverse through the four intervals with and without the presence of the support for portable interceptors. The data of structural metrics show the same direction of change as in the case of the dynamic programming interface. The coupling factor exhibits a larger drop because interceptor support is implemented by a larger number of classes. The runtime performance is equivalent to the original implementation (see Table 12).

5.5 Invocation of Collocated Objects

The key abstraction provided by middleware systems is the transparency of the location of server objects. Location transparency allows remote services to be invoked in the same fashion as calling a method on an object while performing marshalling and unmarshalling behind the scene. Some CORBA implementations optimize the calling process to avoid unnecessary marshal/unmarshal work, in the case where server objects are deployed or migrated into the same process as the client. In ORBacus, the optimization logic is an integral part of the request processing process, which is designed primarily for making remote invocations. We believe the optimization for in-process server objects in ORBacus is logically orthogonal to its remote invocation mechanism. Therefore, we identify the optimization for local invocations as an aspect of the ORBacus implementation of CORBA. Since we treat it as an ORBacus specific phenomenon, we omitted the corresponding aspect mining analysis due to the lack of generality.

- *Aspect implementation*: In ORBacus terms, in-process objects are referred to as collocated objects. To distinguish between normal remote invocation calls and calls to collocated servers, ORBacus uses `CollocatedClient` and `CollocatedServer` to handle corresponding request processing for the client and server, respectively. We completely decouple these class types from the ORBacus source and move them into the aspect package.
- *Crosscutting points*: In ORBacus, the collocated invocation is mainly implemented in the object initialization phase for both the client and the server. We have used AspectJ to reimplement the collocated invocation in aspect programs as follows:
  - We use the `"after"` construct to create the server-side objects that are responsible for processing collocated requests, after the objects for servicing remote invocations are created.
  - We use the `"around"` construct to weave in the client-side logic of checking whether the object reference is pointing to a collocated server. If yes, a different communication model is set up to avoid marshalling and network operations.
- *Evaluation*: The response time is measured by running collocated client/server communications on the original ORBacus and the "aspectized" ORB (see Table 13).

5.6 Logging

- *Aspect implementation*: Logging in ORBacus is carried out mainly by two class types, `Logger` for performing the logging operations and `CoreTraceLevel` for setting the levels of logging. We remove all instances of these two class types from the ORBacus implementation and group them under the aspect package.
- *Crosscutting points*: The crosscutting points for logging includes the following three categories:
  - Recording of the error information in exception handling code.
  - Tracking of the request processing.
  - Recording of network connection activities, such as close of connection, decoding of data, and others.

  We use various AspectJ constructs to implement these crosscutting points in the aspect program.
- *Evaluation*: The response time is measured by using the client and server that are both based on the static invocation interface. From the measurements, it is

TABLE 13
Metric Matrix of Collocated Invocations (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	5.22	449	11.33	21	79	8	43	126
Aspectized	5.00	435	10.67	19.33 1	76	7	41	126

TABLE 14
Metric Matrix for Logging (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	5.22	449	11.33	21	79	8	43	126
Aspectized	5.00	435	10.67	19.33 1	76	7	41	126

TABLE 15
Metric Matrix for Overall Evaluation (for Exact Units, See Text)

	ACC	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Original	105	5393	412.5	528.25	76	9	37	126
Aspectized	101	4899	400	458.25	74	8	37	123
Factored Out	NA	NA	NA	NA	76	9	37	123

easy to observe that, while refactoring of logging decreases all structural metrics, it has a particularly high impact on the size of the program. The runtime performance does not change significantly (see Table 14).

5.7 Overall Assessment and Analysis

We have presented comparative results with respect to each individual feature that is refactored by using AspectJ. We are also interested in the overall effect of the collective refactoring of ORBacus, in terms of change in its structure and the response time. In particular, we want to verify that ORBacus is able to, at least, maintain the same level of service in terms of handling remote object invocations. We use the same experiment settings as in the individual evaluations. The response time data are collected over three ORB implementation instances, the original one, the one with all the "aspectized" features factored out, and the implementation with all the features "woven" in. The structural metrics are collected on the set of all modified classes.

The structural metrics indicate that, to implement the primary functionality of the ORB, we have significantly lowered the complexity of the architecture via the aspect-oriented approach. The data show that the same functionality can be implemented at least by 9 percent less code, 12 methods fewer in total, and 70 instances fewer in terms of coupling with other classes.

From the observation of the response time measurements for the overall refactoring as well as individual ones, we draw the conclusion that the "aspectized" architecture is equivalent to that of the original architecture. The difference is in a few microseconds, which is negligible for Java applications. AspectJ implementations hardly incur any overhead because we are simply moving the code from the original program into the aspect program. In other words, the original ORB is executing the code anyway (see Table 15).

5.8 Limitations

During our aspect-oriented refactoring of ORBacus, we have realized some limitations in our approach due to insufficient research in the area, overwhelming programming effort and limitations in tool support.

1. We did not completely factor out class types such as `Any` and `NVList`, which are used widely for other purposes in addition to the dynamic programming interface, such as for request context passing. While failing to factor these types out does not prevent us from evaluating the aspect-oriented approach, we defer the work until future research when it becomes necessary to exactly quantify all aspects tied to the dynamic programming interface aspect.
2. We decided not to change the IDL-to-Java mapping portion of the implementation, since we believe the appropriate approach is to make the IDL code generator aware of the existence of aspects. We defer the discussion until future works. As a consequence, the user code is still able to use the corresponding OMG interfaces for a feature that is possibly factored out. The ORB throws exceptions during runtime to flag these features do not exist.
3. We decided not to collect the memory usage due to the fact that our "aspectization" experiment is conducted on the Java platform. We do not have an accurate memory profiling tool that allows us to the monitor memory usages of the application objects. Also, the expense of running the full JVM makes the memory improvements achieved by our AOP refactoring almost trivial. Memory footprint will become a more important metric for non-Java-based aspectization projects.

6 Related Work

Although we do not have knowledge of any research project that is directly related to applying aspect-oriented programming to improve the internal architecture of middleware to achieve customization, there are numerous projects that are looking into using AOP to improve the modularity of software systems in general.

Putrycz and Bernard [18] present an aspect-oriented approach to add load balancing functionality to ORBacus using AspectJ. Load balancing code is written in aspect programs to detect server replicas and to redirect requests from the clients to the replicas. Similar functionality is also added to naming servers where repeated naming service requests are intercepted and distributed among replicas.

A conceptual analysis of using aspect is presented by Kienzle and Guerraoui [19], who investigate the aspectization of concurrency and failure in distributed systems. Their analysis indicates that these two properties do lend themselves well for an approach supported by aspects. In our approach, we do not address these two properties, so a comparison is not possible.

Atlas [20] provides one of the earliest case studies of applying aspect-oriented programming as a general architecture approach for building a Web-based learning environment. Atlas uses aspects to support different architectural configurations to implement design patterns and to help the development process by writing modular tracing code. It also presents a notation system for aspects and a set of style rules in designing aspect-oriented systems.

In the middleware field, there are a number of new approaches to improve configurability and adaptability of the middleware platform. Astley et al. [21] achieves middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. Further aspects that crosscut the system implementation are not explicitly addressed. Several projects exploit reflective programming techniques to allow the middleware platform to adapt itself to changing runtime conditions. This includes projects such as openCOM [1], openCORBA [22], and dynamicTAO [2].

Jacobsen and Krämer [23] have developed design patterns to extend middleware platforms with at the interface exposed concerns. This inspired the discussion by Jacobsen [24], [25] who outlined the use of nontraditional programming paradigms for middleware system design.

7 Conclusion

We believe that adaptability and configurability are essential characteristics of middleware substrates. These two qualities require a very high level of modularity in middleware architecture. Traditional software architectural approaches, which we refer to as "vertical decomposition," exhibit serious limitations in preserving the modularity of decomposition models for multiple orthogonal design requirements. Those limitations correspond to the scattering phenomena in the code. The aspect-oriented programming approach has brought new perspectives to software decomposition techniques. The concept of an aspect allows us to compose, with respect to the primary decomposition model, the modular solution for each orthogonal design requirement. Although it is conceptually intuitive that AOP is beneficial to solving the problems of middleware architecture, we still need to apply quantitative analysis to justify our motivation. We perform aspect mining over several legacy implementations of middleware systems. We discover that the scattering of tangled logic is indeed very common as more than 50 percent of classes handle crosscutting logic of some sort. Therefore, if aspect-oriented programming can be successfully applied to modularize the scattered code, the structural complexity of middleware system architecture can be simplified tremendously. As an empirical study, we use AspectJ to refactor a number of aspects identified through the mining process. The implementations, which exist in multiple places of the original code, are grouped within a few aspect units. The successful refactorization shows that middleware systems are able to provide the fundamental services regardless of whether certain pervasive features are factored out or factored in. Aspect-oriented refactorization has shown its superb capability of loading and unloading pervasive features of the system, which is not possible in legacy implementations. The "woven" system transparently supports these refactored features. The runtime performance is equivalent to the original implementation.

In the light of our experimentation, we are very optimistic that aspect-oriented programming will show more promises in conquering the complexity of middleware architectures. It will promote the adaptability and the configurability of middleware systems to an unprecedented level to satisfy a broader range of computing needs. In future work, we will aim at developing more experience in terms of applying aspect-oriented development methodologies to this problem context. We are exploring various techniques to help us define horizontal decomposition procedures more concretely. This experience will be used toward designing a completely new aspect-oriented middleware platform.

References

[1] M. Clarke, G.S. Blair, G. Coulson1, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms,* Nov. 2001.

[2] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Claudio Magalhaes, and R.H. Campell, "Monitoring, Security, and Dynamic Configuration with the Dynamictao Reflective Orb," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing,* 2000.

[3] Cisco Systems, Cisco ons 15327—Sonet Multiservice Platform, http://www.cisco.com/univercd/cc/td/doc/pcat/15327.htm, 2003.

[4] L. DiPalma and R. Kelly, "Applying CORBA in a Contemporary Embedded Military Combat System," *Proc. OMG's Second Workshop Real-Time and Embedded Distributed Object Computing,* June 2001.

[5] "The Common Object Request Broker: Architecture and Specification," Object Management Group, www.omg.org, Dec. 2001.

[6] C. Zhang and H.-A. Jacobsen, "Aspectizing Middleware Platforms," technical report, Univ. of Tortonto, Computer Systems Research Group, CSRG-466, Jan. 2003.

[7] G. Kiczales, "Aspect Oriented Programming," *ACM Computing Surveys (CSUR),* vol. 28, no. 4, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison-Wesley,first ed., Oct. 1994.

[9] R.E. Filman, "Achieving Ilities," *Proc. Workshop Compositional Software Architectures,* Jan. 1998.

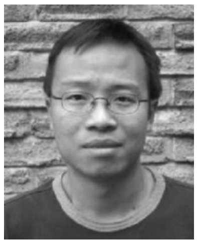
[10] J. Hannemann, "The Aspect Mining Tool," http://www.cs.ubc.ca/~jan/amt/, 2003.

[11] C. Zhang, G. Dapeng, and H.-A. Jacobsen, "Extended Aspect Mining Tool," http://www.eecg.utoronto.ca/~czhang/amtex, Aug. 2002.

[12] M. Robillard, "Feat: A Tool for Locating, Describing, and Analyzing Concerns in Source Code," *Eclipse Technology eXchange (ETX), Proc. Int'l Conf. Software Eng.,* http://www.cas.ibm.ca/conferences/etx/, May 2003.

[13] C. Zhang and H.-A. Jacobsen, "Modularity Analyzer and Aspect Extractor," *Eclipse Technology eXchange (ETX), Proc. Int'l Conf. Software Eng.,* http://www.cas.ibm.ca/conferences/etx/, May 2003.

[14] "IDL to Java Language Mapping Specification,"The Object Management Group, www.omg.org, 2003.
[15] L.H. Rosenberg, "Metrics for Object Oriented Environments," *Proc. Third Ann. EFAITP/AIE Software Metrics Conf.,* Dec. 1997.
[16] "Cyclomatic Complexity," Carnegie Mellon Univ., http://www.sei.cmu.edu/str/descriptions/cyclomatic.html, Sept. 2000.
[17] C. Zhang and H.-A. Jacobsen, "Re-Factoring Middleware Systems: A Case Study," technical report, Univ. of Tortonto, Computer Systems Research Group, CSRG-466, Jan. 2003. Also to appear in *Proc. Int'l Symp. Distributed Objects and Applications,* Nov. 2003.
[18] E. Putrycz and G. Bernard, "Using Aspect Oriented Programming to Build a Portable Load Balancing Service," *Proc. Int'l Conf. Distributed Computing Systems Workshops,* pp. 473-480, 2002.
[19] J. Kienzle and R. Guerraoui, "AOP—Does it Make Sense? The Case of Concurrency and Failures," *Proc. 16th European Conf. Object Oriented Programming,* pp. 37-54, 2002.
[20] M. Kersten and G.C. Murphy, "Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 340-352, 1999.
[21] M. Astley, D.C. Sturman, and G. Agha, "Customizable Middleware for Modular Software," *ACM Comm.,* May 2001.
[22] T. Ledoux, "OpenCorba: A Reflective Open Broker," Lecture Notes in Computer Science, vol. 1616, pp. 197-215, 1999.
[23] H.-A. Jacobsen and B. Krämer, "Design Patterns for Synchronization Adaptors of CORBA Objects," special issue of *L'OBJET J. Object Orientation and Formal Methods,* Hermes Publisher, 2000.
[24] H.-A. Jacobsen, "Middleware Architecture Design Based on Aspects, the Open Implementation Metaphor and Modularity," *Proc. Workshop Aspect-Oriented Programming and Separation of Concerns,* Aug. 2001.
[25] H.-A. Jacobsen, "Re-Thinking Middleware Architecture Design," *Proc. Sixth Biennial World Conf. Integrated Design & Process Technology IDPT,* June 2002.
[26] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Proc. Second Int'l Conf. Aspect Oriented Systems and Design,* Mar. 2003.



Charles Zhang is a graduate student in the Middleware Systems Research Group in the Department of Electrical and Computer Engineering at the University of Toronto. His research interests mainly focus on rearchitecting middleware systems via an effective combination of traditional software decomposition methods and the emerging aspect-oriented programming paradigm. To support such research goals, his research activities mainly include structural analysis of languages, aspect-oriented language design, aspect mining, and aspect refactoring of middleware systems.

Hans-Arno Jacobsen holds a faculty position with the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto. His principal areas of research include middleware systems, distributed systems, and data management. He is a member of the IEEE, the IEEE Computer Society, and the IEEE Communication Society.