Quantifying Aspects in Middleware Platforms

Charles Zhang, Hans-Arno. Jacobsen
Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
10 King's College Circle
Toronto, Ontario, Canada
{czhang,jacobsen}@eecg.toronto.edu

ABSTRACT

Middleware technologies such as Web Services, CORBA and DCOM have been very successful in solving distributed computing problems for a large family of application domains. As middleware systems are getting widely adopted and more functionally mature, it is also increasingly difficult for the architecture of middleware to achieve a high level of adaptability and configurability, due to the limitations of traditional software decomposition methods. Aspect oriented programming has brought us new design perspectives because it permits the superimpositions of multiple abstraction models on top of one another. It is a very powerful technique in separating and simplifying design concerns. In this paper, we first show that, through the quantification of aspects in the legacy implementations, the modularity of middleware architecture is greatly hindered by the ubiquitous existence of tangled logic. We then go one step further by factoring out a number of aspects identified in the mining work and re-implementing them as aspect programs. The aspect oriented re-factorization allows us to apply a set of software engineering metrics to quantify the changes of the re-factored system in both the structural complexity and the runtime performance. The aspect oriented re-factoring proves that the aspect oriented programming 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 a leaner architecture. Further more, the configurability of middleware is dramatically increased because the "aspectized" features can be configured in and out during the compile-time

Keywords

Aspect Oriented Programming, Evaluation and metrics, Distributed systems, Aspect Mining, Middleware, Software Architecture

AOSD 2003 Boston, MA USA

1. INTRODUCTION

In recent years, the adoption of middleware systems such as Web Services, .NET, J2EE and CORBA has dramatically increased. Nowadays, however, the applications of middleware are no longer limited to traditional enterprise computing platforms. A very large family of emerging application domains, such as control platforms, smart devices and networking equipments require middleware to support special computational characteristics such as real-time, stringent resource, high availability and high performance.

The most prominent problem with middleware systems is that the architecture of middleware constantly struggles between generality and specialization. On one hand, vendors want to support many application domains by incorporating a relatively complete set of features in their middleware products. However, these systems usually require large memory spaces and abundant computing resources. On the other hand, middleware architects often want to optimize the architecture to support particular domains with specialized runtime requirements, such as real time, small memory footprint, high availability, and high performance. As a result, for the same technology, there often exist multiple specifications, various branches of code bases, and different implementations. Each of them requires a tremendous amount of effort to maintain the conformity of middleware functionality.

Recent research such as OpenCOM [11] and DynamicTAO [1] deals with the problem by introducing new software engineering techniques like component based architecture and reflection. These new approaches successfully address particular middleware design concerns by using better architectural abstractions to achieve higher levels of modularity and customizability. At the same time, they also have significant drawbacks. For instance, DynamicTAO addresses configurability. However, it has limited means of achieving small memory footprint. OpenCOM achieves adaptability by using meta frameworks which introduce performance overhead.

We recognize that one of the fundamental causes of the problems of middleware architecture is that middleware systems have to support many distinct computational requirements, in addition to distributed computing concerns. We define these additional computational requirements as *orthogonal*

design requirements with respect to the fundamental functionality of middleware systems. The traditional top-down decomposition process produces one decomposition model. A single decomposition model entails that the abstractions of orthogonal design requirements need to be permanently imprinted into the model at some stage of the decomposition process. Traditional modularization methods cannot establish clear boundaries among the implementations of orthogonal design requirements within a single decomposition model.

We think that the phenomenon of handling multiple orthogonal design requirements is in the category of crosscutting concerns, which are well addressed by aspect oriented techniques. Hence, we believe that middleware architecture is one of the ideal places where we can apply aspect oriented programming (AOP) methods to obtain a modularity level that is unattainable 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 the concern crosscutting and to yield a middleware architecture that is more logically coherent. It is then possible to quantify and to closely approximate the benefit of applying AOP to the middleware architecture. This paper contributes to the aspect oriented analysis and design of middleware architecture in the following ways: 1. We show, using aspect mining, that the architecture of middleware inherently suffers from coordinating crosscutting concerns. This paper reports the mining results which quantify the degree of crosscutting. 2. Through the work of aspect mining, we report several new aspects that are specific to the chosen platform. 3. We are the first to perform aspect oriented re-factorization for middleware platforms. AOP based re-factorization is a process of separating certain orthogonal features out, composing them in aspect programs and weaving them back. 4. We quantify the benefit of AOP based re-factorization by applying a set of software engineering metrics. From these metrics, we show that aspect oriented technology lowers the complexity of the architecture, preserves the original design requirements and improves performance as compared to the original implementation.

The rest of the paper is organized as follows: Section 2 provides the background information of middleware and the problems with middleware architecture. Section 3 presents the results and the analysis of aspect mining. Section 4 presents the aspect oriented re-factorization. Section 5 discusses the related work. Section 6 concludes the paper.

2. MIDDLEWARE SYSTEM

Middleware systems consist of a set of services that facilitate the development of distributed applications in a heterogeneous computing environment. A recent striking change in the arena of middleware systems is that the target platforms are no longer limited to traditional enterprise systems and desktop machines. New platforms include mobile devices, network devices, control systems, safety critical systems, and many more. 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[1]. Middleware systems are also adopted by the US Navy as the software bus for subunits in the submarine combat control systems [8].

The widening of the application spectrum has dramatically stretched the capabilities of middleware systems. The characteristics of these application domains differ from each other in significant ways. Many middleware implementations incorporate a large number of features to support a diversified range of target platforms. However, 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 methods to tailor the architecture to fit specific needs at deployment time or at runtime. Another problem is that, 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 domain-specific computational characteristics. For example, even though the role of the object adaptor, an CORBA [4] component responsible for managing servers and dispatching requests, has not changed significantly since the dawn of CORBA, its architecture has gone through a significant evolution in order to support 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. Furthermore, there has been a proliferation of middleware specifications to accommodate different requirements from many application domains. For instance, in addition to CORBA standards, the Object Management Group (OMG) also defines the Real-Time CORBA specification, the Fault Tolerant CORBA specification, and the Minimum CORBA for embedded platforms. The Java platform exhibits the same syndrome, with different types of JVMs for different platforms. The new Microsoft .NET platform is moving into the same direction. Those specifications pose challenges to vendors, who must re-architect the system differently according to a particular specification, as well as to the adopters, who find the platform very complex to comprehend and to use.

One of the fundamental problems in middleware architecture is that software decomposition models obtained using vertical decomposition procedures [14] are incapable of simultaneously modularizing coexisting orthogonal design requirements. Aspect oriented programming, on the other hand, allows us to decompose software systems in different dimensions. We can use the vertical decomposition process to establish the primary decomposition model of middleware, which is optimized for providing transparent network communication facilities. 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.

[1]Cisco ONS 15327. Cisco Systems `http://www.cisco.com/univercd/cc/td/doc/pcat/15327.htm`

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 this hypothesis and to further understand orthogonal design requirements in middleware systems, we need to open up existing legacy implementations and 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 approaches in the next section.

We have picked CORBA implementations as our case study because CORBA has been addressing middleware concerns for over a decade. Its architecture reflects distinct revolutionary cycles in the domain of middleware and can be treated as an excellent case study of the traditional functional composition approach. Because CORBA is an open standard, we are able to achieve a better understanding of its behaviors. There are many open source implementations available for CORBA. All conform to the same OMG standard. That allows us to cross-analyze the results. The core of 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. ORB uses portable object adapters, the interface of which is also standardized, to process invocation requests. ORB uses the General Inter-ORB Protocol (GIOP) as the communication mechanisms on the network to transfer information about the distributed data and operations with other ORBs. The omg:CORBA:ORB interface is a standardized facade [13] to provide abstractions of complicated broker functionalities. For clarity, we use "Orb" to denote the standardized facade interface and "ORB" for the object request broker.

3. ASPECT MINING MIDDLEWARE

In legacy implementations, it is difficult to reason about aspects due to the code scattering problem [6]. However, if the scattering phenomena can be captured, observed and quantified, we have effective means to identify aspects for a particular application domain. Aspect mining consists of the activities that aim at capturing and analyzing the scattering phenomenon in legacy implementations. Before performing aspect mining, it is very important to identify the roles involved in the logical tangling. We consider an 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 define a decomposition model which represents the most fundamental properties of the middleware substrate. We then are able to identify aspects by using that model as the reference to think about tangling in terms of code, and orthogonality in terms of design requirements. We refer to that process as *aspect orientation.*

3.1 Aspect Orientation in Middleware Systems

We think that the fundamental functionality of a middleware system mainly consists of the following major architectural components: 1. A standardized programming model that allows applications to make abstractions of the distributed objects or services. For example, the programming model is supported by IDL in CORBA and MIDL in DCOM and their IDL compilers. 2. The mechanism of publishing the representation of an object or a service to peers. For example, the WSDL files in .NET and IOR file in CORBA represent the remote services. 3. The dispatching mechanism that forwards the requests associated with the published representation to its concrete instance. The dispatching of requests are supported by POA in CORBA and ASP.NET processes in .NET[2]. 4.The commonly agreed representation of data and operations on the network layer in order to exchange information with its remote counterparts. Examples are GIOP in CORBA and SOAP in the .NET platform.

Having identified the primary architecture of a middleware system, we now define *aspects of middleware systems* as abstractions or implementations that crosscut any of those major architectural components enumerated above.

3.2 Extended Aspect Mining Tool

The aspect mining tool (AMT)[3] is designed to capture the code scattering problem by combining the source code of the system with the usage of all class types. We have built the extended aspect mining tool (AMTEX)[4] on top of the original tool to overcome the limitation of visualization based mining. AMTEX provides a much richer set of analytical functionality, such as composing mining activities, managing mining tasks and cross analyzing mining results, to fit the needs of mining very large software systems consisting of thousands of classes.

3.3 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 determining whether these aspects are present. 2. We then apply our definition of middleware aspects to find features in middleware that is orthogonal to the primary functionality of middleware and use AMTEX to capture crosscutting of the corresponding coding structures. 3. We use the ranking functionality 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 provide good hints of potential aspects.

3.4 Mining Results

In this section, we present the mining results for a number of aspects in CORBA implementations. Some aspects have

[2]It is commonly recognized as the responsibility of the xspwp process.

[3]The Aspect Mining Tool. `http://www.cs.ubc.ca/~jan/amt/`

[4]Extended Aspect Mining Tool. `http://www.eecg.utoronto.ca/~czhang/amtex`

been identified prior to this work, such as logging, synchronization, exception handling and pre/post condition checking. We also present new CORBA-specific aspects that are discovered through mining. Those aspects include dynamic programming interface and support for portable interceptors. For each aspect, we report how it crosscuts the primary model using the following sections:
Definition explains the context and the definition of the aspect. **Logic Tangling** discusses the cause for the aspect by trying to identify orthogonal design concerns. **Characterization** is a set of types and textual expressions to represent the implementations of orthogonal design requirements on the code level. This section lists types and expressions used by AMTEX to represent the tangling logic in different ORB implementations. **Mining Results** are collected using AMTEX to quantify the characterization of a particular aspect. AMTEX computes the degree of scattering[5]. The degree of scattering is measured for both Vendor Code Space (VCS)[6] and Complete Code Space (CCS). All results are reported in Table 1. **Result Analysis** provides a brief analysis of the mining data. **AOP Benefit** explains how AOP can theoretically help improving modularity, adaptability and performance of middleware platforms by composing the feature in aspect programs.

The mining data are collected over three open source CORBA implementations: ORBacus, a commercial ORB from IONA Technologies[7]; JacOrb[8], an open source ORB, commercially supported by OCI; and OpenORB[9], a community open source project. All three implementations comply with the CORBA 2.0 specification defined by OMG. For JacOrb, the number of classes in CCS and VCS are 1778 and 579, for ORBacus they are 1777 and 655, and for OpenORB they are 1521 and 287, respectively.

3.4.1 *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. In middleware platforms where the primary programming model is static, the support for dynamic programming model crosscuts the entire architecture.

Characterization: In CORBA, the OMG defined objects that handle DSI/DII are the following classes in the Java 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:

a. All Helper and holder classes[10] contain operations to allow transformation and manipulation of these server objects in the dynamic invocation context.

b. The Orb interface supports dynamic invocations by providing the functionality of composing operations, their parameters and their return types.

c. The request processing classes support dynamic invocations through specialized request objects, such as `org.omg.CORBA.Request` and `org.omg.CORBA.ServerRequest`, and also through extra control logic.

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.

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 deal with DII or DSI in all three implementations. The degree of the spreading for the dynamic programming model increases as the code size increases.

AOP Benefit: Aspect oriented programming methodology can be applied here to separate the dynamic invocation model from the static invocation model. That is, we can write an aspect program for the static model to support the dynamic model, or vice versa. Using AOP, we can make a number of significant improvements to 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, modulated and optimized. Secondly, since in most cases only one invocation model is sufficient for a particular domain application, implementing the dynamic programming interface in aspect programs gives us the option of either "weaving" the feature in or out. The ORB architecture can become more configurable, adaptive and computationally efficient.

3.4.2 *Portable Interceptors*

Definition: Portable Interceptors are hooks into the ORB, through which CORBA services can intercept various stages of the request process. They are observer [13] style entities. Interceptors allow third parties to plug in additional ORB functionalities such as transaction support and security.

Characterization: The implementation for supporting interceptors can be characterized by the usage of the interceptor class types such as `org.omg.IORInterceptors` and related class types such as `PIManager`.

[5]A measure of percentage of the usage of the characterization set throughout the application code. AMTEX measures the total number of classes in an application, and also the number of classes that contains any class types in the characterization set. The ratio of these two numbers is the degree of scattering.
[6]Classes defined and implemented by the vendor, other than those specified by OMG
[7]ORBacus, `http://www.iona.com`
[8]JacOrb, `http://www.jacorb.org`
[9]`OpenORBhttp://openorb.sourceforge.net`
[10]Helper and Holder classes are defined in the IDL to Java mapping specification standardized by OMG, all user defined IDL types should be generated with additional Helper and Holder classes to allow convenient manipulation of these types.

Implementation	DPM (*ccs* | *vcs*)		PI(*ccs* | *vcs*)		Error(*ccs* | *vcs*)		Pre/Post(*vcs*)	Logging(*vcs*)	Synch(*vcs*)
JacOrb	31.56%	23%	0.51%	3.52%	25.8%	46.5%	13.3%	14.66%	14.11%
ORBacus	31.2%	26.56%	2.66%	7.09%	39.3%	45.5%	5.47%	18.9%	9.16%
OpenORB	32%	23%	2.44%	13%	35.3%	44.6%	10.4%	16.3	9.41%

Implementation	*Total number*	*Crosscut by one aspect*	*Ratio*	*Crosscut by three aspects*	*Ratio*
Jacorb	563	322	57.2%	60	10.7 %
OpenOrb	270	149	55.2%	43	15.9 %
ORBacus	621	335	53.9%	80	12.8 %

Table 1: Degree of Scattering for DPI(Dynamic Programming Interface), Portable Interceptor(PI), Error Handling, Pre/Post Condition Checking, Logging, Synchronization, and all the aspects combined

Logic tangling: The specification for interceptors is added to the ORB architecture after the basic functionality of the ORB has been defined. The support for the interceptors is incorporated in the implementation of the POA and other objects that are directly responsible for request processing as follows:
a. The Orb interface contains methods and data members for registering interceptors. It also provides methods to notify these interceptors upon reaching interception points.

b. During the propagation of the request, the invocation context is checked to see if it is modified by interceptors.

c. The POA needs to bundle requests with the information of interceptors, if there are interceptors registered with the ORB.

The number of POAs, is proportional to the number of server objects in the domain applications. Therefore, the effect of scattering could be amplified during runtime.

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 very attractive for other application domains that are unwilling to pay for the cost of managing interceptors.

3.4.3 Common Aspects

We have also performed aspect mining on "common" aspects, such as error handling, logging, synchronization and pre/post condition checking. We term them common aspects since these aspects are not middleware specific and discussed extensively in previous literature. Our mining work confirmed that, like in other software systems, the architecture of middleware is tangled with these concerns as well. Due to space limitation, we present the mining data in Table 1. More detailed analysis of mining for these aspects can be found in the full version of this paper.

3.4.4 Cross Comparisons of Mining Results

The sections above have listed the mining results of six aspects. We use AMTEX to combine the individual results, which yields some statistical information that illustrates the scattering in CORBA platforms from a slightly different angle. The Table 1 shows, in all of the three CORBA implementations, more than 50% of the classes are crosscut by at least one aspect. 10% to 15% of the classes contain three or more aspects.

These measures strongly indicate that scattering of tangling implementations in middleware systems is consistent regardless of whether architecture style is cathedral or bazaar[11]. The vertical decomposition process yields architecture models that inherently suffer from coordinating orthogonal design requirements.

4. AOP RE-FACTORING OF MIDDLEWARE

The aspect mining has shown that the crosscutting of aspects is an inherent property of middleware implementations. However, it is still not clear how aspect oriented programming techniques can improve the middleware architecture while preserving the design requirements. 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. Since the aspect-centric middleware architecture is still our long term research goal, we have taken the approach of re-factoring legacy implementations using AOP. That is, if we identify certain functionality as an aspect from the result of aspect mining, we should be able to separate that functionality from the original implementation. Subsequently, we should be able to compose the functionality in aspect languages and to weave it back into the architecture. All design requirements should still be satisfied transparently to the user. We believe that "AOP re-factoring" closely resembles what happens in a completely aspect oriented approach. The following sections present the set of the software engineering metrics we use to quantify the architectural differences. We then describe our re-factoring approach, the detail of the aspect oriented implementation and the results of the measurements.

4.1 Quantification Metrics

Software metrics are measures for the quality of software designs. We think it is appropriate to use a combination of metrics to address various properties of re-factored architectures, including both the static properties, which directly relate to the cost of development and maintenance,

[11] `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`

and the runtime characteristics, which reflect the cost of adopting the technology. We measure changes of the primary program as the result of re-factorization, as aspect programs are maintained separately. We recognize that the sum of the complexities of both the primary program and aspect programs might not necessarily decrease. However, the complexity is managed by AOP compilers and not of the architect's concern. The following is a list of metrics that are used to quantify the re-factorization. For each metric, we also provide analysis of the expected change due to the re-factorization and the rationale of the change.

Cyclomatic complexity number (CCN) [9]. CCN is a measure of the alternative execution paths in code segments caused by control flow statements. It is an index obtained through heuristics and is independent of specific languages[12]. A lower CCN, e.g. 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 hits and more dynamic optimization opportunities. AOP re-factoring should lower the cyclomatic complexity because it takes the handling of crosscutting properties out from the primary decomposition.

Size. The size of a software system directly ties to the development and maintenance cost. We define size as the total number of executable lines in all measured class source files. AOP re-factoring decreases comments and blank lines, as well as the count of executable statements of the primary program.

Weight of Class [2]. The weight of a class is the average number of methods per class. It reflects the complexity of the class. AOP re-factoring decreases the weight of the class.

Coupling between classes. We measure coupling as the average number of classes that a class has as attributes or invokes upon. Coupling indicates how much the types in the system are related to each other. Good architecture is always less coupled due to good modularization. AOP re-factoring decreases coupling by separating the primary program from the knowledge of the types which implement the crosscutting logic.

Response time. In the context of middleware, response time can be defined as the time taken to respond to an invocation request by the ORB. This is defined as the total time a message takes to traverse through the middleware stack during a roundtrip. We divide the response time into four intervals: **Interval A:** Client side marshalling; **Interval B:** Server side unmarshalling and dispatching; **Interval C:** Server side marshalling;**Interval D:** Client side unmarshalling.

It is necessary for the aspect oriented re-factoring to at least preserve the runtime performance measure. In addition, with crosscutting features factored out, the re-factorization is expected to decrease the processing time due to the simplification of logic.

To ease our discussion, we classify these metrics into two categories: structural and behavioral. Structural metrics include cyclomatic complexity, code size, weight and coupling. Behavioral metrics reflect the runtime characteristics of the system and include the response time.

The structural metrics are collected over classes that are involved in the re-factorization. We use JavaNCSS[13], an open source metric collection tool, to compute cyclomatic complexity, average class size and weight. We use AMTEX to collect the coupling index. To measure the performance, we use a simple C based timing tool together with java native interfaces. We write timing programs in AspectJ and 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 1GHz Linux 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 operations.

4.2 Re-factoring 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, one of the CORBA implementations used previously for the aspect mining, as the component program. To verify the correctness of the re-factorization, we adopt 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 re-factorization. The test programs are also used for performance measurements. As the first step of the re-factorization of the ORBacus implementation, we need to identify the presence of each crosscutting property in two forms: the implementation structure for the property, and the crosscutting points in the primary decomposition model for that property. The tangled code is transformed into three types of class groupings in the aspect oriented implementation, namely primary classes, aspect implementation classes and weaving classes. The transformation can be illustrated by Figure 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.

4.3 Aspect Oriented Re-factoring

In this section, we will present our re-factoring 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

[12]CCN 1-10 means simple programs without much risk. 11-20 means a program has moderate risk and 21-50 means high risk. Software Engineering Institute. `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`

[13]`http://www.kclee.com/clemens/java/javancss/`

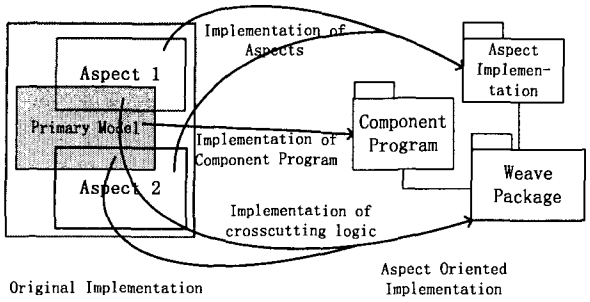

Figure 1: Code transformation for Aspect oriented Re-factorization

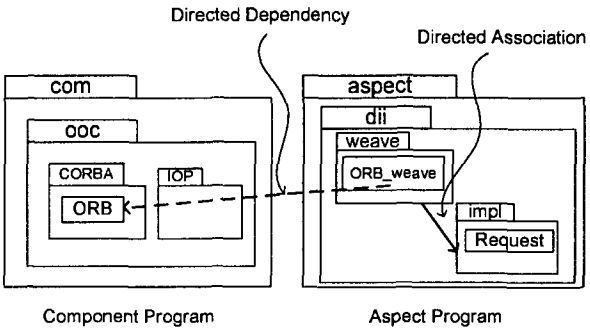

Figure 2: Package Organization for Aspect oriented Re-factorization

construction of the package scheme, which is described in the previous section. We use the package diagrams in Figure 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 measurements of the metrics as the result of factoring out that specific feature from the ORBacus implementation. We also discuss the limitations of our aspect oriented implementation. 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, we use the term "Client" to denote the role that is requesting a service, and the term "Server" the role for providing that service, with the observation that, in the middleware context, the classification of client and server is arbitrary. Again, in this version of our paper, we omit the discussion and only present the measurements of the logging aspect. The detailed description of the entire re-factorization work is presented in [15].

4.3.1 Dynamic Programming Interface

The aspect mining reveals that the dynamic programming interface can be treated as an aspect of the CORBA implementation, which primarily supports stub and skeleton based static invocation methods. Our AOP based re-factoring of the dynamic programming interface consists of two parts, the client side Dynamic Invocation Interface (DII) and the server side Dynamic Skeleton Interface (DSI).

4.3.1.1 Dynamic invocation interface (DII)

a. *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`. Those two class types are taken out of the original implementation and grouped under the aspect implementation package for DII.

b. *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 picked out in step one. Those 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. We group the aspect classes that contain implementations of these "*joinpoints*" in the *weave* package.

1. 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.

2. The "*Introduction*" construct is used to factor out the downcall creation logic for dynamically composed downcalls.

3. "*Introduction*" is used to factor out the code in the object delegates [14], which creates dynamic invocation request objects.

c. *Metric comparison.* Table 2 presents the measurements of both structual metrics and the performance speed. The structural metrics are collected on the ORBacus implementation prior to AOP re-factoring and after 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 DII is same as our prediction. The runtime performance of DII is equivalent to that of the original implementation.

4.3.1.2 Dynamic skeleton interface (DSI)

a. *Aspect Implementation.* The server side facility for 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.

b. *Crosscutting points.* We implement the crosscutting points of the aspect DSI in the primary decomposition model as follows:

1. We first remove the code segments which dispatch client requests to a dynamic server implementation from the request dispatching code. Then we used the *"around"* construct to replace the request dispatching call with an alternative implementation, which appropriately handles dynamic server implementations.

2. ORBacus prohibits direct invocation of DSI server implementations. We moved the logic of checking whether an

[14] Delegate is a standard Java interface defined in the OMG IDL to Java mapping

	Structural Metrics				Runtime Interval			
	ACC	*Size*	*CW*	*CI*	*A*	*B*	*C*	*D*
Dynamic Invocation Interface								
O	5.08	1559	40	40.67	105	125	37	59
R	4.76	1490	37.67	39.33	108	124	35	59
Dynamic Skeleton Interface								
O	3.64	274	9.33	14	79	126	43	8
R	3.46	262	9.33	13.5	76	119	41	9
Portable Interceptor Support								
O	4.11	3016	26.8	34.75	78	118	42	8
FO	4.0	2909	26.7	32.38	79	122	42	9
Collocation Invocation								
O	5.22	449	11.33	21	79	126	43	8
FO	5.00	435	10.67	19.33	76	126	41	7
Logging								
O	3.93	2453	14.25	36.75	79	126	43	8
R	3.46	2275	12.52	35.25	76	119	41	9
Overall								
O	4.2	5393	16.50	21.13	76	126	37	9
FO	4.04	4899	16	18.33	76	123	37	9
R	n/a	n/a	n/a	n/a	74	123	37	8

Table 2: Metric Matrix for the factorization of DII,DSI, Portable Interceptor, Collocation Invocation, and overall (CW: Weight of Class; CI: Coupling Index; O: Original ORB; R: Re-factored ORB; RO:ORB with aspects factored out).

invocation is towards a dynamic or static implementation into the aspect implementation. The *"before"* construct is used to precede the normal invocation process in order to prevent direct invocation.

c. *Metric comparison.* The data show that factoring out DSI has simplified the control flow and decreased the class size. The average weight of classes does not change because server side support for dynamic programming interface is much simpler. No additional methods are used to support DSI in the original implementation. We use the static invocation interface on the client side. Therefore, the client-side process time, interval A and interval B, dramatically decreases as compared with DSI. However, that change is irrelevant to our AOP re-factorization.

4.4 Support for Portable Interceptors

a. *Aspect Implementation.* In ORBacus, the functionality of portable interceptors is implemented by three categories of classes. They include the classes related to implementing the interceptor interfaces defined by OMG. They also include ORBacus specific interceptor initialization classes and request processing classes that support portable interceptors. We separate classes in these three categories from ORBacus and grouped them under the aspect implementation package.

b. *Crosscutting points.* We implemented 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 the portable interceptor. That is, an ORB implementation must allow interceptions of the client request process, the server request process, and the creation process of server objects. The following is a summary of our AOP implementations.

1. The portable interceptors allow ORB requests to be intercepted before they are sent. Therefore, in ORBacus, the request sending process (i.e. the downcall creation process) needs to check if any client request interceptors are registered. Instead of performing the checking regardless of whether portable interceptors are used, we moved the code segments into the aspect program in an "around" construct. As a result, the "aspectized" ORBacus only performs checks if portable interceptor is required for a particular application.

2. A similar situation occurs in the server side request dispatching process (i.e. the upcall creation process). We move the checking code and upcall creation code into the aspect implementation.

3. The portable object adaptor (POA) plays a key role in the process of object creation. It needs to notify all the interceptors registered for intercepting object creation process. 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.

4. The ORB also contains the initialization code for loading portable interceptors and registering them with the ORB. We moved the corresponding code into the aspect implementation so that, if interceptor support is not needed, it is not necessary for the ORB to perform the extra initialization procedures.

c. *Metric comparison.* 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 structural metrics show the same direction of change as in the case of the dynamic programming interface. The coupling factor exhibits a bigger drop because interceptor support is implemented by a larger number of classes. The runtime performance is equivalent to the original implementation.

4.5 Invocation of Collocated Objects

a. *Background.* 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 scenes. 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 invoking in-process server objects in ORBacus is logically orthogonal to, and conflicting with its remote invocation mechanism. Therefore, we identify the optimization for local invocations as an aspect of ORBacus. Since currently we treat it as an ORBacus specific phenomenon, we defer the corresponding aspect mining analysis to our future work.

b. *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 moved them into the aspect package.

c. *Crosscutting points* In ORBacus, the collocation invocation is mainly implemented in the object initialization phase for both the client and the server. We have used AspectJ to re-implement the collocation invocation in aspect programs as follows:
1. We use the *"after"* construct to create the server-side objects that are responsible for processing collocation invocation after the objects for servicing remote invocations are created.

2. 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 so, a different communication model is set up to avoid marshalling and network operations.

d. *Metric comparison.* The response time is measured by running collocated client/server communications on the original ORBacus and the "aspectized" ORB. The time taken to traverse the middleware stack is much shorter than original invocations because of the optimization mechanism in ORBacus. The re-factorization collocation invocation also indicates no performance overhead. We observe a slight improvement on the client side (interval A,D) due to the simplification of the downcall procedures.

4.6 Overall Assessment and Analysis

We have presented comparative results with respect to each individual feature that is re-factored using AspectJ. We are also interested in the overall effect of the collective re-factor ing 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 handling remote object invocations. We use the same experiment settings as before. 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 data indicate that, while implementing the primary functionality of and ORB, we can significantly lower the complexity of the architecture via the aspect oriented approach. The data show that the same functionality can be implemented by 9% less code, 0.5 fewer methods per class, and 3 fewer instances in terms of coupling with other classes per class.

From the observation of the response time measurements for the overall re-factoring as well as the individual ones, we can make the conclusion that "aspectized" architecture is equivalent to that of the original architecture. The difference is 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 executes these code segments anyway.

4.7 Limitations

During our aspect oriented re-factoring of ORBacus, we have also 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 the request context passing. While failing to factor these types out does not prevent us from evaluating the aspect oriented approach, we defer the exact quantification of the aspect of dynamic programming interface to future work.

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 to future work. 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 the non-existence of these features.

3. We decided not to collect memory usage statistics since our "aspectization" experiment is conducted on the Java platform. We do not have an accurate memory profiling tool that allows us to monitor memory usages of the application objects. Also the expense of running the full JVM makes the memory improvements achieved by our AOP re-factoring almost trivial.

5. RELATED WORK

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

Bernard and Putrycz [3] 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 these replicas. Similar functionality is also added to naming servers where repeated naming service requests are intercepted and distributed among replicas.

Murphy and Kersten [10] provide one of the earliest case studies of applying aspect oriented programming as a general architecture approach to building a web based learning environment. Atlas uses aspects to support different architectural configuration, 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 also a number of new approaches to improve configurability and adaptability of the architecture. Astley [12] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol compo-

sition. 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 [11], openCORBA [7], and dynamicTAO [1]. Recent progress in this area has been summarized in a reflective middleware workshop[15].

In [5] we have outlined the use of non-traditional programming paradigms for middleware system design.

6. 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 over 50% 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 re-factor 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 re-factorization 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 re-factorization 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 re-factored 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 promise in conquering the complexity of middleware architecture. It will promote the adaptability and the configurability of middleware systems to an unprecedented level to satisfy future computing needs. In our future work, we will try to gain more experience in applying aspect oriented development methodologies. We are exploring various techniques to help us define horizontal decomposition procedures more concretely. We will eventually use all this experience to design a fully aspect oriented middleware platform.

[15]Reflective middleware workshop 7th-8th, April, 2000 `http://www.comp.lancs.ac.uk/computing/rm2000/`

7. REFERENCES

[1] Fabio Kon Manual Roman Ping Liu Jina Mao Tomonori Yamane Luiz Claudio Magalhaes Roy H. Campell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb,. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2000.

[2] S. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.

[3] Guy Bernard Erick Putrycz. Using aspect oriented programming to build a portable load balancing service.

[4] Object Management Group. The Common Object Request Broker: Architecture and Specification. December 2001.

[5] Hans-Arno Jacobsen. Middleware architecture design based on aspects, the open implementation metaphor and modularity. Workshop on Aspect-Oriented Programming and Separation of Concerns, August 2001. Lancaster, UK.

[6] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[7] T. Ledoux. OpenCorba: a reflective open broker. *Lecture Notes in Computer Science*, 1999.

[8] Robert Kelly Louis DiPalma. Applying corba in a contemporary embedded military combat system. OMG's Second Workshop on Real-time And Embedded Distributed Object Computing, June 2001.

[9] Arthur H. McCabe, Thomas J. Watson. Software complexity. *Crosstalk, Journal of Defense Software Engineering 7*, (12):5–9, December 1994.

[10] Gail C. Murphy Mik Kersten. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. *ACM*, 1999.

[11] Clarke M. Blair G. Coulson G. Parlavantzas N. An efficient component model for the construction of adaptive middleware. *IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, November 2001.

[12] M. Astley D.C. Sturman and G. A. Agha. Customizable middleware for modular software. *ACM Communications*, May 2001.

[13] Erich Gamma Richard Helm Ralph Johnson John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[14] Charles Zhang and Hans-Arno Jacobsen. Aspectizing middleware platforms. Technical Report CSRG-466, University of Toronto, January 2003.

[15] Charles Zhang and Hans-Arno Jacobsen. Re-factoring middleware systems: A case study. Technical Report CSRG-465, University of Toronto, January 2003.