Re-factoring Middleware Systems: A Case Study

Charles Zhang and 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. Aspect oriented programming brings us new design perspectives since 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 provide detailed descriptions of our aspect oriented re-factoring of ORBacus, an industrial strength CORBA implementation. The re-factored features are the dynamic programming interface, support for portable interceptors, invocations of local objects. Their associated IDL-level re-factorization is addressed by an aspect-aware IDL compiler. In addition, we present the quantification for the changes in terms of both the structural complexity and the runtime performance. The aspect oriented re-factorization proves that AOP is capable of composing non-trivial functionality of middleware in a superimposing manner. The final "woven" system is able to correctly provide both the fundamental functionality and the "aspectized" functionality with negligible overhead and leaner architecture. Furthermore, the "aspectized" feature can be configured in and out during compile-time, which greatly enhances the configurability of the architecture.

1 Introduction

In recent years, the adoption of middleware systems such as Web Services, .NET, J2EE and CORBA 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 constraints, high availability, and high performance. For example, middleware systems are used on the Cisco ONS 15454 optical transport platform to manage hardware customizations and communications among the management software and hardware drivers [14]. Middleware is being used as the software bus for subunits in the submarine combat control systems by the US Navy [4].

The fast broadening of the application spectrum has brought many difficult challenges to the design of middleware. We observe that one of the most prominent problems is that the architecture of middleware constantly struggles

between two conflicting goals: generality and specialization. Generality means vendors desire to support as many application domains as possible by incorporating a large set of features in their middleware implementations. As direct consequence, these systems usually require large memory spaces and abundant computing resources. For example, ORBacus [15], one of the Java implementations of CORBA [8], requires around 7 megabytes of memory[1]. The C-based CORBA implementation ORBit [7] requires at least 2MB of memory space. Therefore, it is very expensive to deploy these types of middleware systems on many handheld devices or wireless devices. This is because, for example, commercial handheld devices typically support memory size of a few mega-bytes with limited processor power. [2]. The computation resource in most cell phones is even more constrained[3].

To accommodate these computing environments with stringent resource constraints and special runtime requirements, middleware architects often choose to specialize the architecture of middleware in order to optimize its performance for domain specific characteristics such as real time, small memory space, 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 these implementations require a tremendous amount of effort to develop and to maintain. It is a challenge for the vendor to ensure that the distributed computing properties are consistent across many different versions of the same technology. It is also a challenge for users of middleware to well understand the differences and, although not always possible, to match specific implementations with their specific needs.

Recent research such as OpenCOM [12] and DynamicTAO [10] mainly aim at improving the configurability and the adaptability of middleware by introducing new software engineering techniques like component based architecture and reflection. Astley *et al.* [3] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. LegORB [13] and Universally Interoperable Core (UIC)[4] are middleware platforms designed for hand-held devices, which allow for interoperability with standard platforms. Both offer static and dynamic configuration and aim to maintain a small memory footprint by only offering the functionality an application actually needs. The QuO project at BBN Technologies constitutes a framework supporting the development of distributed applications with QoS requirements (see [11] for an example). QuO supports a number of description languages, referred to as Quality Description Languages (QDL). The QDLs are used to specify client-side QoS needs, regions of possible level of QoS, system

[1] This includes the JVM memory footprint. Classes of ORBacus take more than 4MB of memory. This is estimated by comparing the size of ORBacus runtime with a simple Java program.

[2] The new Palm M515 devices support 8M of memory and operate at 33MHz. `http://www.palm.com/products/palmm515/m515ds.pdf`

[3] Cypress corporation predicts in 1999 that newer cell phones would have SDRAM of 4M. See *MoBL: The New Mobile SRAM. Cypress Whitepaper*

[4] http://www.ubi-core.com/

conditions that need to be monitored, certain behavior desired by clients, and QoS conditions [11]. Further extensions of these languages are envisioned to also be able to define available system resources and their status. Loyall *et al.* [11] interpret these different description languages as aspect languages that are processed by a code generator to assemble a runtime environment supporting the desired and expected quality of service by client and server in a distributed application. Zinky *et al.* [18] further elaborate on the issue of adaptive middleware code that cross-cuts the platform's functional decomposition. It is illustrated that aspect orientation could be used to manage the QoS of a connection in a distributed application. The QuO approach to specifying QoS guarantees is very powerful. However, the focus in the QuO project lies on managing communication QoS, which are important aspects for distributed applications, but QuO dose not address the problem of re-factoring a legacy middleware platform to make it configurable and customizable for a particular application domain or even application, which is the focus of our work.

Our approach differs from the afore described work as we believe that it is more concrete and effective to study the benefit of applying AOP to the legacy architecture of middleware. This is because, as AOP claims, conventional decomposition methods cannot modularize crosscutting concerns and, therefore, cause a considerable degree of logic tangling and concern scattering. Following this theoretical conjecture, we first provided quantification of aspects in legacy middleware systems [16, 17]. We proved, through the method of aspect mining and aspect oriented re-factorization, that concern crosscutting is an inherent problem in CORBA-based middleware systems implemented by conventional means. In this paper, we complement our previous work by presenting an architectural view of this aspect oriented re-factorization work. We describe, in UML diagrams and code examples, how a number of non-trivial internal features of middleware are captured in a separate set of aspect modules. We also present a prototype of the aspect-aware interface definition language (IDL) compiler, which performs the aspect oriented re-factorization at the stub/skeleton generation stage.

The rest of the paper is organized as follows: Section 2 introduces the new language features of AspectJ [1], the aspect oriented language we use to perform the re-factoring. Section 3 presents a detailed description of building four major CORBA features using AspectJ for the ORBacus implementation. It also includes the evaluation, which reflects both architectural changes and the performance effects of the aspect oriented implementations. In contrast to [16], we, here, present aggregated results to quantify the total sum of changes and to illustrate the benefit of our approach in a different perspective.

2 Aspect Oriented Programming and AspectJ

Aspect oriented programming offers an alternative design paradigm, which achieves a very high degree of *separation of concerns* in software development. "Aspects tend not to be units of the system's functional decomposition, but rather be properties that affect the performance or semantics of the components

in systemic ways." [9] Examples of such properties include security, reliability, manageability, and more [5]. The existence of aspects is attributed to handling crosscutting concerns using the traditional "vertical" decomposition paradigms. AOP overcomes the limitations of traditional programming paradigms by providing language level facilities to modularize these systematic properties as separate development activities. The AOP compiler is capable of producing the final system by merging the aspect modules and the primary functionalities together. We employ the following AOP artifacts to address problems in the middleware design.

Component language. A component language is used for performing the primary decomposition. It can be any regular programming languages such as Java or C.

Aspect language. The aspect language defines logic units that can be used to compose aspects into modules. Representative aspect languages are AspectJ [1] and Hyper/J [2]. We can use these languages to implement crosscutting concerns.

Aspect weaver. The responsibility of an aspect weaver is to instrument the component program with aspect programs to produce a final system. In the context of middleware architecture, the implementations of both the core functionality of middleware and the features as aspects can be defined separately and coexist in the final "woven" system.

There are a number of aspect-oriented languages. Hyper/J supports multi-dimensional programming by allowing programmers to compose the system differently according to specific concerns in Java. The HyperJ compiler performs bytecode transformations to generate different final systems according to extraction specifications. Each extraction is analogously termed as "hyperslicing". AspectJ [1], designed as an extension of the Java language, is a mature aspect oriented programming language. AspectJ provides "`pointcut`" constructs to designate a collection of interception points in the execution flow of software systems. AspectJ also provides method-like constructs called advices, such as "`before`", "`after`", and "`proceed`". These constructs can contain normal Java code, which gets executed before, after, or in place of the interception points designated by the "`piontcut`" constructs. It also contains *inter-type declarations*, also called *introduction*s, which are used to declare new members (fields, methods, and constructors) in other types. In the later sections, we illustrate in detail how these special constructs can be used to re-factor crosscutting concerns in middleware systems.

3 Aspect Oriented Re-factorization of CORBA

We have chosen CORBA as our case study because CORBA has been addressing middleware concerns for over a decade. Its architecture reflects distinct evolution cycles in the domain of middleware and can be treated as an excellent example of traditional decomposition approaches. CORBA is a long term standardization

effort by OMG[5]. We use the ORBacus CORBA code base as our case study. ORBacus is an industrial-strength and open source CORBA implementation. The version used for the re-factoring is 4.1.1. It follows the Open Connector Interface (OCI) architectural model, which provides further standardization of the internal structures of the Orb.

In this section we use a number of software engineering metrics to track the changes resulting from re-factoring the ORBacus code base with aspects. We present the detailed re-factoring of the following ORBacus features: dynamic programming interface, support for portable interceptors, and invocation of local servers. To support the re-factoring at the IDL and stub layer, we describe an AOP-based design of an aspect-aware IDL compiler. We then present the quantification as the result of factoring out specific features from the ORBacus implementation. We also discuss the limitations of our aspect-oriented implementation.

3.1 Quantification Metrics

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 the "aspectized" architecture, including both the structural metrics, which directly reflect the cost of development and maintenance, and the runtime metrics, which reflect the cost of adopting the technology. The structural metrics include cyclomatic complexity, size, weight of class, and coupling between classes. Cyclomatic complexity is an index which measures the complexity of the control flow in a program. The size measures the number of lines of executable code. The weight of class refers to the number of methods in a class definition. The coupling metric measures the number of references to other classes in a particular class. please refer to [16] for a detailed discussion and the collection method of these metrics. To measure the response time of the broker, we divide the total time for the roundtrip of a request 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 response time of the broker. In the case of having crosscutting features factored out, AOP re-factorization is expected to decrease the processing time due to the simplification of program logic.

3.2 AOP Based Performance Measurement

Each of the four intervals in the traversal of the middleware stack requires measurements taken at many different points in the execution path of ORBacus. To avoid changing the ORBacus code for these different measurements, we write the timing code in Java and define four sets of pointcuts in AspectJ. To obtain high-resolution time, we use a simple C-based timing tool written in Java Native

[5] Object Management Group. `http://www.omg.org`

Interfaces. Since the instrumentation code for time measurement is nicely captured in one module, it also becomes convenient to perform more advanced statistical analysis of the response time. The inserted calls to aspect methods incur slight performance overhead in the order of a few microseconds. This overhead is eliminated when performing comparative analysis. To verify the correctness of the re-factorization, we adopted the demonstration code, which is a part of the standard ORBacus source distribution as test cases as well as for taking performance measurements. The re-factored Orb is transparent to the test programs. 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.

3.3 Code Transformation

As the first step in the re-factorization, we need to identify, before re-factorization, the presence of a particular crosscutting property in two forms, the *implementation structure* of the property and the crosscutting structure for that property with the primary decomposition model. Therefore, the tangled code is transformed into three class groupings in the AOP implementation, namely primary classes, aspect implementation classes, and the weaving classes. The transformation is 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 among 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. We use the package diagrams in Figure 1 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.

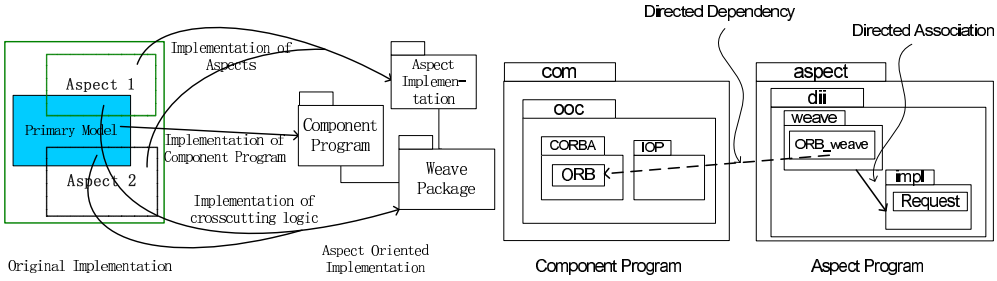

Fig. 1. (1) Code transformation for re-factoring. (2) Package organization for re-factorizat

3.1 Dynamic Programming Interface A dynamic programming model allows an application to be designed without prior knowledge of the interface definitions of the invoked objects. Instead, invocations on a remote interface can be composed during runtime. In middleware platforms, where the primary programming model is static, the support for the dynamic programming model crosscuts the entire architecture. Our AOP based re-factoring of the dynamic programming model consists of two parts, the client-side Dynamic Invocation Interface (DII) and the server-side Dynamic Skeleton Interface (DSI).

Dynamic invocation interface (DII) 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. 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. These places are the crosscutting points of the DII aspect. In AspectJ, these crosscutting points can be implemented as "*join-points*" instead.

Special note on UML: Since UML has yet no direct support for AOP notions, we model an "aspect" as a regular class. We model an `advice` as a regular class method. We model "Introduction" constructs as regular attributes and methods. Their names are prefixed with the names of the classes within which these attributes and methods are declared. Due to the special construct of `advice`, most UML tools would generate some oddities on the diagram.

Figure 2 presents the UML diagram of the aspect implementation of the DII. As a concrete mapping of Figure 1, the AOP implementation involves three packages. The primary program package on the left represents the original implementation of the ORB objects with the logic of the DII removed. The DII code is placed in the package on the right as the aspect implementation. The package organization of these classes is left intact. The package in the middle of the diagram includes the "weaving" modules which define how the aspect implementation of the DII interacts with the primary program. The "weaving"

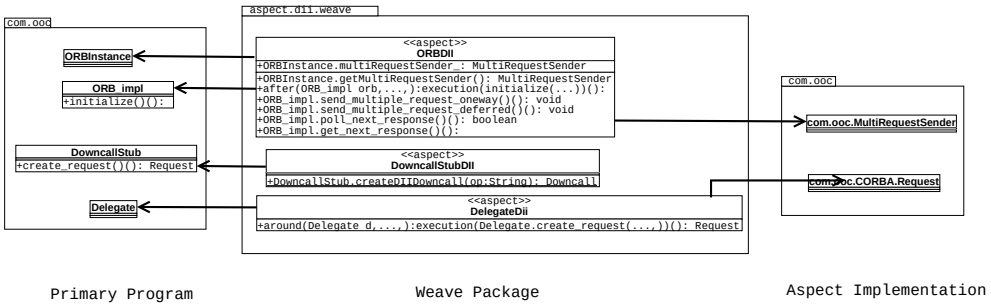

Fig. 2. UML Diagram of The DII Aspect Implementation

modules in the UML diagram shows that the aspectization of the DII involves four classes in the Orb, namely, `ORB_impl`, `ORBInstance`, `DowncallStub` and `Delegate`. To interpret the diagram for the `aspect.dii.weave` package, we use the aspect module `ORBDII` as an example. In the aspect module `ORBDII`, an extra field `MultiRequestSender` and an additional method `getMultiRequestSender` are added to the class `ORBInstance` to support the sending of multiple DII requests. Extra code is executed after the execution of the `initialize` method of the `ORB_impl` class to perform DII specific initializations during the ORB start-up time. Four DII related methods are also declared in the class `ORB_impl` to support DII operations. In other aspect modules, we use the "*inter-type declarations*" to inject the downcall creation logic for dynamically composed downcalls. We use "`around`" to change the behavior of the request creation in the `Delegate` class.

Figure 3 shows a major fragment of the aspect module ORBDII responsible for sending multiple DII requests. In this code snippet, lines 7-12 declare a new attribute and a new method to support multiple DII request sending in class `ORBInstance`. Lines 14-19 create the runtime instance of the new attribute `multiRequestSender_` "after" the initialization work of `ORB_impl` finishes. Lines 21-28 enable the DII multiple request sending capability of `ORB_impl` by adding new application programming interface (API) `send_multiple_requests_oneway`.

Dynamic skeleton interface (DSI) The server side facility for the dynamic programming model is supported through the ORBacus implementations of the OMG interfaces including `ServerRequest` and `DynamicImplemenation`. We first remove these two class types and group them under the aspect implementation package. Figure 4 presents the organization of the classes for the AOP implementation of DSI. As in the case of DII, the "`aspect.dsi.weave`" package defines how DSI implementation is added back to the regular ORB implementations. This package identifies the crosscutting points which are implemented as follows:

1. We used the *"around"* construct to replace the request dispatching call with an alternative implementation which dispatches client requests to a dynamic server implementation.
2. ORBacus prohibits the direct invocations for DSI server implementations. We use the *"around"* construct to check whether an invocation is towards a dynamic implementation preceding the normal invocation process in order to prevent direct invocation.

To illustrate how DSI is implemented, we present the complete AspectJ code in figure 5 for weaving the checking logic into the class `ActiveObjectOnlyStrategy`, an activity described previously in the second item.

```
package aspect.dii.weave;                                                    1
//imports are omitted                                                        2
privileged aspect ORBDII                                                     3
{                                                                            4
  //introduce a new field multirequest sender in ORBInstance.This field is   5
  //initialized by ORB_Impl,which is executed before ORBInstance             6
  private MultiRequestSender ORBInstance.multiRequestSender_;                7
                                                                             8
  public MultiRequestSender ORBInstance.getMultiRequestSender()              9
  {                                                                          10
     return multiRequestSender_;                                             11
  }                                                                          12
                                                                             13
  after(ORB_impl orb,org.omg.CORBA.StringSeqHolder args,..):                 14
  execution(private void initialize(org.omg.CORBA.StringSeqHolder,...,       15
  String, int, java.util.Properties, int, int, int))                         16
  &&target(orb)&&args(//omitted){                                            17
    orb.orbInstance_.multiRequestSender_= new MultiRequestSender();          18
  }                                                                          19
                                                                             20
  public synchronized void                                                   21
  ORB_impl.send_multiple_requests_oneway(Request[] requests){                22
   if(destroy_) throw                                                        23
   new org.omg.CORBA.OBJECT_NOT_EXIST("ORB is destroyed");                   24
   com.ooc.OB.MultiRequestSender multi =                                     25
    this.orbInstance_.getMultiRequestSender();                               26
   multi.sendMultipleRequestsOneway(requests);                               27
  }                                                                          28
}                                                                            29
```

Fig. 3. DII: Multiple request sending

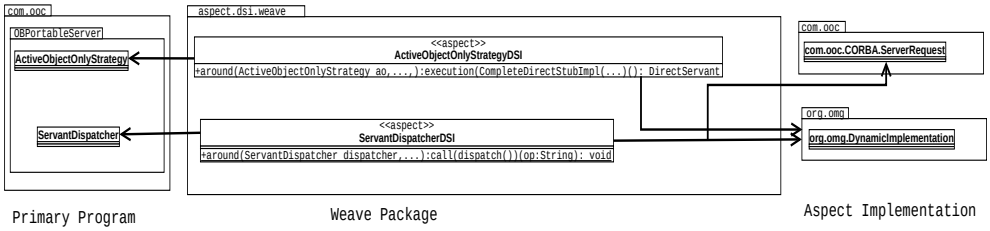

Fig. 4. UML Diagram of The DSI Aspect Implementation

```
package aspect.dsi.weave;
privileged aspect ActiveObjectOnlyStrategyDSI
{
  DirectServant around( ActiveObjectOnlyStrategy ao, . . .,)
  : execution(protected DirectServant ActiveObjectOnlyStrategy.
  completeDirectStubImpl( org.omg.PortableServer.POA, . . .))
  &&target(ao)&&args(. . .)
  {
   if(servant instanceof org.omg.PortableServer.DynamicImplementation){
     return null;
   }
   return proceed(ao,poa,rawoid,servant,policies);
  }
}
```

Fig. 5. Dynamic Skeleton Interface

3.2 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 ORBacus implementation of CORBA.

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.

In ORBacus, the collocation invocation is mainly implemented in the object initialization phase for both the client and the server. We present the AOP implementation of local server invocation in the UML diagram in Figure 6. The mechanism of collocation invocation is implemented by the `aspect.collo.weave` package which includes the following actions:

1. The *"around"* construct in `ClientManagerCo` weaves into the class `ClientManager` 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.

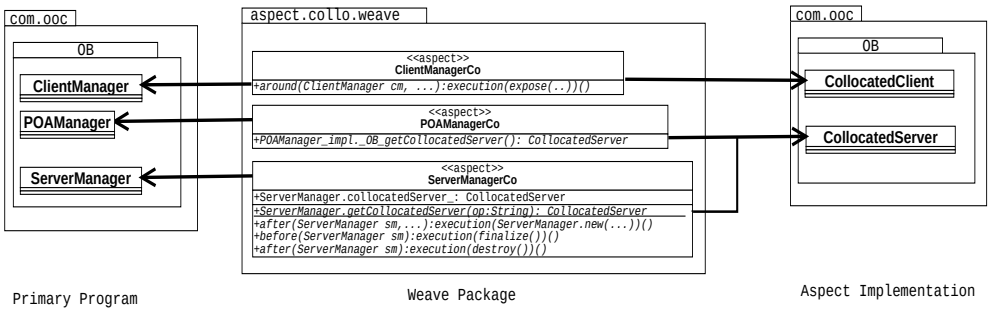

Fig. 6. UML Diagram of Aspect Implementation for Collocated Invocations

2. The `ServerManagerCo` aspect first adds a new attribute of type `CollocatedServer` to the class `ServerManager`. The "after" construct creates the runtime instance of the `CollocatedServer` after the constructor of `ServerManager` is executed. The second "after" advice disposes the `CollocatedServer`. The "before" construct verifies the validity of the `CollocatedServer` instance.
3. The `POAManagerCo` aspect first adds a method to allow the access to the `CollocatedServer`s.

Figure 7 presents the AspectJ code of `POAManagerCo`. Lines 5-9 declare one attribute and the accessor for that attribute in the class `ServerManager`. Lines 11-14 enforce some condition checking before the `finalize` method is invoked. Lines 16-20 create the runtime instance of the collocated server and add it to the list of servers. Lines 22-25 destroy the runtime instance for garbage collection as an additional step after the `destroy` method in the base implementation finishes execution.

3.3 Support for portable interceptors Portable Interceptors are hooks into the Orb through which CORBA services can intercept various stages during the request process. They are observer [6] style entities. Interceptors allow a third party to plug in additional Orb functionalities such as transaction support and security.

In ORBacus, the functionality of 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 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. Figure 8 presents the UML diagram for the portable interceptor implementation as aspect programs. The crosscutting points where the primary ORB model tangles

```
package aspect.collocation.weave;                                    1
import com.ooc.OB.*;                                                 2
privileged aspect ServerManagerCo                                    3
{                                                                    4
 private CollocatedServer ServerManager.collocatedServer_;           5
 public synchronized CollocatedServer                                6
 ServerManager.getCollocatedServer(){                                7
  return collocatedServer_;                                          8
 }                                                                   9
                                                                     10
 before(ServerManager sm):execution( * ServerManager.finalize()     11
   throws Throwable)&&target(sm){                                    12
  Assert._OB_assert(sm.collocatedServer_ == null);                   13
 }                                                                   14
                                                                     15
 after(ServerManager sm,. . .): execution(ServerManager.new(. . .)) 16
 &&target(sm)&&args(. . .){                                          17
   sm.collocatedServer_ = new CollocatedServer(oaInterface, concModel); 18
   sm.allServers_.addElement(sm.collocatedServer_);                  19
 }                                                                   20
                                                                     21
 after(ServerManager sm):execution(public synchronized void destroy()) 22
 &&target(sm){                                                       23
   sm.collocatedServer_ = null;                                      24
 }                                                                   25
}                                                                    26
```

Fig. 7. ServerManager Collocation Invocation

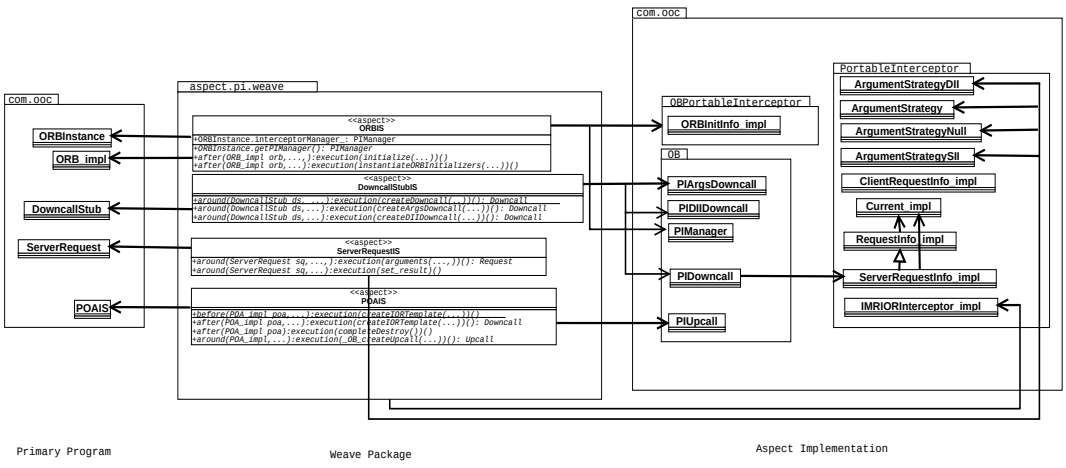

Fig. 8. UML Diagram of Portable Interceptor Support Aspect Implementation

with support for portable interceptors correspond to the standardized behaviour of portable interceptor. That is, an ORB implementation must allow interceptions made to the client request process, the server request process and the creation process of server objects. Since the "weaving" implementation of the portable interceptor suppport is rather complex, we present a summary of our AOP implementations as follows:

1. The portable interceptors can intercept the request sending process before it starts. Therefore, in ORBacus, the request sending process, i.e., the downcall creation process, needs to check if any client request interceptors are registered. If so, a downcall object is initialized with the portable interceptor information. Instead of letting ORBacus perform the checking regardless of whether portable interceptors are used or not, we moved the code segments into the aspect program in a "around" construct. As the result, the "aspectized" ORBacus only performs necessary checks if a portable interceptor is required for a particular application.
2. A similar situation occurs in the server-side request dispatching process, e.g., the upcall creation process. We moved the checking and upcall creation code into the aspect implementation. That makes the server request processing leaner and more precise. That is, it needs to reference and to handle portable interceptors only when it is necessary.
3. The portable object adaptor (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 the 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 such that, if the interceptor support is not needed, it is no longer necessary for the ORB to perform the extra initialization procedures.

We present two code snippets since the implementation of portable interceptor support is more complex than previous cases. Figure 9 is part of the POA related implementation of interceptor support. As defined in the OMG specification, compliant ORB implementations must notify interceptors of the object creation time. Lines 3-9 notify the portable interceptor manager "before" creating the internet object reference (IOR). The `after` advice notifies the manager when IOR is created (lines 11-16).

Figure 10 adds the support for portable interceptors to the class `ORB`. Firstly, the aspect code adds the new attribute `PIManager` and the corresponding accessor

```
package aspect.pi.weave;                                                    1
privileged aspect POAIS{                                                    2
  before(POA_impl poa,IORInfo_impl iorInfoImpl):                            3
  execution(private void POA_impl.                                          4
  createIORTemplate(com.ooc.PortableInterceptor.IORInfo_impl))              5
  &&target(poa)&&args(iorInfoImpl){                                         6
    com.ooc.OB.PIManager piManager = poa.orbInstance_.getPIManager();       7
    piManager.establishComponents(poa.iorInfo_);                            8
  }                                                                         9
                                                                            10
  after(POA_impl poa,. . .):execution(private void POA_impl.                11
  createIORTemplate(com.ooc.PortableInterceptor.IORInfo_impl))              12
  &&target(poa)&&args(iorInfoImpl){                                         13
   com.ooc.OB.PIManager piManager = poa.orbInstance_.getPIManager();        14
   piManager.componentsEstablished(poa.iorInfo_);                           15
  }                                                                         16
}                                                                           17
```

Fig. 9. POA Portable Interceptor Support

to the class `ORB`(Lines 5 - 8). `PIManager` is responsible for managing the interceptors registered in the ORB. Lines 13-15 create the runtime instance of `PIManager` as the first task after the ORB finishes initialization. Lines 18-24 instantiate a codeset interceptor and register it with the manager. Lines 26-38 invoke customized `ORBInitializer`s after the normal initialization of ORB finishes. The last line (line 39) notifies all the interceptors registered with `PIManager` of the event that the ORB has been initialized.

3.4 Aspect-Aware IDL Compiler

Our re-factoring work not only resolve the crosscutting of the internal architecture, but also aims at simplifying the user's view of the Orb by developing the aspect-aware IDL compiler. This is because certain CORBA features, such as the dynamic programming interface and the collocated server invocation, require special treatment and support in the CORBA API, i.e., the standardized IDL definitions and the associated language mappings. The complete re-factorization of these features must include the associated API code. This is because even if these features are not required by particular applications, the associated API code still contributes to the complexity of the overall API set and consumes computing resources.

In this case study, we explore the functionality of the aspect-aware IDL compiler by implementing two additional tasks as compared to the ordinary stub compilers during the language translation. These tasks are *API splitting* and

```
package aspect.pi.weave;
privileged aspect ORBIS{
 //introduce a new field PIManager in ORBInstance. This field is initialized by
 //ORB_Impl, which is executed before ORBInstance
 private PIManager ORBInstance.interceptorManager_=null;
 public PIManager ORBInstance.getPIManager(){
   return interceptorManager_;
 }

 after(ORB_impl orb, . . .,):execution(private void
 initialize(StringSeqHolder, String, . . .))&&target(orb)&&args(. . .)
 {
    PIManager piManager = new PIManager(orb);
    piManager.setORBInstance(orb.orbInstance_);
    orb.orbInstance_.interceptorManager_=piManager;
    // Initialize Portable Interceptors - this must be done after installing the OCI
   //plug-ins to allow an ORBInitializer or interceptor to make a remote invocation
    try{
      piManager.addIORInterceptor(new com.ooc.OB.
          CodeSetIORInterceptor_impl(nativeCs, nativeWcs),false);
    }
    catch(DuplicateName ex){
       com.ooc.OB.Assert._OB_assert(false);
    }
   //set up ORBInitInfo
   if(!orb.orbInitializers_.isEmpty()){
   com.ooc.OBPortableInterceptor.ORBInitInfo_impl info =
   new com.ooc.OBPortableInterceptor.ORBInitInfo_impl(. . .);
   java.util.Enumeration e = orb.orbInitializers_.elements();
   while(e.hasMoreElements()){
    ((ORBInitializer)e.nextElement()).pre_init(info);
   }
   e = orb.orbInitializers_.elements();
   while(e.hasMoreElements()){
     ((ORBInitializer)e.nextElement()).post_init(info);
   }
   info._OB_destroy();
  }
  piManager.setupComplete();
}
```

Fig. 10. ORB Portable Interceptor Support

Local Invocation Optimization. Both features require modifications to the IDL compiler code in a crosscutting manner and can therefore be implemented using AspectJ. Our implementation is experimental and based on the JacORB IDL compiler since the source of ORBacus IDL compiler is not part of the open source distribution. The implementation consists of the following modifications added to JacORB compiler using AspectJ:

1. Two additional compiler options are added. The "-split" argument is followed by a subset of original IDL definitions, which corresponds to the features that are already factored out. This tells the compiler to generate these IDL definitions as AspectJ modules consisting of "inter-type declarations"s. At the same time, it skips the same API defined in the original IDL definition. In this way, we do not change the original IDL definition. The "-local" argument is designed to eliminate parts of the generated stub code deciding if the optimization of collocated invocations are needed. These additional compiler options are added in an "after" advice, which parses the command line arguments and sets the corresponding flags in the parser.
2. The stub code for the original IDL definition is split into two sets of modules, the standard language mapping and AspectJ modules. This is done through three steps: 1. The parser first reads the IDL definitions following the "-split" switch and stores the declarations of the interfaces and methods; 2. An "around" advice is defined to replace the code generation method in the IDL compiler. It checks if this particular interface needs to be split by doing a lookup from the storage. If the interface contains operations supporting the "aspectized" feature, a separate print stream is set up for generating the AspectJ code. Upon the completion of code generation for the interface, the print stream generates the enclosing AspectJ symbols if necessary; 3. Before generating the stub code for methods, a "before" advice uses the alternative output stream set up earlier if the method is to be translated into AspectJ "inter-type declaration"s. Figure 11 illustrates a simple usage of this feature. The original IDL definitions contain two methods, where the underlined method supports a re-factored feature. This method is re-defined using IDL syntax in a separate file and read by the compiler following the "-split" option. The generated Java language mapping as well as the AspectJ code is shown at the bottom of the figure.
3. The "-local" switch is to tell our compiler to simplify the control logic of the stub code if the feature of optimizing for invoking collocated servers is not needed. In our current implementation, using the switch will eliminate the conditional statements in the stub code, which checks if the remote server is actually located in-process. This is done by replacing the original translation code with our own methods defined in a "around" advice.

Our IDL compiler implementation has the following properties: 1. We achieve the maximum reuse of existing code as no modifications is made to the original IDL compiler code. 2. New compiler features can be added and removed very easily as they are all implemented in AspectJ. In addition, since these features are implemented in separate aspect modules, they can be independently added or

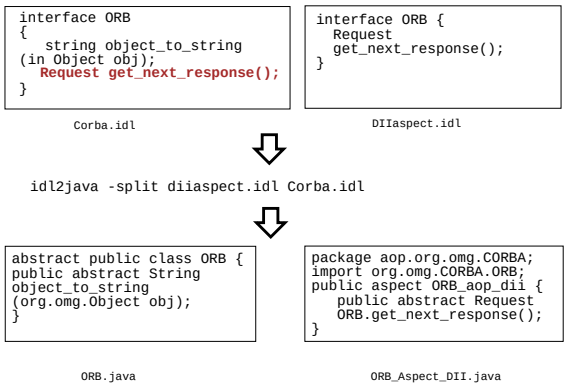

Fig. 11. Aspect-aware IDL Compiler

removed without affecting others. 3. Existing IDL language features are sufficient for our compiler to generate "aspectized" code. No aspect oriented extension are needed to support the "aspectization" at the IDL definition level.

Our implementation introduces some overhead to the compilation process because we need to check, for every method, whether the AspectJ code needs to be generated. Testing shows a lookup from storage of interfaces which are to be generated in the aspect code takes around 20 microseconds. The overhead is therefore not noticeable even compiling a very large set of IDL files.

3.5 Re-factorization Results

Table 1 presents the measurements of both the structural metrics and the runtime metrics for the AOP re-factoring of dynamic programming interface, portable interceptor support, collocation invocation. The structural metrics are collected on the ORBacus implementation prior to and after the AOP re-factoring. Table 1 reports the accumulated reductions for every re-factored features. The data indicates that, through the aspect oriented re-factorization, we reduce the size of ORBacus by more than two thousand lines and around 70 references to other class types. The structure of ORBacus becomes less complex with features taken out and still capable of supporting transparent remote invocations with improved response time to the original implementation. Table 1 also reports the runtime interval measurements for the four segments of the ORBacus stack traversal. The response time is measured using the original implementation, the "woven" implementation(re-combined), and the implementation with features factored out(re-factored). The runtime performance of the three Orbs is largely equivalent. The sizes are in lines of executing code; weight is in number of methods; interval is in micro-seconds. Other metrics are indexes, see earlier section for an interpretation.

	Structural Metrics				Interval			
	CCN	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Dynamic Invocation Interface								
Original	25.4	2000	200	203.35	105	125	37	59
Re-factored	23.8	1490	188.35	196.55	108	124	35	59
Dynamic Skeleton Interface								
Original	7.28	369	18.66	28	79	126	43	8
Re-factored	6.92	262	18.66	27	76	119	41	9
Portable Interceptor Support								
Original	24.66	4218	160.8	208.5	78	118	42	8
Re-factored	24.0	2909	160.2	194.28	79	122	42	9
Collocation Invocation								
Original	15.66	638	33.99	63	79	126	43	8
Re-factored	15.00	435	32.01	57.99	76	126	41	7
Overall								
Original	105	7320	412.5	528.25	76	126	37	9
Re-factored	101	4899	400	458.25	76	123	37	9
Re-combined	n/a	n/a	n/a	n/a	74	123	37	8

Table 1. Metric Matrix for the re-factorization of DII, DSI, Portable Interceptor, Collocation Invocation, and overall assessment (CCN - Cyclomatic Complexity Number.)

3.6 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 the tool support.

1. We did not completely factor out class types such as Any and NVList, which are used widely for other purpose 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 work until future research when it is necessary to exactly quantify the aspect of dynamic programming interface.
2. Our re-factorization of the CORBA features in the generated stub code and API code is not complete. Our aspect-aware IDL compiler is of a prototype nature and needs to be extended. This is due to the fact that new aspects of CORBA are still being discovered. The role and the features of the aspect-aware IDL compiler ought to be thoroughly analyzed. We defer this discussion to future work. As the 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 that 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 re-factoring almost trivial.

4 Conclusion

We believe that adaptability and configurability are essential characteristics of middleware substrates. Those two qualities require a very high level of modularity in the middleware architecture. Traditional software architectural approaches exhibit serious limitations in preserving the modularity in the process of establishing decomposition models for crosscutting design concerns. 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 aspect allows us to compose, with respect to the primary decomposition model, the most appropriate solution for each design requirement. By weaving the aspects together, we are able to improve the modularity of final systems in the dimension of aspects.

To better approximate the benefit of designing middleware with AOP, we use AspectJ to re-factor a number of aspects. 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 with certain pervasive features 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 promises in conquering the complexity of middleware architecture. In our future work, we will try to gain more experience in terms of applying aspect oriented development methodologies. We are exploring various techniques to help us define horizontal decomposition procedures more concretely. We will eventually use all these experience to design a fully aspect oriented middleware platform.

References

1. AspectJ. URL: `http://www.eclipse.org/aspectj`.
2. Hyper/J. URL: `http://www.alphaworks.ibm.com/tech/hyperj`.
3. M. Astley, D.C. Sturman, and G. A. Agha. Customizable Middleware for Modular Software. *ACM Communications*, May 2001.
4. Louis DiPalma and Robert Kelly. Applying CORBA in a contemporary embedded military combat system. OMG's Second Workshop on Real-time And Embedded Distributed Object Computing, June 2001.

5. Robert Filman. Achieving ilities. URL: http://ic.arc.nasa.gov/~filman/text/oif/wcsa-achieving-ilities.pdf.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
7. Gnome. ORBit. URL: http://www.gnome.org/projects/ORBit2/.
8. Object Management Group. The common object request broker: Architecture and specification. December 2001.
9. G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4), 1996.
10. Fabio Kon, Manual Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhaes, and 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.
11. Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. QoS aspect languages and their runtime integration. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers. Lecture Notes in Computer Science, Vol. 1511, Springer-Verlag*, Pittsburgh, Pennsylvania, USA, May28-30, 1998.
12. Clarke M., Blair G., Coulson G., and Parlavantzas N. An efficient component model for the construction of adaptive middleware. *IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, November 2001.
13. M. Rom, D. Mickunas, F. Kon, and R. H. Campbell. LegORB and Ubiquitous Corba. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, pages 1–2, Palisades, NY, USA, 2000.
14. Cisco Systems. Cisco ons 15327 - sonet multiservice platform. URL: http://www.cisco.com/univercd/cc/td/doc/pcat/15327.htm.
15. Iona Technologies. ORBacus. URL: http://www.iona.com/products/orbacus_home.htm.
16. Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd International Conference on Aspect Oriented Systems and Design*, pages 130–139, Boston, MA, March 2003.
17. Charles Zhang and Hans-Arno Jacobsen. Re-factoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 2003. (accepted for publication).
18. John Zinky, Joe Loyall, Partha Pal, Richard Shapiro, Richard Schantz, James Megquier, Michael Atighetchi, Craig Rodrigues, and David Karr. An AOP challenge problem: Managing QoS on iteractions between distributed objects. In *White Paper for ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, April 2000.