Improving Modularity of Reflective Middleware with Aspect-Oriented Programming

Nelio Cacho[1] Thais Batista[3] Alessandro Garcia[1]
Claudio Sant'Anna[2] Gordon Blair[1]

[1]Computing Department, Lancaster University, United Kingdom
[2]Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Brazil
[3]Computer Science Department, Federal University of Rio Grande do Norte – UFRN, Brazil
{n.cacho, garciaa, gordon}@comp.lancs.ac.uk,
claudios@inf.puc-rio.br, thais@ufrnet.br

ABSTRACT
Reflective middleware has been proposed as an effective way to enhance adaptability of component-oriented middleware architectures. To be effectively adaptable, the implementation of reflective middleware needs to be modular. However, some recently emerged applications such as mobile, pervasive, and embedded applications have imposed more stringent modularity requirements to the middleware design. They require support for the conception of a minimal middleware while promoting fine-grained modularity of reflective middleware features. The key problem is that fundamental mechanisms for decomposing reflective middleware implementations, such as object-oriented ones, suffer from not providing the proper means to achieve the required level of localizing reflection-specific concerns. This paper presents a systematic investigation on how aspect-oriented programming scales up to improve modularity of typical reflection-specific crosscutting concerns. We have quantitatively compared Java and AspectJ implementations of an OpenORB-compliant reflective middleware using separation of concerns metrics.

1. INTRODUCTION
Middleware platforms [12] provide high-level abstraction to make it easier the development of distributed component-based applications. Traditional middleware architecture includes a variety of features to satisfy distinct application domains. This broad range of features has increased the popularity of such platforms but, on the other hand, it has increased the size and complexity of middleware systems. In addition, the black-box nature of conventional middleware makes it difficult to adapt it for a specific purpose. The idea of reflective middleware [15,17] has been recognized as a promising way to overcome these problems. It relies on computational reflection [23] to support configurability and adaptability of the platform. However, reflective architectures are not enough to broadly satisfy requirements imposed by some recently emerged applications such as mobile, pervasive, and embedded applications. Embedded applications, for instance, are resource constrained and demand a minimal middleware [9, 27].

In fact, several researchers [15-18,34] have tried hard to push reflective middleware systems a step further by allowing the definition of customized instances of the platform with small resource footprints. It has been observed that the practical effectiveness of such middleware adaptability is directly dependent on the implementation model because it specifies the modular structure of the middleware internal elements [9, 27]. A proper modular implementation is also essential to the definition of a minimal middleware core that satisfies the memory and resource requirements of nowadays applications. However, highly-adaptable implementations of such systems have been often shown as a challenge because many of the core reflective features typically crosscut the modular object-oriented decomposition of the middleware architecture. Some examples are the mechanisms dedicated to support causal connection, state recovery of objects and meta-objects, and introspection about the binding process. These features have a broadly-scoped effect over the target component model, thereby both liming the conception of a minimal middleware and reducing its modularity and adaptability.

In this context, it is important to systematically verify the suitability of Aspect-Oriented Programming (AOP) [4] to improve the modularity of reflective middleware systems. In this direction, there are some efforts [5,19-21,25] that use AOP to improve general middleware modularity. However, the authors commonly "aspectize" a certain category of crosscutting concerns, such as logging, monitoring, and statistics, which are not part of the core reflection model. Even works that explicitly deal with middleware architecture [20, 21] they do not address reflective middleware. Hence, there is no systematic study that investigates the impact of AOP on the modularity and configurability of elementary crosscutting concerns in reflective middleware design. In fact,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEM'06, November 10, 2006, Portland, Oregon, USA
Copyright 2006 ACM 1-59593-585-1/06/11 ...$5.00.

little attention has been given to analyze to what extent the aspectization of typical implementation models of reflective middleware architectures can address limitations of other decomposition paradigms such as object-orientation. Also there is no investigation that quantifies the benefits and drawbacks involving the interplay of reflective middleware and AOP.

This paper presents an in-depth case study in which we have compared the modularity of object-oriented (OO) and aspect-oriented (AO) implementations of a typical reflective middleware system. The case study is structured according to the OpenORB architecture [16], which is a precursor model of the reflective middleware idea. We have implemented both Java and AspectJ [29] versions of OpenORB. In order to produce a well modularized system, in both implementations we have used design patterns [1]. Our investigation complements the existing empirical body of knowledge in the component-oriented middleware arena, since our evaluation has quantified through a metric suite the effects of aspectizing a reflective middleware on primary modularity attributes, such as separation of concerns. Our analysis encompassed a plethora of elementary middleware features related to the underlying component model, the communication infrastructure, the acceptance and connection infrastructure, and reflection-specific concerns, such as causal connection and introspection of the binding process.

This paper is organized as follows. Section 2 presents the relevant information related to the setting of our experimental study. Section 3 discusses the OO design of the OpenORB middleware and its modularity problems; it also discusses the aspectization strategy for each of those problems. Section 4 presents the quantitative modularity evaluation of the Java and AspectJ implementations. Section 5 discusses related work, while Section 6 presents the concluding remarks.

2. STUDY SETTING

This section describes the configuration of our empirical study. Our investigation is focused on assessing the positive and negative influences of AOP on the modularity of an OpenORB-compliant reflective middleware [15]. Section 2.1 introduces the main concepts of reflective middleware, while Section 2.2 describes the OpenORB model. Section 2.3 introduces the main comparison procedures and Section 2.4 describes our selected metrics suite for evaluating different modularity facets. Finally, Section 2.5 discusses our measurement procedures.

2.1 Reflective Middleware

Reflective middleware relies on computational reflection [23] to define a causally connected self representation that supports inspection and adaptation of its behavior. Computational reflection is the ability of a system inspecting and manipulating its internal implementation. This is achieved by a two-level representation of the system: the *base-level* and the *meta-level*. The base-level is composed of *base-objects*. The meta-level is represented by *meta-objects* that monitor and influence the base-level. The meta-level performs computation about the system itself. In reflective middleware platforms, the middleware core is represented by base-objects. Meta-objects are associated with base-objects and a *causal connection* allows that changes in the meta-level are reflected in the base-level and vice-versa.

2.2 OpenORB Architecture

OpenORB is a pioneer reflective middleware. Unlike the black-box nature of traditional middleware, in the OpenORB component model the middleware is organized as a set of components and dynamic adaptation is supported by means of computational reflection.

The main elements of OpenORB base-level are interfaces, local bindings, components and capsule. A *component* is a unit of independent deployment that provides and requires services via interfaces. Interactions between components are specified via interfaces. *Interfaces* represent the access point of a component, the so-called *ports*. Each interface can export and import methods. *Exported methods* are those provided by a component while *imported methods* are those required by a component. *Bindings* between interfaces are themselves components. There are two types of bindings: *local bindings* and *distributed bindings*. A *local binding* associates exported interfaces with imported interfaces. They are used to bind interfaces that are in the same address space (capsule). *Distributed bindings* are distributed components which may span capsule and machine boundaries [18]. They are composed of components bound by local bindings. A *Capsule* is a logical container of components that provides an API for loading and binding components. Connection between components of different capsules is through *implicit and explicit bindings* [18]. An *implicit binding* is created between two remote interfaces with no interference of a programmer. This binding is used when an interface is imported. An *explicit binding* is created by a programmer and can be of the following types: Operational, Signal, Stream. An *Operational binding* is used to send method invocations and, optionally, to receive the result. A Signal binding supports a set of one-way signals. The *Stream binding* is similar to the signal binding except that it supports the transmission of continuous data such as audio and video.

The OpenORB meta-level defines the reflective mechanisms that implements a causal connection with the base level. In order to organize the meta-level, OpenORB proposes four meta-models [15]: (1) *Encapsulation*: responsible for exposing the encapsulation provided by objects. It allows the inspection, modification and extension of the implementation of an object. This meta-model can be used to monitor and to control the access to an object, its attributes and methods; (2) *Composition*: responsible for providing the bindings graph of a component. Using this meta-model it is possible to insert, to remove and to replace components; *(3) Environment*: responsible for exposing the execution environment of each interface including method invocations for servers and clients.; (4) *Resource:* responsible for reifying and managing the resources used by each object.

2.3 Comparison Procedures

Our study has focused on the assessment of two versions of an OpenORB-compliant middleware implementation: an object-oriented (OO) and an aspect-oriented (AO) version. First, we have used the Java programming language to develop the OO version. We have used a number of design patterns [1] in order to produce a well modularized design. The choice of the applied patterns was driven by maximizing the separation of reflection-specific concerns and other relevant middleware concerns in order to make them adaptable spots in the middleware implementation.

Afterwards, we have reengineered the existing Java implementation with AspectJ in order to produce the AO version of the middleware. We have tried to modularize the same concerns as in the Java implementation. However, the AspectJ version also relies on the use of aspects to modularize reflective middleware concerns which exhibited crosscutting behavior, and should not be modularly captured with the OO implementation. Where it was possible, we used the AspectJ solutions proposed by Hannemann and Kickzales [28] for implementing design patterns. We have compared the modularity of the Java and AspectJ implementations using the metrics suite described in the next section. Section 4 presents the evaluation results.

2.4 The Modularity Metrics Suite

The quantitative assessment was based on the application of a metrics suite [6] to the two versions of the middleware implementation. These metrics are useful to capture important modularity dimensions in the system design, namely separation of concerns (SoC), coupling, cohesion, and size. Due to space limitation, we are going to focus on discussing the SoC measures because they have provided the most interesting results. All the results for the coupling, cohesion and size measures can be found at [35].

The SoC [6] metrics capture the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. The used SoC metrics are briefly described in Table 1; an extensive explanation and justification about their value to assess modularity are out of the scope of this work and can be found at [6]. Table 1 presents a brief definition of each metric and associates them with the attributes measured by each one. These metrics have already been extensively used in several studies [3, 5-7, 10], where they have been proved to be useful quality indicators. We have applied the chosen metrics to both Java and AspectJ versions.

2.5 The Measurement Process.

In the measurement process, the data collection of the SoC metrics was preceded by the shadowing of every class, interface and aspect in both implementations of the middleware. The shadowing consists of annotating elements in the code (e.g. attributes, operations, statements, and so on) that realize the implementation of relevant concerns [6]. In this case study, both AspectJ and Java implementations were shadowed according to the reflective middleware features that should be modularized in both solutions, and were the main subject of assessment in this work.

In fact, we have treated both reflection-specific and other fundamental features of the middleware design as concerns of interest in order to investigate its crosscutting structure in both Java and AspectJ implementations. After the shadowing, the data of the SoC metrics (CDC, CDO, and CDLOC) was manually collected at that time (even though we have nowadays a measurement tool available for this purpose [30]). The other measurements for size, coupling, and cohesion were based on the use of our AJATO tool [30]. Since our selected metrics are oriented to fine-grained units such as CDLOC, we had an additional standardization task before collecting the data. This task aimed at ensuring that the two developed middleware versions implement the same functionalities. This activity also removed problems related to different coding styles. Section 3 will describe some of the investigated solutions in both OO and AO implementations, while Section 4 presents the description of the measurement results and their analysis.

3. ASPECTIZING REFLECTIVE MIDDLEWARE

This section describes different situations of crosscutting that prevents proper modularity of the target reflective middleware system and also a subset of aspect-oriented solutions we have used to improve the modularization of such crosscutting middleware features.

In this way, Figure 1 illustrates a slice of the OpenORB design, including some basic classes and behaviors defined by the reflective infrastructure to support adaptation, such as: *Component, Port* and *ConcreteBind*. Each component has one or more *Ports* which are implemented by *Receptacles* and *Interfaces*. To deal with the interaction between component ports, *ConcreteBind* is responsible for providing a mechanism to route the invocations to the proper target object. It also defines a binding state machine that allows redefining the binding flow during its execution. The central part of the open binding design is based on the *Mediator* pattern in order to route the invocations. The *ConcreteBind* class is in charge of coordinating the interaction among *Ports*. It also promotes loose coupling by avoiding that ports refer each other explicitly. This facility allows the designer to easily change the binding.

To support the client invocation, Receptacle has a reference to the *BindMediator*, once this class operates on behalf of the client rule to invoke the functionalities defined on Interfaces. In order to provide the component functionalities, the Interface class has a reference to the object that implements the required functionality. Hence, *Receptacle* requires services and *Interface* provides services.

In addition to the above description, it is possible to see that some concerns, such as the binding state machine definition and detection mechanism crosscut the design illustrated by Figure 1. Due to the limited space, many others crosscutting concerns listed in table 2 are not going to be described.

Table 1. Metrics for Separation of Concerns

Metrics	Definitions
Concern Diffusion over Components (CDC)	Counts the number of classes and aspects whose main purpose is the implementation of a concern and the number of other classes and aspects that access them.
Concern Diffusion over Operations (CDO)	Counts the number of methods and advices whose main purpose is the implementation of a concern and the number of other methods and advices that access them.
Concern Diffusions over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a “concern switch”.

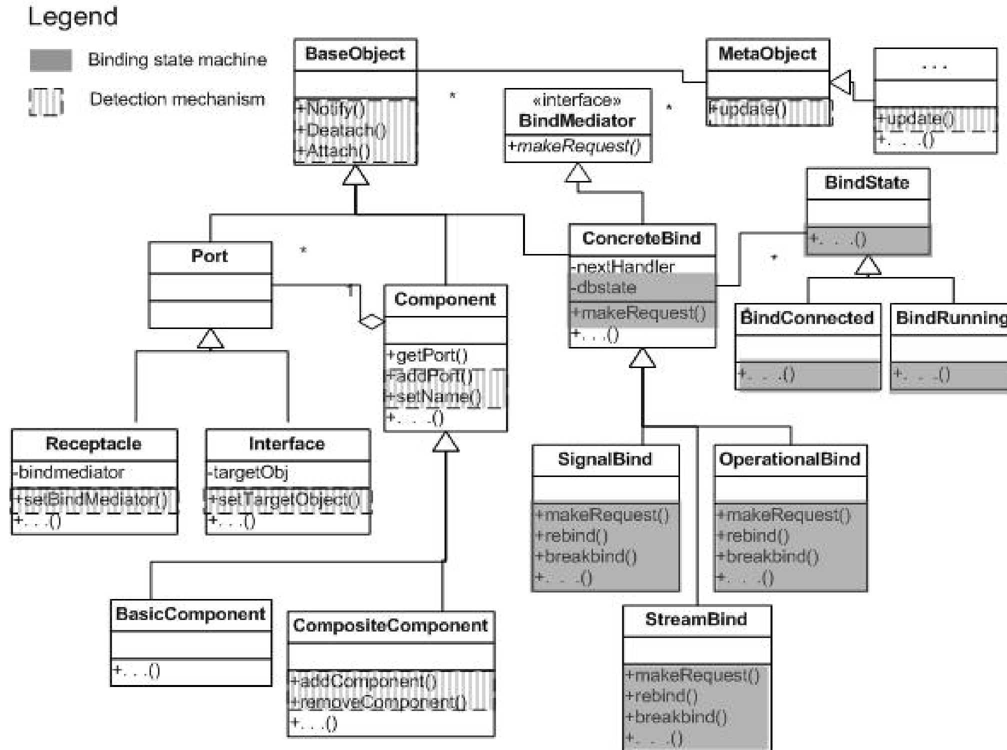

Figure 1: Crosscutting Concerns in the OpenORB component infrastructure

The *binding state* machine contains a number of state transitions that are related to the status of the resources, closing binding and aborting communication. Figure 1 depicts the design of the *binding state* machine as a gray box which defines the states of binding objects. The *BindState* interface provides a set of states such as *connected* and *running* which change according to the binding operation.

```
public Object makeRequest(String met,
Object[] args){
               . . .
  Class classes[] = null;
  dbstate = new BindRunning();
  Class oClass = getTargetReceptacle();
  Method metExe = getMethod(met, args,
                            oClass);
  Object[] realargs = getArgs(met,args);
  //invoke target method
               . . .
  dbstate = new BindConnected();
}
```

Figure 2: State transitions involving open binding

As illustrated in Figure 2, these state modifications crosscut all minimal core to ensure the state machine integrity. In order to define the current state, the attribute *dbstate* receives a predefined state within the some methods, such as *makeRequest*. Based on the defined state, a different set of operations are available or not to be executed. For example, if the *dbstate* is defined as *connected*, the application can perform more operations compared to the situation where the state is *running*. In addition to the different *State* behavior, each kind of binding has a different set of states and transitions points. Thus, different transitions points are scattered over the policies of signal, stream and operational bindings.

In order to remove the scattered code produced by the state definition, we use AspectJ to promote a different strategy to manage the state transitions. Aspects are used to remove the state transitions from implementation of the binding policies; pointcuts and advice are respectively used to detect the transition points (join points) and to define the transition itself. This approach allows decoupling the states from each other and also facilitates the maintenance of the state transition control once the state transition code is located in an aspectual module. Thus, Figure 3 describes how to separate state transactions from the core implementation. *ChangeState* defines pointcuts and advices to intercept all state-changing invocations. For instance, after the *makeRequest* execution, the result is used to define the new state of the binding. As a consequence of this modularized approach, it is straightforward to insert and change the state machine in order to adapt to new operational situations. It also promotes safer adaptation of the state machine, as the modular visibility of the transitions points helps the prediction of harmful change effects and the reliable insertion of new states.

In addition to the *binding state machine*, reflective middleware allows to insert new functionalities in its internal implementation. This is achieved by a two-level representation of the system: the *base-level* and the *meta-level*. The base-level is composed of *base-objects*. The meta-level is represented by *meta-objects* that

monitor the base-level. The meta-level performs computation about the system itself. In reflective middleware platforms, the middleware core is represented by base-objects. Meta-objects are associated with base-objects and a *causal connection* allows that changes in the meta-level are reflected in the base-level and vice-versa.

However, the OO implementation of the reflective model imposes some restrictions to the ORB modularity, once the maintenance of the causal connection requires the intrusive introduction of reflection code across the component infrastructure. Reflection-specific concerns are then superimposed to concerns specific to the component model. It occurs mainly because the causal connection needs to associate the base-objects with the meta-objects and to guarantee that changes to the base-level are reflected into the meta-level and vice-versa. This inter-level state dependency between objects is critical for several reasons, such as: (i) it defines a tightly connected object model, and (ii) as the middleware system evolves, it tends to include more and more complex relationship between meta and base objects.

```
pointcut changeState(
 SignalBind bind) : execution(
 public Object SignalBind.makeRequest(..))
                         && target(bind);
   before(SignalBind bind)  :
      changeState(bind)
        {  bind.bdstate =
                  new BindRunning();
        }
   after(SignalBind bind)  :
        changeState(bind)
        {
            bind.bdstate =
                 new BindConnected();
        }
   . . .
}
```

Figure 3: Open binding state changing

One of the main parts of the causal connection implementation is represented by the *detection mechanism*. This mechanism detects changes in the base-object and informs these occurrences to the appropriate meta-objects. Figure 1 illustrates the design of the detection mechanism by dotted boxes which is centered on an instance of the *Observer* pattern [1]. It defines an one-to-many dependency involving a base-object and any number of meta-objects so that, when the state of the base-object changes, all its meta-objects are automatically notified and updated. The goal of this OO decomposition is to support the reuse of base-objects without reusing their meta-objects, and vice versa. In addition, we can add a meta-object without modifying the base-object or other meta-objects. These features are compatible with the intention of causal connection. To maintain meta-objects in a consistent state, base-objects (Port, Component and Binding strategies) update the methods of its meta-objects whenever a change occurs that could make its meta-object state inconsistent with its own. After being informed of a change in the concrete base-object, a meta-object may query the base-object for information.

Despite the methods that implement the causal connection are defined in the *BaseObject* class, the invocation of the *update* method is scattered within the code of many methods in the base-objects hierarchy. The AO version of the OpenORB implementation defines an aspect to modularize the scattered code relative to the change detection mechanism. As a result, base-level module implementations do not have to be aware of the meta-level elements. Figure 4 illustrates a slice of the *DetectMechanism* aspect which defines some pointcuts, one for each method where the causal connection should be monitored. For instance, *subjectChange* pointcut in Figure 4 monitors the execution of the *Component.setName* through the advice definition to update the meta-objects associated with it. Hence the implementation of the AO approach was able to remove all the dotted boxes that represent the change-detection crosscutting in Figure 1.

```
public aspect DetectionMechanism {


protected pointcut subjectChange(Object obj):
  call(public void Component.setName(String))
   && target(obj);
                      . . .
after(Object obj): subjectChange(obj) {
     Object[] argss = thisJoinPoint.getArgs();
     Signature sig=(Signature)thisJoinPoint.
                              getSignature();
     String operation = sig.getName();
     updateMetaObjects(obj, operation, argss);
    }
}
```

Figure 4: Aspect to modularize change detections

4. QUANTITATIVE EVALUATION

This section describes the quantitative comparison of the Java and AspectJ implementations for the OpenORB-compliant reflective middleware, based on a suite of concern-oriented modularity metrics. The idea is to assess to what extent the AO and OO design elements (Section 3) and other ones defined in both OpenORB implementations support the middleware modularity. We present the results by means of tables that put side-by side the values of the metrics for the OO and AO versions of each target middleware system. We present the results of our analysis in terms of modularity measures with respect to separation of the reflective middleware concerns. Table 2 shows the obtained results for the three separations of concerns (SoC) metrics. The measures are presented according to elementary features of the reflective middleware, namely the underlying component model, the communication infrastructure, and reflection-specific concerns, such as causal connection and the meta-model. Table 2 also indicates the main OO design pattern used to implement each of the middleware elements. Due to space constraints, we have concentrated on the most important features and subfeatures of the reflective middleware. We have selected those features in which their modularity, according to our observation, has been suffered no impact (Section 4.1) and positive influences (Section 4.2) on the application of AOP.

Table 2. Quantifying Separation of Concerns

	CDC		CDO		CDLOC		Superior
CONCERNS (Pattern)	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**Solution**
Component Model	10	4	18	17	30	2	AO+
. composite structure of components (Composite)	3	3	11	11	10	2	AO
. binding	42	30	109	93	22	4	AO+
.. binding state (State)	4	4	12	12	14	2	AO
.. managing required/provided interfaces (Mediator)	5	5	18	18	22	2	AO
.. remote invocation mechanism (Proxy)	24	19	69	59	2	2	AO
.. family of binding strategies (Strategy)	4	3	9	5	8	8	AO
Communication Infrastructure	21	20	77	75	20	18	AO
. message assembling (Builder)	8	8	28	28	10	10	=
. isolation of API details (Adapter)	5	5	25	24	2	2	AO
. efficient storage of connections (Flyweight)	11	10	29	27	20	18	AO
Reflection Infrastructure							
. causal connection	9	11	21	19	38	18	AO
.. detection mechanism (Observer)	8	8	14	7	28	12	AO
.. capture mechanism (Memento)	4	2	8	12	12	8	AO
. metamodel	8	6	22	18	16	8	AO+
.. metamodel composite (Composite/Visitor)	4	3	7	7	6	2	AO
.. encapsulation (Decorator)	4	3	15	11	10	8	AO
Success Total	**1 vs. 10**		**1 vs. 11**		**0 vs. 13**		**0 vs. 16**

4.1 No Effect of AOP

Table 2 omits the data relative to other elements in the component model, such as (i) the structure responsible for storing the list of components, interfaces and methods (implemented as Iterator pattern [1]), (ii) the alternative behaviors for binding (Template Method pattern), (iii) the unit representing the capsule (Singleton pattern) that controls the components made available by the OpenORB, and (iv) the main interface of OpenORB (Façade pattern). These elements are example parts of the core behavior of the OpenORB system that have been similarly implemented in Java and AspectJ. Similarly, we have not included other elements, such as the subfeatures of the acceptance/connection infrastructure and complementary mechanisms in the communication infrastructure.

4.2 Increased Separation with AOP

An analysis of Table 2 shows that the AO version of the middleware system performed better than the OO version for most the measures. In particular, the AspectJ version is superior for all the crosscutting concerns discussed in Section 3, such as the binding state, and the causal connection mechanisms. There was only one exception relative to the implementation of the message assembling mechanism, where it was clear that such a middleware feature has already been nicely realized through the OO implementation of the Builder pattern. There were some isolate points were the AO solution achieved a slightly worse result, namely: (i) the number of operations (CDO metric) to implement the functionality responsible for recovery states of objects and meta-objects, and (ii) the tally of modular units (CDC metric) to implement the causal connection feature. However, the other SoC metrics for the same features have compensated these punctual breakdowns. For example, although there are more operations dedicated to implement state recovery, they are localized in fewer components and with reduced amount of tangling or transition points (CDLOC metric).

The SoC metrics indicate significant modularity improvements in a number of features and subfeatures of the aspectized reflective middleware, including binding, binding state, management of involved interfaces, remote invocation mechanism, and so forth. More importantly, the separation of all the reflection subfeatures relative to causal connection and metamodel has been clearly enhanced by the aspect-oriented mechanisms. The AspectJ superiority exceeds 50% in some cases, such as the level of tangling (CDLOC) in the implementation of the composite structure of components, causal connection, change notification of meta-objects, and metamodel composite.

4.3 Study Constraints

Our study focuses on the comparison of a single AO language - namely AspectJ – and an OO language. Although many ideas presented here also apply to other AO languages, some surely do not. Arguably, the employed metrics suite does not cover all the possible modularity dimensions. There are a number of other existing metrics and other modularity dimensions that we could be exploited in our study. We have decided to focus on the metrics described in Section 2.4 because they have already been proved to be effective quality indicators in several case studies (e.g. [3, 5-7, 10]). In fact, despite the well-known limitations of these metrics, they complement each other and are very useful when analyzed together. In addition, there is no way in a single study to explore all the possible measures. For every possible metrics suite there will be some dimensions that will remain uncovered. In addition, future case studies can use additional metrics and assess the aspectization of reflection-specific features using different modularity dimensions.

5. RELATED WORK

There are a number of works that address adaptability of middleware platforms. The most representative work in this context are reflection-based middleware platforms

[13,15,16,17,24] that exploit the power of adaptability provided by the reflective features. In this work we goes a step further by using AOP to modularize the scattered code of the reflective features and, as a consequence, to improve adaptability of reflective-based middleware since the reflective code can be removed and inserted according to the target application. This is an important differentiator of our work comparing to other non-AOP and AOP middleware implementations.

The idea of refactoring middleware implementation using AOP has been explored by certain authors [19-22]. The principles of Horizontal Decomposition (HD) were proposed in [20] in order to address the so-called “feature convolution” problem – the lack of modularity in the implementation of middleware features. This work focuses on defining horizontal decomposition guidelines and it applies the HD principles to implement ORBacus using AspectJ. It defines the functionalities that compose the middleware core and classifies them in three layers (IDL layer, messaging layer that defines synchronous invocations, and transport and protocol layer that implements IIOP) and the functionalities that can be represented by aspects: oneway invocation that supports asynchronous invocation, dynamic typing that implements reflective composition of remote invocations, the encoding conversion mechanism, and the local invocation supports. In [5] the HD principles are also applied to the Prevayler database system in order to validate the HD principles.

Our research differs from HD-specific studies in the sense our main goal of our refactoring process is to improve middleware modularity while the other work aims to address performance. Second, we apply a number of modularity measures that evaluate the impact of AOP on the isolation of reflective features by comparing the AO and OO versions for the middleware system. In [21] a set of software metrics are also applied to the middleware implementation. However, reflective features are not exploited in this experimentation. The third difference is that the refactorization of the legacy middleware is incomplete. It does not provide a complete AO middleware. Our Open-Orb compliant implementation provides all the features of the reflective Open-Orb model.

Hannemann and Kiczales (HK) [28] have undertaken a qualitative study in which they have developed and compared Java and AspectJ [29] implementations of the 23 GoF patterns [1]. The basic idea was the identification of the common part of several patterns and the isolation of their implementations in aspectual modules. Nevertheless, for each of the 23 patterns they used a very simple example that made use of the pattern. In our work, we used design patterns in the Java and AspectJ versions of the middleware in order to guarantee we have good designs in both solutions. In the AspectJ version we used HK solutions for the patterns. However, since we use them in a real system, we have to make a number of changes in their solutions in order to use patterns in composition with others [8]. Our work aims at studying how these pattern implementations support the improvement of adaptability in the context of reflective middleware systems. In their study, HK undertook other types of analysis.

Alice [26] exploits the container concept available in middleware for component-based software development and proposes a combination of a minimal container with AOP and Java 1.5 Annotations. It uses the annotation facility to provide meta-information about components and services. It focuses on the improvement of container-based middleware, via AOP and Annotations. Our research differs from this one by focusing on the aspectization of reflective middleware. The evaluation of our implementation by a suite of software metrics is also an important difference from this work.

The paper [31] proposes a Modelware methodology that combines the MDA (Model Driven Architecture) approach and AOP in order to address middleware customization and improve perfomance. However, this paper also does not assess the interplay of AOP and reflective middleware modularization. Finally, combining AOP and reflection is not a new idea. However, most of the existing works in the literature applies reflection to support AOP (e.g. [32]). In this work we apply AOP to modularize reflective features of a middleware platform in order to modularize them and to leverage the adaptability support provided by reflective middleware.

5. FINAL REMARKS AND FUTURE WORK

In this paper we have applied AOP to improve the modularity of reflective middleware by aspectizing reflection-specific crosscutting concerns. We have discussed how the reflective middleware features of a middleware crosscut the middleware architecture. We have quantitatively compared two OpenORB-compliant reflective middleware implementations: an OO implementation in Java and an AO in AspectJ. The quantitative evaluation showed that the AO version of the middleware is superior to the OO version in most of the crosscutting concerns. The SoC metrics showed significant modularity improvements mainly in the reflective features.

As future work, we are planning to investigate to that extent the AO solution impacts positively and negatively on performance issues. We have made some initial performance measurements [35]. We have already found some preliminary limitations in the AspectJ version, such as the need to implement the binding manager as a singleton aspect [8], leading to serious performance bottlenecks [35]. In addition, we plan to directly compare AO solutions and reflective ones (e.g. [33]) for each of the middleware concerns (such as the binding state) analyzed in this first case study. Hence, we would be able to go a step forward on our empirical understanding about the comparison of AOP with other programming techniques (such as reflection) to implement modular reflective middleware.

ACKNOWLEDGEMENTS. Nelio Cacho and Alessandro Garcia are supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

REFERENCES

1. E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns*. Addison Wesley. ISBN 0.201-63361-2.
2. Metsker, S. J.: *Design Patterns Java™ Workbook*, Addison Wesley, 2002.
3. Filho, F., Rubira, C., Garcia, A. A Quantitative Study on the Aspectization of Exception Handling. Proc. ECOOP Workshop on Exception Handling in OO Systems, Glasgow, Scotland, July 2005.

4. Kiczales, G. et al. Aspect-Oriented Programming. Proc. of ECOOP'97, LNCS 1241, Finland, June 1997, 220-242.
5. Godil, I., Jacobsen, H. Horizontal Decomposition of Prevayler. In Proc. of CASCON 2005, Richmond Hill, Canada, October 2005.
6. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. LNCS Transactions on Aspect-Oriented Software Development, vol. 1, n. 1., LNCS 3880, Springer, 2006, pp. 36-74.
7. Garcia, A. et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940, Jan 2004.
8. Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. 5th International Conference on Aspect-Oriented Software Development (AOSD'06),Bonn, Germany, 20-24 March 2006.
9. IONA Technology, Orbix/E. http://www.iona.com/
10. Soares, S. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, Oct 2004.
11. Figueiredo, E. et al. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. Proc. of the 9th ECOOP Workshop on Quantitative Approaches in OO Soft. Engineering (QAOOSE.05), Glasgow, July 2005.
12. Bernstein. P. Middleware. Communications of the ACM, 39(2), February 1996.
13. Agha, G. Adaptive Middleware. Communications of the ACM, Vol. 45, No. 6, pp. 31-32, June 2002.
14. Tripathi, A. Challenges Designing Next-Generation Middleware Systems. Communications of the ACM, Vol. 45, No. 6, pp 39-42, June 2002.
15. Blair, G. S. et al. An Architecture for Next Generation Middleware. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer-Verlag, pp. 191-206, 1998.
16. Blair, G., Costa, F., Saikoski, K and Parlavantzas. The design and implementation of Open ORB version 2. IEEE Distributed Systems Online Journal, 2(6), 2001.
17. Kon, F. et al. The case for reflective middleware. Commun. ACM, v. 45, n. 6, p. 33–38, 2002.
18. Fitzpatrick, T., Blair, G., Coulson,G. S., Davies, N. and Robin,P. Supporting Adaptive Multimedia Applications through Open Binding. In Proceedings of 4th International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, Maryland, US, May 1998.
19. Zhang, C. and Jacobsen, H. A. Quantifying aspects in middleware platforms. In: Proceedings of the 2nd international conference on Aspect-oriented software development. [S.l.]: ACM Press, 2003. p. 130–139. ISBN 1-58113-660-9.
20. Zhang, C.; Gao, D. and Jacobsen, H. A. Towards just-in-time middleware architectures. In: Proceedings of the 4th international conference on Aspect-oriented software development. ACM Press, 2005. p. 63–74. ISBN 1-59593-043-4.
21. Zhang, C. and Jacobsen, H. (2003) Re-factoring Middleware with Aspects. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, pp. 1243-1262.
22. Hunleth, F., Cytron, R. abd Gill, C. Building Customizable Middleware using Aspect Oriented Programming. In OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. 2001. Tampa, Florida.
23. Maes, P. Concepts and Experiments in Computational Reflection. In Proceedings of OOPSLA'87, pages 147-155, Orlando, Florida, 1987.
24. Kon, F. et al. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective Orb. Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing, 2000.
25. Colyer, A. and Clement, A. Large-Scale AOSD for middleware. In 3rd International Conference on Aspect-oriented Software Development (AOSD'04), pp. 56-65, Lancaster, UK, 2004.
26. Eichberg, M. and Mezini, M. Alice: Modularization of Middleware using Aspect-Oriented Programming. Software Engineering and Middleware (SEM), 2004
27. Sun Microsystems, PersonalJava Application Environment. http://java.sun.com/product/Personaljava
28. Hannemann, J.; Kickzales, G. Design Pattern Implementation in Java and AspectJ, Proc. 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02). Seattle, USA, 2002, pp. 161-173.
29. The AspectJ Team. The AspectJTM Programming Guide. http:/www.eclipse.org/aspectj.
30. Figueiredo, E., Garcia, A, Lucena, C. AJATO: an AspectJ Assessment Tool. Proc. of the ECOOP.06, Demo Session, Nantes, France, July 2006. http://www.teccomm.les.inf.puc-rio.br/emagno/
31. Zhang, C., Gao, D., Jacobsen, H.: Generic Middleware Substrate Through Modelware. Middleware 2005: 314-333.
32. N.Cacho, T.Batista (2005) Using AOP to Customize a Reflective Middleware. Int'l Symposium on Distributed Objects and Applications - DOA, Agia Napa, Cyprus, November 2005, LNCS, Volume 3761, Pages 1133 – 1150.
33. L. Ferreira, C. Rubira, "The Reflective State Pattern". Proc. of the 5th Languages of Programs Conference (PLoP'98), August 98, Monticello, EUA.
34. L. Capra, W. Emmerich and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering* 29(10): pp. 929--944, Oct 2003.
35. *Reflective middleware with AOP*. http://www.consiste.dimap.ufrn.br/~cacho/openorbresults.html