A Middleware Transparent Approach for Developing CORBA-based Distributed Applications

Brahmila Kamalakar and Sudipto Ghosh
Department of Computer Science
Colorado State University
Fort Collins CO 80523, USA
{brahmila,ghosh}@cs.colostate.edu

ABSTRACT

Complex distributed applications are developed using a variety of middleware technologies. Design and evolution of an application and subsequent migration from one middleware technology to another is cumbersome because of the high degree of coupling between the design of business functionality and middleware functionality. We propose an MDA compliant middleware transparent software development approach to address this problem. We encapsulate as generic aspect models all middleware functionality that would otherwise crosscut elements of business functionality. We bind these generic aspect models to an application-specific context and map them to code aspects. We weave the code aspects with the implementation of business functionality to obtain the complete application. Our approach enables the reuse of generic middleware aspect models in multiple applications. Developers can reuse designs and code corresponding to the business functionality and quickly migrate to new middleware technologies. We illustrate the approach with a CORBA-based application.

Keywords

CORBA, Model Driven Architecture, OMG, UML, aspect-oriented software development, distributed computing, modeling and meta-modeling, middleware technologies.

1. INTRODUCTION

The rapid growth of the Internet has resulted in widespread use of distributed applications that communicate with the help of middleware. Middleware features are often scattered across and tangled with modules of system design. In client-server systems, client and service classes include middleware functionality that enables clients to access the remote services in a transparent manner. In peer-to-peer (P2P) applications, each peer has middleware functionality that enables itself to transparently access other peers in the network.

Using current software development techniques, application design and implementation becomes tightly coupled with the specific middleware technology that is incorporated into the application. Even though certain middleware services may be provided as components (e.g., security), there are other middleware features (e.g., events and transactions) that crosscut these components. The crosscutting nature of middleware makes understanding, analyzing and changing middleware features difficult. Since businesses need to keep up with advances in middleware technology, entire applications need to be redesigned and reimplemented to migrate from one middleware technology to another.

We propose a middleware transparent software development (MTSD) approach that decouples the design of middleware specific features from the design of core business functionality. The approach uses aspect-oriented modeling and programming techniques because they provide the necessary constructs for encapsulating crosscutting design and code elements. Software developers design primary models of the core application functionality. Elements of the application that are specific to the middleware are modeled separately as aspects and seamlessly woven into the application later in the development process. MTSD eases the evolution of distributed applications, supports easy incorporation of new middleware technologies, and enables reuse of high-level application design and architectures that are independent of the middleware. MTSD supports the OMG's MDA initiative. In this paper, we present the MTSD approach and illustrate it with the design and implementation of a CORBA application.

We summarize related work in Section 2. We provide background information on CORBA and our modeling notation in Section 3. We describe the approach in Section 4 and illustrate it with a CORBA example in Section 5. We present our conclusions and outline directions for future work in Section 6.

2. RELATED WORK

Aspect-oriented software development [14, 15] has introduced aspect languages like AspectJ and Aspect C# which help in abstracting and encapsulating crosscutting concerns at the programming level. Simmonds et al. [19] captured Jini middleware details in the form of code aspects. Pichler et al. [17] demonstrated the use of aspects with the Enterprise Java Beans container architecture. Zhang and Jacobsen [21] analyzed the use of aspects in middleware architectures and quantified crosscutting concerns in the implementations of middleware applications.

Bussard [2] described the encapsulation of CORBA features as code aspects and proposed the creation of a library of aspects for different CORBA features to ease the development of CORBA applications. He observed composition problems when certain kinds of aspects are composed together with the same application and proposed the definition of another aspect to manage the composition of such aspects. Hunleth [13] proposed the creation of an Aspect-IDL for CORBA to support several new types of AspectJ introductions: interface method and field, interface, super class, structure field, oneway specifier, and IDL typedefs and enumerations.

Code aspects are usually implemented for a specific application and are not reusable. This limitation can be overcome by describing distributed systems at a higher level of abstraction. The MDA initiative [18] employs design level abstraction to describe software systems. In the MDA approach, the Platform Independent Model (PIM) captures the functionality and behavior of the application free from the middleware technology. Integration of middleware technology specific mappings with the PIMs yields the Platform Specific Models (PSM).

Clarke et al. [3, 4, 5] use the subject-oriented modeling approach using composition patterns to capture reusable patterns of cross-cutting behavior at the design level. Each requirement is treated as a separate design subject. Design subjects are composed to obtain the complete system design. Subjects may also be mapped to AspectJ aspects. The approach has some limitations in the types of compositions that can be performed.

France et al. [7] propose an aspect-oriented modeling (AOM) approach in which software designers specify primary models (base functionality), aspect models (non-orthogonal crosscutting functionality) and composition directives to obtain the integrated design. Cross-cutting design concerns are captured in aspect models using the Role-Based MetaModeling language [6]. France et al. [7, 9, 10, 11] use AOM to develop applications with security and other dependability concerns. We adopt the AOM approach for the development of middleware-based applications.

3. BACKGROUND

In this section, we present the necessary background information on CORBA and the Role-Based Metamodeling Language (RBML) that is used to specify aspect models.

3.1 CORBA

The Common Object Request Broker Architecture (CORBA) [1, 12, 16, 20] is an OMG standard for open distributed object computing. CORBA enables communication between applications irrespective of the type of programming language and hardware platform used. The object request broker (ORB) provides a mechanism for transparently conveying client requests to service object implementations in heterogeneous distributed environments. When a client invokes an operation, the ORB locates the object implementation, delivers the request to the object, and returns the response to the client. The ORB is also responsible for parameter marshaling, fault recovery and security.

Developers use the CORBA interface definition language (IDL) to describe the interfaces of distributed objects that are implemented in a programming language, such as Java, C++, or Smalltalk. Language mappings defined for each programming language specify how the IDL interfaces are mapped to constructs of the programming language. The CORBA IDL compiler converts the interface definitions in the IDL to the stubs and skeletons in the target programming language. Stubs and skeletons facilitate location transparency. The object adapter assists the ORB with delivering requests to the server object implementations. It also activates and deactivates servant objects and links them with the ORB. The Portable Object Adapter (POA) is a server side component that allows the construction of CORBA server applications that are portable across ORB implementations.

In our work, we use the JacORB 1.4.1 implementation of CORBA for the Java programming language. In JacORB, the following Java interfaces and classes are generated for any interface named *InterfaceType*:

1. InterfaceType.java : This contains the declaration of the methods in the interface.
2. InterfaceTypeHolder.java : This provides support to handle the IDL inout, and out parameters.
3. InterfaceTypeHelper.java : This contains various static methods for type-specific operations such as the *narrow* method, which narrows object references of type *CORBA.Object* to the specific interface type.
4. InterfaceTypePOA.java : This is the skeleton class.
5. _InterfaceTypeStub.java : This contains the stub code to create a client side proxy for the object implementation.
6. InterfaceTypeOperations.java, InterfaceTypePOATie.java : These are used for the CORBA *Tie* mechanism on the server side.

There are two ways of associating an object implementation class (*InterfaceTypeImpl*) with a skeleton class. One is by using inheritance, with the POA skeleton class. The implementation class extends the servant base class (*InterfaceTypePOA*). This is the approach used in the paper. The other is by using delegation, which requires the *InterfaceTypePOATie* and *InterfaceTypeOperations* classes. The delegation approach is used when the implementation class needs to inherit from more than one class. The implementation class in this case implements *InterfaceTypeOperations*.

For any CORBA application, the IDL interface, the object implementation class, the main server class, and the main client class are implemented by the application developer. The object implementation class contains the method implementations of the interfaces in the IDL. The server class initializes the environment, creates the implementation object, makes it available to clients, and listens for events. The client requests services by invoking methods defined in the IDL on the service objects. All the entities that need to be developed by the application developer are coupled with CORBA specific details. Our approach is directed at decoupling these middleware concerns from the core functionality of the application.

3.2 Role Based Meta-Modeling Language

We treat a design aspect model as a pattern that characterizes a family of design solutions for a crosscutting feature. We use the RBML to specify families of UML models. The RBML defines a sub-language of the UML and provides a pattern specification notation which is used to describe the middleware design aspect models. An RBML specification is a structure of roles, where a role defines properties that must be satisfied by conforming UML model elements. The patterns described in this paper consist of the following RBML specifications:

1. A Static Pattern Specification (SPS) that characterizes conforming class diagrams.
2. A set of Interaction Pattern Specifications (IPSs) that characterize conforming interaction diagrams.

A pattern's SPS defines a family of conforming UML class diagrams and the IPS defines a family of conforming UML sequence diagrams. Details of the RBML notation can be seen in France et al. [6].

3.2.1 Static Pattern Specification:

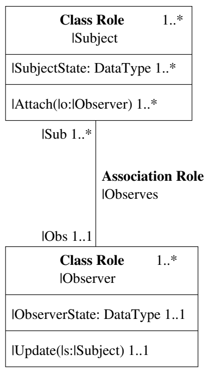
An SPS consists of classifier and relationship roles, where a classifier role is connected to other classifier roles by relationship roles. Properties in a classifier role are expressed in three forms:

1. *StructuralFeature* roles specify structural features of conforming classifiers. A structural feature can be an attribute. *StructuralFeature* roles can be associated with constraint templates that are used to produce OCL constraints associated with conforming structural elements.
2. *BehavioralFeature* roles specify behavioral features of conforming classifiers. A behavioral feature can be implemented by one or more operations. *BehavioralFeature* roles can also be associated with constraint templates that are used to produce operation specifications associated with conforming operations.
3. Metamodel-level constraints are well-formedness rules that restrict the form of conforming model elements.

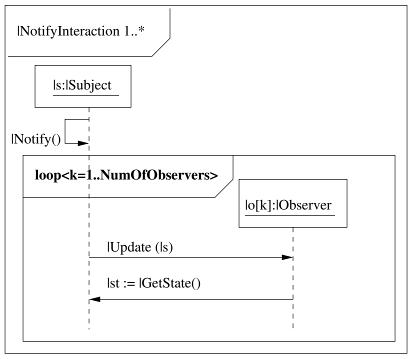
Figure 1(a) shows a partial SPS for a variant of the Observer design pattern [8]. This pattern specifies one or more *Observer* classes and one or more *Subject* classes in a conforming class diagram such that each *Observer* class is associated with exactly one *Subject* class, and vice versa.

The "|" is used to indicate roles in RBML specifications. The SPS shown in Figure 1(a) consists of two class roles, *Subject* and *Observer*, that are connected by an association role *observes.* Each role in an SPS can be associated with a binding multiplicity that restricts the number of conforming elements that can be bound to the role in a conforming model.

A class that conforms to the *Subject* role can have one or more structural features that conform to the *SubjectState*


Class Role 1..*
|Subject
|SubjectState: DataType 1..*
|Attach(|o:|Observer) 1..*
|Sub 1..*
Association Role
|Observes
|Obs 1..1
Class Role 1..*
|Observer
|ObserverState: DataType 1..1
|Update(|s:|Subject) 1..1

(a) Observer SPS



(b) Observer IPS

Figure 1: A Variant of the Observer Pattern specified in RBML.

role and one or more behavioral features that conform to the *Attach* role. The association role *Observes* specifies associations between *Subject* and *Observer* classes. Each end of an association role has an association-end role. The binding multiplicity on the *Obs* association-end role (1..1) specifies that a conforming *Observer* class must be part of only one *Observes* association. However, the binding multiplicity on the *Sub* association-end role (1..*) specifies that a conforming *Subject* can be part of one or more *Observes* association.

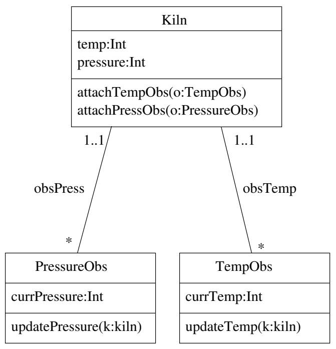
3.2.2 Interaction Pattern Specifications:

IPSs are used to define interactions between pattern participants. An interaction role is a structure of lifeline and message roles. Each lifeline role is associated with a classifier role in an SPS: a participant that plays a lifeline role is an instance of a classifier that conforms to the classifier role in the SPS. A message role is associated with a behavioral feature role in an SPS: a conforming message specifies a call to an operation that conforms to a behavioral feature role. Figure 1(b) shows the IPS describing the pattern of interactions that take place as a result of invoking a subject's *Notify* operation. In the figure, the lifeline role $|s : |Subject$ represents instances of a class that conforms to the classifier role Subject in Figure 1(a) and the message role *Update* represents asynchronous calls to operations that conform to the feature role *Update.* The *Notify* behavior results in calls to operations that conform to the *Update* role for each observer associated with the subject. Each *Observer* then calls the operation that conforms to the *GetState* role in the subject. The behavioral roles *GetState* and *Notify* are not shown in the SPS.

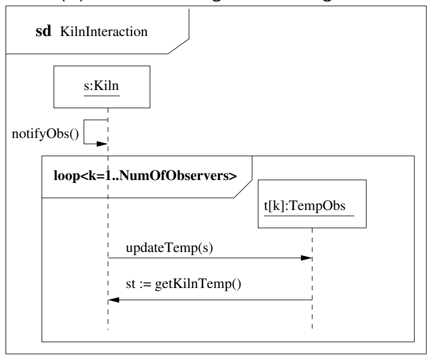
The loop structure shown in Figure 1(b) allows one to specify iterative behavior in a concise manner. The lifeline labeled $o[k]$ represents the k^{th} observer attached to the subject.

3.2.3 Obtaining Conforming Models from the SPS and IPS:

This process involves binding application-specific model elements and roles. Figure 2 shows part of a model obtained this way from the Observer pattern described in Figure 1.



(a) Conforming class diagram



(b) Conforming sequence diagram

Figure 2: Conforming UML Diagrams for the Observer Pattern.

The class diagram shown in Figure 2(a) describes a system in which a kiln sensor records the kiln's temperature and pressure. The sensor is linked to temperature and pressure observers that monitor kiln temperature and pressure. The *SubjectState* role binding multiplicity allows one or more model elements to be bound to it. This role is bound to two attributes, *temp* and *pressure*. The binding multiplicity associated with the *Sub* association-end can be associated with a class that conforms to the *Subject* role. The two association-ends attached to the *Kiln* class are bound to the *Sub* role.

The sequence diagram shown in Figure 2(b) is obtained by binding model elements representing kiln system concepts to roles in the observer IPS. One sample sequence diagram that conforms to the *NotifyInteraction* IPS is shown.

4. OVERVIEW OF THE MIDDLEWARE TRANSPARENT APPROACH

The MTSD approach is illustrated in Figure 3. In this approach, the application developer models the application free from middleware concerns in a *Middleware Transparent Design* (MTD) model. The MTD model contains UML class diagrams and interaction diagrams. Other views (e.g., statecharts and activity diagrams) may also be used. In the MDA terminology, the MTD is the PIM.

Middleware specific features are localized in an aspect model. Generic aspect models in the form of aspect libraries are provided by the middleware platform vendor. These models are described using the RBML in terms of SPSs and IPSs. There is one aspect model for each feature (e.g., connectivity, directory service, security, and replication). In this paper, we illustrate the generic aspect model for CORBA connectivity in Section 5.

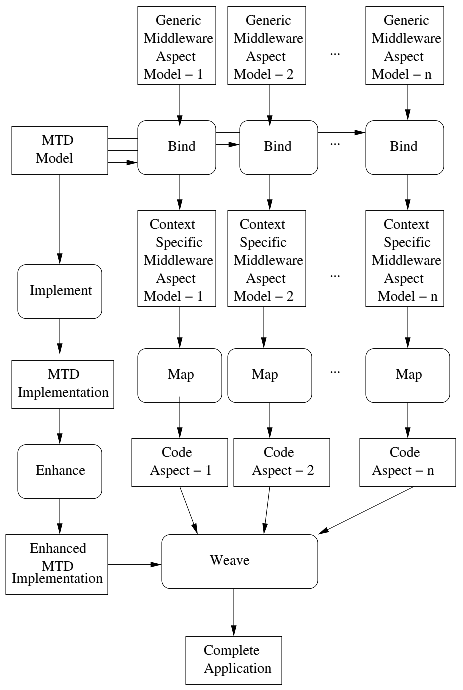

Generic Middleware Aspect Model – 1
Generic Middleware Aspect Model – 2
...
Generic Middleware Aspect Model – n
MTD Model
Bind
Bind
...
Bind
Context Specific Middleware Aspect Model – 1
Context Specific Middleware Aspect Model – 2
...
Context Specific Middleware Aspect Model – n
Implement
MTD Implementation
Map
Map
...
Map
Enhance
Code Aspect – 1
Code Aspect – 2
Code Aspect – n
Enhanced MTD Implementation
Weave
Complete Application

Figure 3: The MTSD Approach.

The application developer specifies the bindings from the generic aspect models to the application context and generates the context-specific aspect models. The generation can be automated in part; it still requires binding information as input from the application developer.

These models are implemented in an aspect language (currently in AspectJ). The developer transforms an aspect model to code aspects with the help of mappings that convert aspect model constructs to AspectJ constructs.

The application developer implements the MTD model. The MTD implementation then needs to be converted into an *Enhanced MTD implementation* to make it ready for aspect weaving for a specific middleware platform. Different middleware platforms need different application architectures. For example, Jini requires that a proxy object be implemented by the developer. The code for the proxy object is usually written as a Java inner class. Java RMI requires proxy stubs to be automatically generated from the server implementation. The client implementation in the MTD does not possess knowledge about the nature of proxies and assumes that the server object is local. Thus, the client (and the server in some cases) may need to be modified to adapt them to the middleware platform used in the application. We have automated the enhancement process for CORBA and Jini. We are investigating if the enhancement can be automated for other middleware platforms. The enhanced MTD implementation is woven with code aspects using the AspectJ compiler.

The generic aspect models can be reused to develop other applications. The mappings from the context-specific aspect models to code aspects in AspectJ are described in terms of generic aspect models and AspectJ constructs. Thus, these mappings can also be implemented in a tool and reused. The mappings defined to enhance an MTD implementation and convert it into a form required for CORBA compliance are also reusable and can be implemented in a tool. The MTD models and implementations can be reused with aspects for other middleware technologies when migrating from one middleware technology to another.

5. MTSD FOR CORBA

We illustrate our MTSD approach by incorporating CORBA connectivity into a simple *HelloWorld* application. Connectivity between the client and the service object is achieved by using the Interoperable Object Reference (IOR) of the service object. There are three ways to obtain the IOR:

1. *string_to_object*: Sometimes, the IOR is available in the form of a string (*stringified* IOR) and stored in a file. If the client can obtain the stringified IOR, it invokes this call to convert it into an IOR.
2. *resolve_initial_references*: This call returns the IORs of pre-defined service objects, such as the *Naming Service* and *RootPOA*. For example, to use the *RootPOA*, the client gets its reference from the ORB by invoking `resolve_initial_references(‘‘RootPOA’’)`.
3. *resolve*: With the reference of the naming service, a client can look up the IOR of a required service.

In this paper, we use option 2 to get the *RootPOA* and option 1 to get the IOR from its stringified form.

5.1 MTD for the HelloWorld Application

The MTD in Figure 4 describes the functionality of a simple application using a UML class diagram. The MTD contains a Java server class called *HelloWorldServer* that implements an interface called *HelloWorldInterface*. This interface contains a method, `hello()`, which returns the string message *"Hello World"*. The Java client class, *HelloWorldClient*, invokes the `hello()` method on the server.

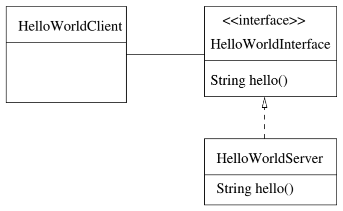
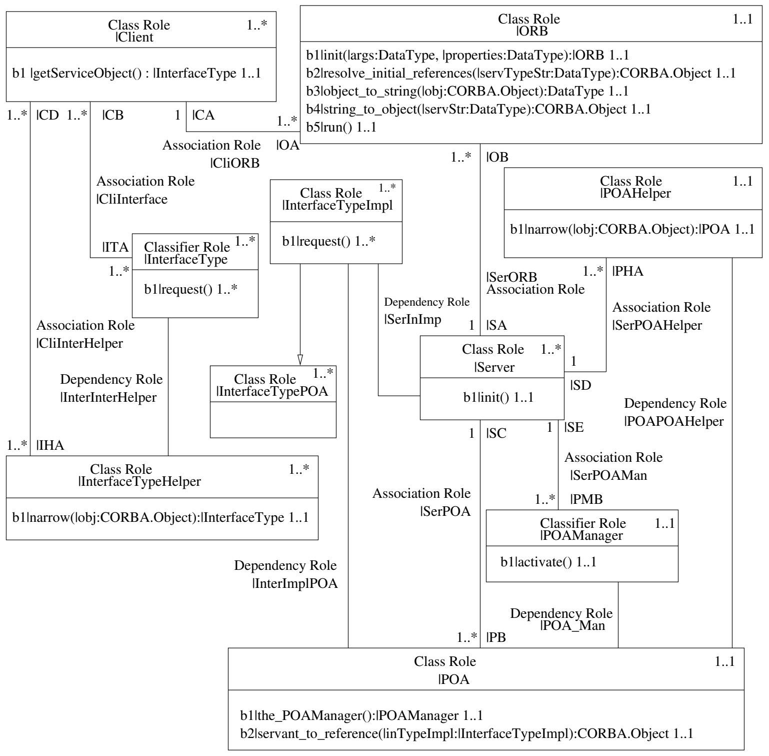


Figure 4: MTD for the HelloWorld Application.



Class Role |Client 1..*
b1 |getServiceObject() : |InterfaceType 1..1
Class Role |ORB 1..1
b1|init(|args:DataType, |properties:DataType):|ORB 1..1
b2|resolve_initial_references(|servTypeStr:DataType):CORBA.Object 1..1
b3|object_to_string(|obj:CORBA.Object):DataType 1..1
b4|string_to_object(|servStr:DataType):CORBA.Object 1..1
b5|run() 1..1
1..* |CD 1..* |CB 1 |CA 1..*
Association Role |CliORB
|OA
1..* |OB
Association Role |CliInterface
Class Role |InterfaceTypeImpl 1..*
b1|request() 1..*
Class Role |POAHelper 1..1
b1|narrow(|obj:CORBA.Object):|POA 1..1
|ITA
Classifier Role |InterfaceType 1..*
1..*
b1|request() 1..*
|SerORB Association Role
1..* |PHA
Association Role |SerPOAHelper
Association Role |CliInterHelper
Dependency Role |SerInImp
1 |SA
Class Role |Server 1..*
b1|init() 1..1
1
|SD
Dependency Role |InterInterHelper
Class Role |InterfaceTypePOA 1..*
Dependency Role |POAPOAHelper
1 |SC
1 |SE
Association Role |SerPOAMan
1..* |IHA
Class Role |InterfaceTypeHelper 1..*
b1|narrow(|obj:CORBA.Object):|InterfaceType 1..1
Association Role |SerPOA
1..* |PMB
Classifier Role |POAManager 1..1
b1|activate() 1..1
Dependency Role |InterImpl|POA
Dependency Role |POA_Man
1..* |PB
Class Role |POA 1..1
b1|the_POAManager():|POAManager 1..1
b2|servant_to_reference(|inTypeImpl:|InterfaceTypeImpl):CORBA.Object 1..1

Figure 5: SPS for CORBA Connectivity using IOR.

5.2 Generic Aspect Model for IOR Mechanism

The generic aspect model for CORBA connectivity using the IOR mechanism is specified with an SPS (Figure 5) and two IPSs (Figures 6 and 7).

5.2.1 SPS for CORBA Connectivity:

We encapsulated the CORBA specific functionality pertaining to the IOR mechanism as a reusable pattern for use with any CORBA application. Figure 5 shows the SPS.

The *ORB*, *POA*, and *POAHelper* class roles are played by classes provided by the CORBA implementation (e.g., JacORB). The *POAManager* classifier role is played by an interface, which is defined by CORBA. The *ORB* classifier role has the following behavioral feature roles:

1. `init`: performs initialization of the ORB.
2. `resolve_initial_references`: returns the references of predefined services like the naming service.

3. `object_to_string`: handles the type conversion of the IOR from
CORBA.Object to a string.
4. `string_to_object`: handles the type conversion of the IOR from string to *CORBA.Object*.
5. `run`: starts the ORB.

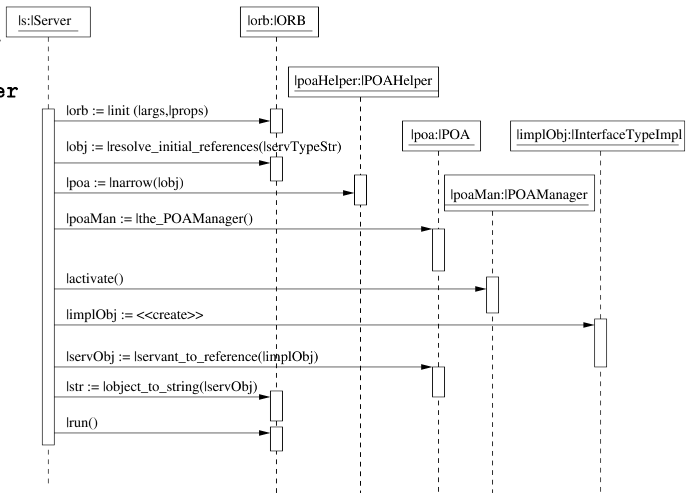
The roles for which the instances are provided by CORBA are described in this paragraph. The *POA* class role has behavioral feature roles `servant_to_reference` and `the_POAManager`. The responsibility of the `servant_to_reference` behavioral role is to provide the IOR of the service object. The `the_POAManager` role is responsible for returning a reference of type *POAManager*. The *POAManager* classifier role has the `activate` behavioral role to activate the *POA*. The classifier role *POAHelper* has the behavioral role `narrow`, which converts an object of type *CORBA.Object* to type *POA*. The binding multiplicity on the roles *ORB*, *POA*, *POAManager* and *POAHelper* is 1..1, signifying that a conforming class diagram can have only one instance of each of these roles.

The roles that need to be merged with MTD-specific classes at the time of binding are described in this paragraph. The *InterfaceTypeHelper* and *InterfaceTypePOA* class roles are specific to an interface type. The classes conforming to these roles are generated by the CORBA-IDL compiler from the interface definition. The *InterfaceTypeHelper* class role is played by the helper class pertaining to the interface type. It contains the behavioral role `narrow`. The *InterfaceTypePOA* class role is played by the servant base class (or skeleton class) pertaining to the interface type. The *InterfaceType* classifier role is played by the interface generated by CORBA-IDL compiler. It contains the behavioral role `request`, whose instances are the services offered to the client. Since there can be many operations that play the role of `request`, the multiplicity associated with this role is 1..*. The *InterfaceTypeImpl* class role is played by the class which contains the implementation of the service and is provided by the application developer. The *InterfaceTypeImpl* role implements the instances of the behavioral role `request`. The binding multiplicity on the *InterfaceTypeHelper*, *InterfaceTypePOA*, *InterfaceType* and *InterfaceTypeImpl* roles is 1..*. This signifies that a conforming class diagram can have any number of instances (at least 1) of each of these roles.

The *Client* class role is played by classes that request services. It contains the `getServiceObject` behavioral role that returns a reference to the service object. CORBA-specific functionality corresponding to the `getServiceObject` role is subsequently added to actual clients in the MTD implmentation using AspectJ aspects. The *Server* class role is played by the class that creates the IOR of the service object. It contains the behavioral role `init` that instantiates the service object and publishes its IOR. The binding multiplicity on the *Client* and *Server* roles is 1..*. This signifies that a conforming class diagram can have any number of instances (at least 1) of each of these roles. This is because a distributed system can have more than one *Client* and *Server*.

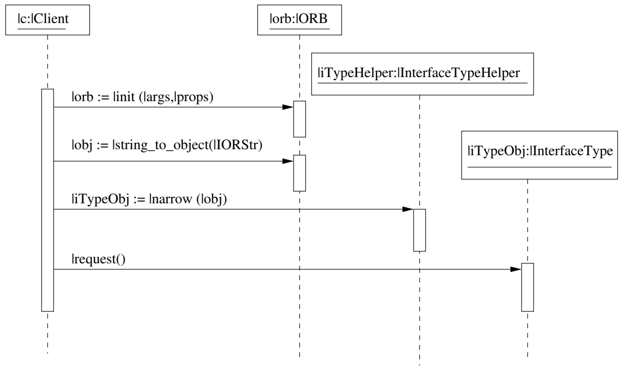
We have use-dependency roles between *POAHelper* and *POA*, *InterfaceTypeHelper* and *InterfaceType*, *POA* and *POAManager*, *Server* and *InterfaceTypeImpl*, and *POA* and *InterfaceTypeImpl*. The corresponding relationships in a conforming class diagram must be use-dependency relationships.

Behavioral roles are labeled using strings of the form "b#" (e.g., b1, b2, ...), where "b" denotes behavioral feature roles. These labels are used in application models to mark features (possibly more than one) that are bound to each role.



|s:|Server
|orb:|ORB
|poaHelper:|POAHelper
|poa:|POA
|implObj:|InterfaceTypeImpl
|poaMan:|POAManager
|orb := |init (|args,|props)
|obj := |resolve_initial_references(|servTypeStr)
|poa := |narrow(|obj)
|poaMan := |the_POAManager()
|activate()
|implObj := <<create>>
|servObj := |servant_to_reference(|implObj)
|str := |object_to_string(|servObj)
|run()

Figure 6: Server-side IPS for CORBA Connectivity using IOR.



|c:|Client
|orb:|ORB
|iTypeHelper:|InterfaceTypeHelper
|iTypeObj:|InterfaceType
|orb := |init (|args,|props)
|obj := |string_to_object(|IORStr)
|iTypeObj := |narrow (|obj)
|request()

Figure 7: Client-side IPS for CORBA Connectivity using IOR.

5.2.2 *IPSs for CORBA Connectivity:*

We describe the IPSs that specify the pattern of interactions taking place as a result of invoking the `init` operation in the *Server* and the `getServiceObject` operation in the *Client*.

Figure 6 depicts the details of server side interactions. The *Server* class role contains the `init` role. The lifeline role `|s: |Server` represents an instance of a class that conforms to the classifier role *Server* in Figure 5. First, the *Server* initializes an instance of the *ORB* and obtains its reference. The *Server* then gets the IOR of the *POA* instance by invoking `resolve_initial_references` and narrowing it down. The *Server* uses `the_POAManager` to obtain a reference of the *POAManager*. The *Server* activates the *POA* instance by invoking `activate` on the *POAManager*. The service is instantiated when the *Server* creates an instance of *InterfaceTypeImpl*. The service IOR is obtained by invoking `servant_to_reference` on the *POA*. The IOR is converted to a string by invoking `object_to_string` on the *ORB*. The *Server* starts the *ORB* by invoking `run`.

Figure 7 shows the details of the `getServiceObject` role in the *Client* class using which the client obtains the IOR of the required service. The lifeline role `|c: |Client` represents an instance of a class that conforms to the classifier role *Client* in Figure 5. The client first gets the stringified IOR (e.g., from a file), then initializes the *ORB* and obtains its reference. The client converts the stringified IOR to type `CORBA.Object` by invoking `string_to_object` and narrows it to the appropriate reference type using `narrow` in the *InterfaceTypeHelper*.

5.3 Context-specific Aspect Model for the IOR Mechanism

The IOR pattern is used to produce a set of UML diagrams for a specific application context. Figure 8 shows the context-specific class diagram obtained from the SPS. The stereotypes on the diagram elements indicate the SPS roles they conform to. Table 1 shows how elements are either (1) added to the class diagram by stamping them out from the SPS, or (2) obtained by explicitly binding with existing diagram elements in the MTD class diagram, or (3) generated by the IDL compiler, and thus, stamped out or bound to existing diagram elements. In Figure 5, the binding multiplicity associated with the *CB* association-end role indicates that one or more association-ends can be associated with a class that conforms to the *Client* class role in Figure 8. This implies that many client classes (only *HelloWorldClass* in our example) can be associated with one service type.

The context-specific interaction diagrams are obtained from the IPSs by binding appropriate roles to *HelloWorld* application elements. The lifeline and message bindings are the same as those used to produce the class diagram. The message parameters are specified by the developer. Figures 9 and 10 show the sequence diagrams conforming to the IPSs in Figures 6 and 7 respectively.

5.4 MTD Implementation

A typical Java implementation of the *HelloWorld* MTD contains the classes *HelloWorldClient* and *HelloWorldServer*. The *HelloWorldServer* class implements the methods exposed by *HelloWorldInterface*. An instance of the *HelloWorldClient* class gets a reference to a *HelloWorldServer* and invokes the `hello()` method on it.

5.5 Mapping Context-Specific Aspect Models to Code Aspects

Figures 11 and 12 show the implementation of the context-specific aspect model for the IOR mechanism as code aspects in AspectJ. We use AspectJ's join-point and advice mechanism.

The `init` behavior on the server side is encapsulated in the `init` method defined in the *ServerAspect*. The `init` method creates the servant IOR string and stores the IOR in a location accessible to the client. CORBA requires the implementation class to extend the *InterfaceTypePOA* (skeleton class). The *ServerAspect* does this by using AspectJ's inter-type declaration "declare parents". The `init` method needs to be called before the `main` method. Hence, we define an *execution* primitive pointcut on the execution of `main`. The `init` method is invoked in the *before* advice associated with the pointcut.

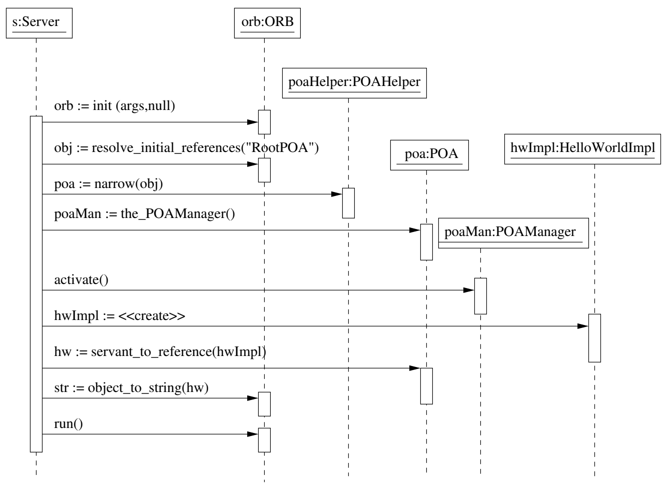


Figure 9: Server-side IOR Sequence Diagram for HelloWorld.

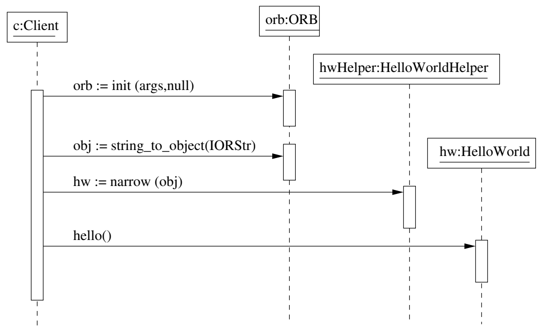
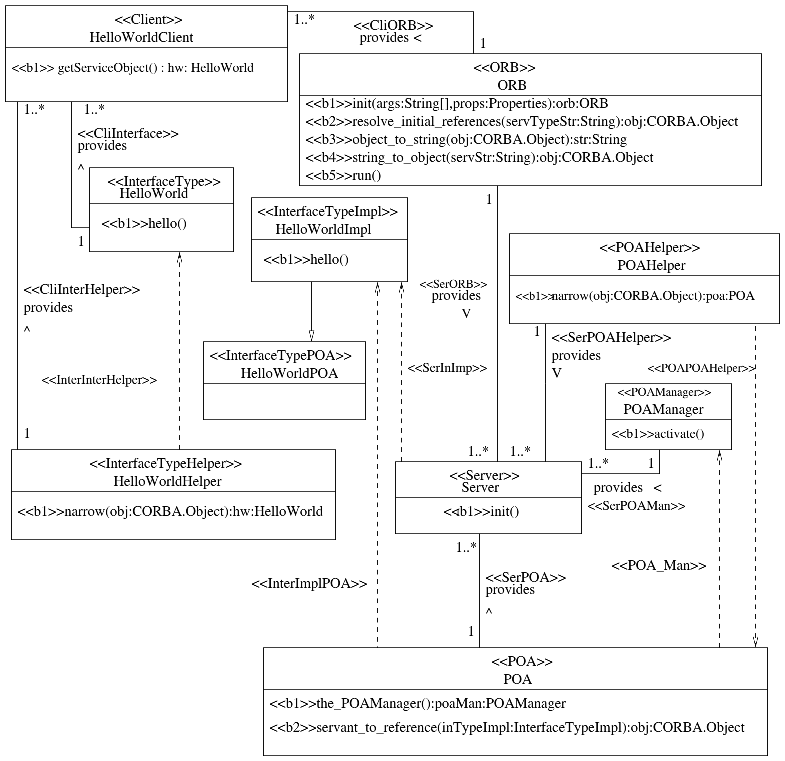


Figure 10: Client-side IOR Sequence Diagram for HelloWorld.

The transformations required to generate the *ServerAspect* are as follows:

1. Import the appropriate packages.
2. Include the appropriate `declare parents` clause.
3. Define the `init` method with all the steps shown in Figure 11.
4. Define the pointcut on the execution of `main`.
5. Define the associated *before* advice.

The `getServiceObject` behavior on the client side is encapsulated in another aspect, *ClientAspect*. The `getServiceObject` method in the *HelloWorldClient* class is responsible for re-



<<Client>> HelloWorldClient
<<b1>> getServiceObject() : hw: HelloWorld
<<CliORB>> provides <
<<ORB>> ORB
<<b1>>init(args:String[],props:Properties):orb:ORB
<<b2>>resolve_initial_references(servTypeStr:String):obj:CORBA.Object
<<b3>>object_to_string(obj:CORBA.Object):str:String
<<b4>>string_to_object(servStr:String):obj:CORBA.Object
<<b5>>run()
<<CliInterface>> provides
<<InterfaceType>> HelloWorld
<<b1>>hello()
<<InterfaceTypeImpl>> HelloWorldImpl
<<b1>>hello()
<<POAHelper>> POAHelper
<<b1>>narrow(obj:CORBA.Object):poa:POA
<<CliInterHelper>> provides
<<InterInterHelper>>
<<InterfaceTypePOA>> HelloWorldPOA
<<SerORB>> provides
<<SerInImp>>
<<SerPOAHelper>> provides
<<POAPOAHelper>>
<<POAManager>> POAManager
<<b1>>activate()
<<InterfaceTypeHelper>> HelloWorldHelper
<<b1>>narrow(obj:CORBA.Object):hw:HelloWorld
<<Server>> Server
<<b1>>init()
provides <
<<SerPOAMan>>
<<InterImplPOA>>
<<SerPOA>> provides
<<POA_Man>>
<<POA>> POA
<<b1>>the_POAManager():poaMan:POAManager
<<b2>>servant_to_reference(inTypeImpl:InterfaceTypeImpl):obj:CORBA.Object

Figure 8: Context-specific IOR Class Diagram for HelloWorld.

Table 1: Bindings for the HelloWorld Class Diagram

Role	Class diagram element	How to obtain
POA	*POA*	stamp out
`the_POAManager`	`the_POAManager`	stamp out
`servant_to_reference`	`servant_to_reference`	stamp out
POAHelper	*POAHelper*	stamp out
`narrow`	`narrow`	stamp out
ORB	*ORB*	stamp out
`init`	`init`	stamp out
`resolve_initial_references`	`resolve_initial_references`	stamp out
`object_to_string`	`object_to_string`	stamp out
`string_to_object`	`string_to_object`	stamp out
`run`	`run`	stamp out
POAManager	*POAManager*	stamp out
`activate`	`activate`	stamp out
InterfaceTypePOA	*HelloWorldPOA*	generate
InterfaceTypeHelper	*HelloWorldHelper*	generate
`narrow`	`narrow`	generate
InterfaceType	*HelloWorld*	generate
`request`	`hello()`	generate
InterfaceTypeImpl	*HelloWorldServer*	bind
`request`	`hello()`	bind
Client	*HelloWorldClient*	bind
`getServiceObject`	`getServiceObject`	stamp out
Server	*Server*	stamp out
`init`	`init`	stamp out

turning the servant object reference to the client. In the MTD implementation, this method is unaware of details regarding CORBA connectivity. We incorporate CORBA connectivity by defining a pointcut on the execution of the

```
package demo.helloworld;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public aspect ServerAspect {
   declare parents: HelloWorldImpl extends HelloWorldPOA;

   public void init(String[] arg) throws Exception {
      ORB orb = ORB.init( arg, null );
      POA poa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      poa.the_POAManager().activate();
      HelloWorldImpl hwImpl = new HelloWorldImpl();
      org.omg.CORBA.Object obj = poa.servant_to_reference(hwImpl);
      FileWriter out = new FileWriter("ior");
      out.write(orb.object_to_string(obj));
      out.close();
      orb.run();
   }

   pointcut serPt(Server s, String[] arg): execution(void Server.main(..))
                                        && target(s) && args(arg);

   before(Server s, String[] arg): serPt(s, arg) {
      try{
         init(arg);
      } catch(Exception e) {
         System.out.println(e);

      }

   }
}
```

Figure 11: Server IOR aspect in AspectJ

```
package demo.helloworld;
import java.io.*;
import org.omg.CORBA.*;
public aspect ClientAspect {
   pointcut cliPt(HelloWorldClient c, String[] arg): execution(*
   HelloWorldClient.getServiceObject(..)) && target(c) && args(arg);
   HelloWorld around(HelloWorldClient c, String[] arg): cliPt(c, arg) {
      try{
         String ior;
         ORB orb = ORB.init(arg,null);
         ior = new BufferedReader(new FileReader("ior")).readLine();
         org.omg.CORBA.Object obj = orb.string_to_object(ior);
         HelloWorld hw = HelloWorldHelper.narrow(obj);
         return hw;
      } catch(Exception e) {
         System.out.println("Exception" + e);
         return null;
      }
   }
}
```

Figure 12: Client IOR aspect in AspectJ

```
// Server.java

package demo.helloworld;
import java.io.*;

public class Server {
   public static void main(String[] args)
   {}
}
```

```
// HelloWorldImpl.java

package demo.helloworld;
public class HelloWorldImpl {
   public HelloWorldImpl( ) { }

   public String hello() {
      return "Hello World !";
   }
}
```

```
// HelloWorldClient.java

package demo.helloworld;
import java.io.*;

public class HelloWorldClient {

   public HelloWorld getServiceObject() {
      HelloWorld hw = (HelloWorld) new HelloWorldImpl();
      return hw;
   }

   public static void main(String args[]) {
      try {
         HelloWorldClient cli = new HelloWorldClient();
         HelloWorld hw = cli.getServiceObject();
         System.out.println( hw.hello());
      } catch(Exception ex) {
         System.err.println(ex);

      }

   }
}
```

Figure 13: Code snippets from the Enhanced Server, Client and HelloWorld Implementation

getServiceObject method with an execution primitive pointcut. The steps for CORBA connectivity are included in an *around* advice associated with the pointcut. The around advice returns the servant object reference to the client which calls the **hello** method on it.

The transformations required to generate the *ClientAspect* are as follows:

1. Import the appropriate packages.
2. Define a pointcut on the execution of **getServiceObject**.
3. Define the around advice with steps as shown in Figure 12.

5.6 Enhancing the HelloWorld MTD Implementation and Weaving with Aspects

Prior to weaving the MTD implementation with the code aspects, we need to enhance the MTD implementation to ensure CORBA compliance. Figure 13 shows the enhanced MTD implementation.

CORBA requires the declaration of the service interfaces in an IDL format. Hence, the Java interface written by the developer needs to be converted to the CORBA IDL format. The developer specifies the directory structure in which the IDL file must be placed. An IDL file corresponding to the Java interface may be generated automatically. On compiling the IDL file with an IDL compiler, the required CORBA files (*HelloWorldHelper*, *HelloWorldPOA* and *HelloWorld*) are created.

We rename the service implementation class *HelloWorldServer* provided by the application developer as *HelloWorldImpl*. The clause "**implements HelloWorldInterface**" is deleted from the *HelloWorldImpl* class because it must now "**extend HelloWorldPOA**". This extension is done in the *ServerAspect*.

We need to have a dummy **getServiceObject** operation in the *HelloWorldClient* so that the *around* advice specified in the *ClientAspect* can work. This new operation instantiates a service object and returns a reference of type *InterfaceType*. This is the same type as required by the advice. The operation needs to be called from the client's **main** method for the advice to execute once it is woven. In AspectJ, the behavior specified by the around advice is executed instead of the original operation, so the behavior of the dummy **getServiceObject** operation in the enhanced MTD implementation has no effect in the woven application.

Finally, the complete application is generated by weaving the client and server code with the respective aspects.

6. CONCLUSIONS AND FUTURE WORK

This paper presented the MTSD approach for the development of CORBA-based distributed applications. We applied MTSD to incorporate CORBA connectivity into an application design that was free from any CORBA-specific design elements. The design aspects representing the connectivity feature and the mapping to code aspects can be reused in other CORBA applications. The primary models of business functionality can be reused with other middleware technologies.

We have also successfully applied MTSD to CORBA applications where the naming service approach is used for locating services. We have used MTSD with the CORBA *Tie* mechanism as well.

Our next step is to apply MTSD to different application architectures and investigate the effect of using multiple aspects with one primary model. We will also extend the work to generate code mappings for aspect models of other CORBA features, such as security and fault tolerance. A comprehensive CORBA aspect library will be created. Further investigation will be carried out to apply MTSD to other middleware technologies. This will help us characterize properties of middleware features that make them isolatable as aspects. We are also working on the development of a prototype tool to automate the mapping process.

7. REFERENCES

[1] S. Baker. *CORBA Distributed Objects Using Orbix.* ACM press, Addison-Wesley, USA, 1997.

[2] L. Bussard. Towards a Pragmatic Composition Model of CORBA Services Based on AspectJ. In *Proceedings of ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia Antipolis and Cannes, France, June 2000.

[3] S. Clarke. "Extending Standard UML with Model Composition Semantics". *Science of Computer Programming*, 44(1):71–100, July 2002.

[4] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[5] S. Clarke and J. Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In *Proceedings of the International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 1998.

[6] R. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3), March 2004.

[7] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *To be published in IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, to appear*, 2004.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, Reading, MA 01867, 1995.

[9] G. Georg, R. France, and I. Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.

[10] G. Georg, R. France, and I. Ray. Designing High Integrity Systems using Aspects. In *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, Bonn, Germany, November 2002.

[11] G. Georg, I. Ray, and R. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.

[12] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, USA, 1999.

[13] F. Hunleth, R. Cytron, and C. Gill. Building Customizable Middleware Using Aspect Oriented Programming. In *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, USA, October 2001.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–353, Budapest, Hungary, June 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, June 1997.

[16] OMG — The Object Management Group. *Common Object Request Broker Architecture CORBA/IIOP 2.6.* OMG, 2002.

[17] R. Pichler, K. Ostermann, and M. Mezini. "On Aspectualizing Component Models". *Software Practice and Experience*, 33(10):957–974, August 2003.

[18] Richard Soley. MDA, An Introduction. *URL* `http://omg.org/mda/presentations.htm/`, 2002.

[19] D. Simmonds, S. Ghosh, and R. B. France. Middleware Transparent Software Development and the MDA. In *UML 2003 Workshop on SIVOES-MDA, to appear in Proceedings SIVOES 2003, Electronic Notes in Theoretical Computer Science, Elsevier*, San Francisco, CA, October 2003.

[20] A. Vogel and K. Duddy. *Java Programming with CORBA.* John Wiley and Sons, USA, 1998.

[21] C. Zhang and H.-A. Jacobsen. "Refactoring Middleware with Aspects". *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.