Middleware Transparent Development of Dependable CORBA Applications

Brahmila Kamalakar, Sudipto Ghosh, Peter Vile
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
{*brahmila,ghosh,vile*}*@cs.colostate.edu*

Abstract

Middleware technologies such as CORBA provide dependability features in the form of security and fault tolerance services. A major challenge to software development organizations is the complexity of creating and evolving distributed systems resulting from the tangling of middleware-specific functionality with core business functionality in system designs. We present an MDA-compliant middleware transparent software development approach in which application designs are developed independently of the middleware platform. Middleware features corresponding to dependability are encapsulated as aspects and woven with artifacts that realize core functionality. Our approach enables easy replacement of one dependability mechanism by another, and easy migration from one middleware platform to another. The approach also promotes reuse of aspects in multiple applications. This paper illustrates our approach with CORBA security services.

Keywords: *CORBA, MDA, dependability, security, authentication, authorization, SSL, aspect-oriented modeling, aspect-oriented programming, software reuse*

1. Introduction

The rapid growth of the Internet has resulted in widespread use of distributed applications that communicate with the help of middleware. Middleware platforms provide dependability features in the form of security and fault tolerance services. However, such features are often scattered across and tangled with modules providing core functionality. 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., naming and trading), these components may be crosscut by other middleware features (e.g., security, events and transactions). 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. Dependability features of the application that will be realized as middleware services 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 dependability implementations and 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 an application that incorporates CORBA security features.

We summarize related work in Section 2. In Section 3 we provide background information on CORBA security and the modeling notation used in this work. We present an overview of the MTSD approach in Section 4 and illustrate it with CORBA security examples in Section 5. We present our conclusions and outline directions for future work in Section 6.

2. Related Work

Aspect-oriented software development [10] has introduced aspect languages like AspectJ and Aspect C# which help in abstracting and encapsulating crosscutting concerns

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

at the programming level. Simmonds et al. [14] captured Jini middleware details in the form of code aspects. Pichler et al. [12] demonstrated the use of aspects with the Enterprise Java Beans container architecture. Zhang and Jacobsen [15] analyzed the use of aspects in middleware architectures and quantified crosscutting concerns in the implementations of middleware applications.

Bussard [3] 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. Hunleth [8] 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.

The MDA initiative [13] 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. [4] used the subject-oriented modeling approach 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.

France et al. [6] propose an aspect-oriented modeling (AOM) approach in which software designers specify primary models (base functionality), aspect models (non-orthogonal crosscutting functionality for dependability) and composition directives to obtain the integrated design. Cross-cutting design concerns are captured in aspect models using the Role-Based Metamodeling Language (RBML) [5]. 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 security and the RBML notation that is used to specify aspect models.

3.1. CORBA

The Common Object Request Broker Architecture (CORBA) [1] 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. Developers use the CORBA interface definition language (IDL) to describe the interfaces of distributed objects. 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. Object adapters assist the ORB with delivering client requests to the server object implementations. In our work, we use the Visibroker implementation of CORBA for the Java programming language.

VisiSecure provides a framework for incorporating security in Visibroker applications [2]. It uses the Java Authentication and Authorization System (JAAS), which defines extensions that allow pluggable authorization and user-based authentication.

Authentication is the process of verifying the identity of an entity. A client may need to be authenticated before being allowed access to resources. Server authentication may also be required. For example, to use the Secure Socket Layer (SSL) for transport layer security, the server needs to identify itself to the client. When servers make further invocations to other servers, they need to identify themselves before they can act on behalf of the original requester.

Authentication can be done by using passwords or public-key certificates. To authenticate clients, the server needs to expose the set of authentication realms it supports to the client. An authentication realm describes a set of users and their credentials (e.g., by using LDAP), and has user authentication mechanisms. Each realm corresponds to a JAAS *LoginModule* that actually performs the authentication. The client provides a username, password, and a realm under which it wishes to be authenticated. A server configuration file is used to specify the realm that is associated with a LoginModule. For client authentication, each process uses a configuration file containing the configuration for the set of authentication realms supported by the system. Developers can use the security API to provide authentication features. These features crosscut the client and server functionality. In this paper we show how the features can be cleanly encapsulated in the form of aspects.

Visisecure uses profiles which are repositories of security information. Profiles are of three types: (1) *Configuration files* that contain information on authentication realms and login modules, (2) *Rolemaps* that contain mappings of users and groups to roles, and (3) *Property files* that contain definitions of security properties.

Authorization is the process of making access control decisions on resources for an already authenticated entity based on certain privileges. An authorization domain allows users to act in given roles as specified in the rolemap.

The application developer defines the access control policies for access to resources in the application. The re-

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

sources are server objects and their methods. The policies are defined in terms of roles that provide a logical set of permissions to access a set of resources. The application developer needs to assign the roles that can access a particular resource. The assignment of roles, and the implementation of authorization functionality get coupled together with the implementation of business functionality and our goal is to decouple them.

3.2. Role Based Meta-Modeling Language

We treat an aspect model as a pattern that characterizes a family of design solutions for a crosscutting dependability 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 a static pattern specification (SPS) that characterizes conforming class diagrams, and a set of interaction pattern specifications (IPSs) that characterize conforming interaction diagrams. Details of the RBML notation can be seen in France et al. [5].

3.2.1 Static Pattern Specification

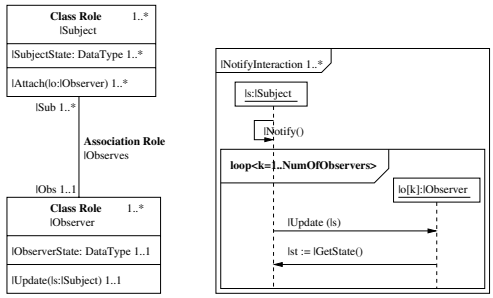
An SPS consists of classifier and relationship roles, where a classifier role is connected to other classifier roles by relationship roles. Properties in a classifier role are expressed in three forms:

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 [7]. 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.

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* 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 (1..*) on the *Sub* association-end role specifies that a conforming *Subject* can be part of one or more *Observes* association.



(a) Observer SPS (b) Observer IPS

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

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 calls to operations that conform to the feature role *Update*. The *Notify* behavior results in calls to operations that con-

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

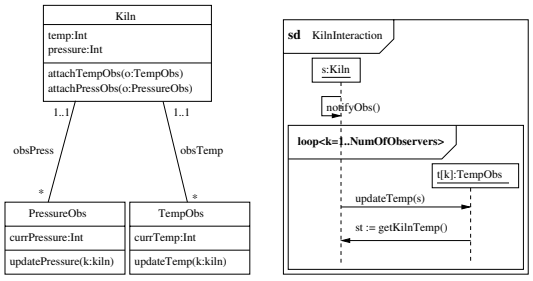
form to the *Update* role for each observer associated with the subject. Each observer then calls the operation that conforms to the *GetState* role in the subject. The behavioral roles *GetState* and *Notify* are not shown in the SPS. The loop structure shown in Figure 1(b) allows one to specify iterative 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

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.

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.



(a) Class diagram (b) Sequence diagram

Figure 2. Conforming UML Diagrams for the Observer Pattern.

4. Overview of the MTSD 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.

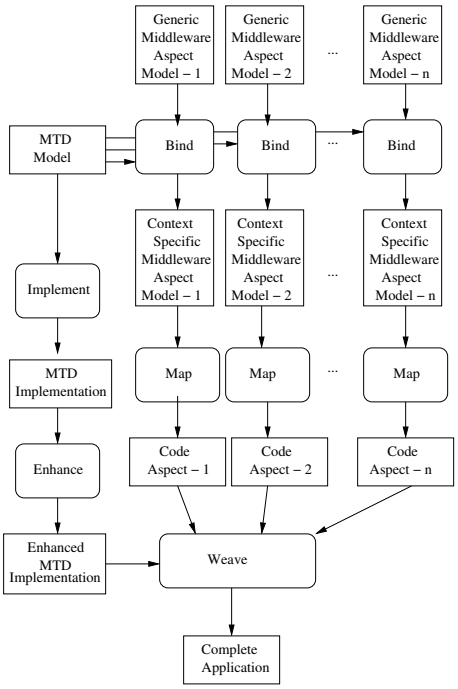


Figure 3. The MTSD Approach.

Middleware specific features are localized in aspect models. 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 generic aspect models for CORBA security features in Section 5.

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. This task may also require input from the application developer. For example, for the authorization aspect, we need to provide information in the form of application-specific role names and their access privileges.

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. Differ-

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

ent 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. 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. CORBA Security

We illustrate the MTSD approach by incorporating CORBA security features into a simple *Bank* application. The MTD in Figure 4 shows the design of the application. The MTD contains server-side classes *AccountServer* that implements the balance query operation and *AccountManagerServer* that implements the account creation operation. The client class uses these interfaces to create a new account and query the balance in the account. We do not show the MTD implementation for lack of space. The MTD model and implementation do not contain any CORBA specific feature.

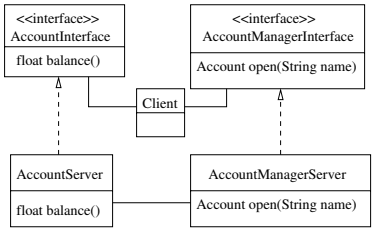


Figure 4. Bank Application MTD.

In addition to incorporating CORBA security features, we need to introduce CORBA connectivity to enable the client to talk to the server side. In Kamalakar and Ghosh [9], we present the MTSD approach for introducing CORBA connectivity using the JacORB implementation of CORBA. We have also implemented the aspect models and code mappings for the Visibroker implementation of CORBA connectivity, but we do not show the details in this paper. Instead we focus only on security features: authentication, authorization and the SSL.

5.1. Developing Aspect Models

We demonstrate the development of generic and context-specific aspect models for authentication and authorization.

5.1.1 Authentication

The invocation of the CORBA security API for authentication is encapsulated in the authentication security aspect. The generic aspect model for CORBA authentication using a *Context* object is shown in Figure 5 (SPS) and Figure 6 (IPS). The *ORB* contains the behavioral role `resolve_initial_references` which is used to obtain the interoperable object reference (IOR) of the Visibroker context that conforms to the *Context* classifier role. This role represents the security context under which the program executes. The classes conforming to the *ORB* and *Context* roles can be obtained by importing the packages *org.omg.CORBA.ORB* and the *com.Borland.security.Context* packages. The *Client* class role is played by the class which requests the services. The client class needs to be implemented by the application developer. We need similar roles on the server side for server authentication.

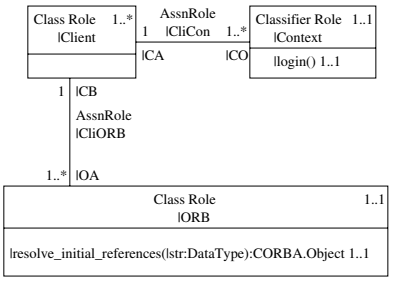


Figure 5. Authentication Aspect SPS.

The IPS describes client authentication. The client first obtains the IOR of the Visibroker security context by invoking `resolve_initial_references` on an instance of the *ORB*, and then casts it to the type `com.Borland.security.Context`. The client invokes the operation conforming to the `login` role on the security context IOR.

Context-specific aspect models are obtained from the SPS using the bindings shown in Table 1. The elements

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

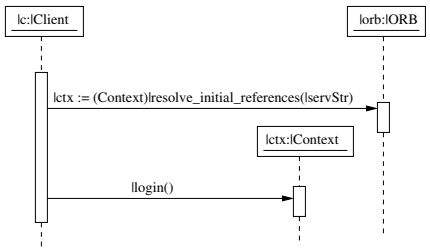


Figure 6. Authentication Aspect IPS.

Table 1. Authentication bindings.

Role	Class diagram element	How to obtain
ORB resolve_initial_references	ORB resolve_initial_references	stamp out stamp out
Context login	Context login	stamp out stamp out
Client/Server	Client/Server	bind

are added to the class diagram either by stamping them out directly from the SPS (if they are CORBA specific), or by explicitly binding with existing elements in the MTD class diagram (if they are application specific). The interaction diagrams are obtained from the IPSs by binding roles to the appropriate MTD elements. Figures 7 and 8 show the class and sequence diagrams respectively.

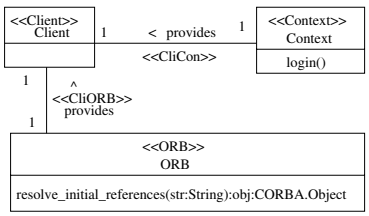


Figure 7. Authentication Class Diagram.

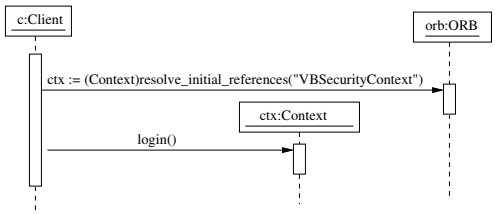


Figure 8. Authentication Interaction Diagram.

5.1.2 Authorization

Generic aspect models for authorization are shown in Figure 9 (SPS) and Figure 11 (server-side IPSs). There are no authorization-specific details that crosscut the client, and thus, there is no client-side authorization IPS. The *ObjectAccessPolicy* classifier role has two behavioral feature roles, `getRequiredRoles` and `getRunAsRole`. The former takes a method name as its argument and returns the sequence of roles that are allowed access to that method. The latter returns a run-as role for accessing the method of the object.

The *AccessPolicyManager* contains the `domain` and `getAccessPolicy` feature roles. The `domain` role returns the authorization domain associated with the particular rolemap and is used to set the access manager in the *ServerQOPConfig* object. The location of the rolemap file is set in the property file. The `getAccessPolicy` operation takes an instance of the servant, the object identity and the adapter identity and returns an instance of *ObjectAccessPolicy*.

The *ServerQOPConfigDefaultFactory* class role contains the `create` behavioral role which creates a *ServerQOPConfig* object. The *ServerQOPConfigHelper* class role has the behavioral feature role `insert` which is used to insert values into objects of type *Any*.

The *ORB* class role has three feature roles: (1) `resolve_initial_references` to obtain the IORs of a predefined service, (2) `create_any` to create an object of CORBA type `Any`, and (3) `create_policy` to create a `Policy` object.

The *PolicyManagerHelper* contains the behavioral role `narrow` that narrows a reference of type `CORBA.Object` to *PolicyManager*. The *PolicyManager* role is for handling `ORB` level policies. It contains the `set_policy_overrides` behavioral role which overrides the current set of policies with a list of `Policy` overrides.

The *com.Borland.security.csiv2* package contains (1) interfaces that conform to the *ObjectAccessPolicy* and *AccessPolicyManager* roles, and (2) classes that conform to the *ServerQOPConfig*, *ServerQOPConfigHelper*, *ServerQOPPolicy* and *ServerQOPConfigDefaultFactory* roles. The classes that conform to the *ORB*, *Policy* and *PolicyManagerHelper* roles and the interface that conforms to the *PolicyManager* role are available in the *org.omg.CORBA* package. The classes conforming to the *InterfaceTypeImpl* and *Server* roles are designed by the application developer.

The server-side authorization behavior is shown in Figure 11. Figure 11(a) shows the behavior of the *Server* role. The server creates an instance of the configuration factory (*ServerQOPConfigDefaultFactory*) and an instance of *AccessPolicyManager* that contains implementations of the methods `getAccessPolicy` and `domain`. Using the configuration factory, the server creates a configuration instance (*ServerQOPConfig*). The server creates an instance

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

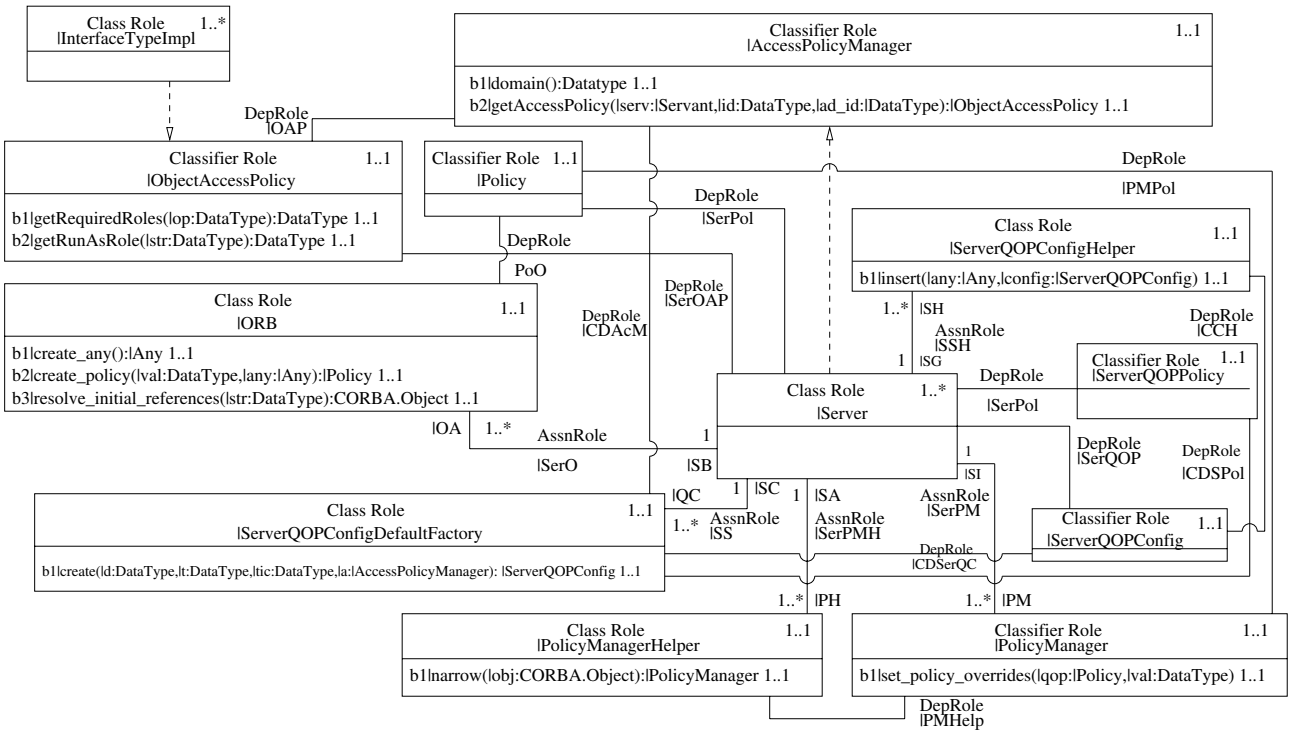


Figure 9. Authorization Aspect SPS.

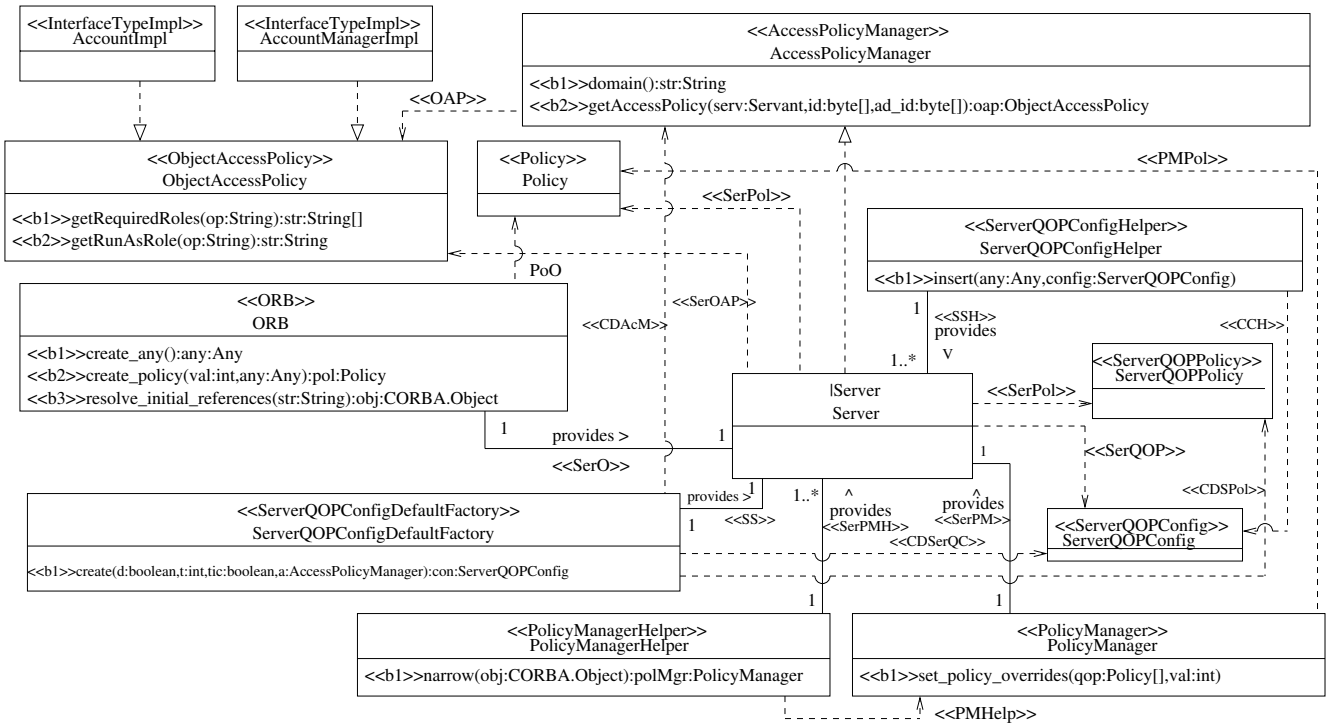


Figure 10. Authorization Class Diagram.

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

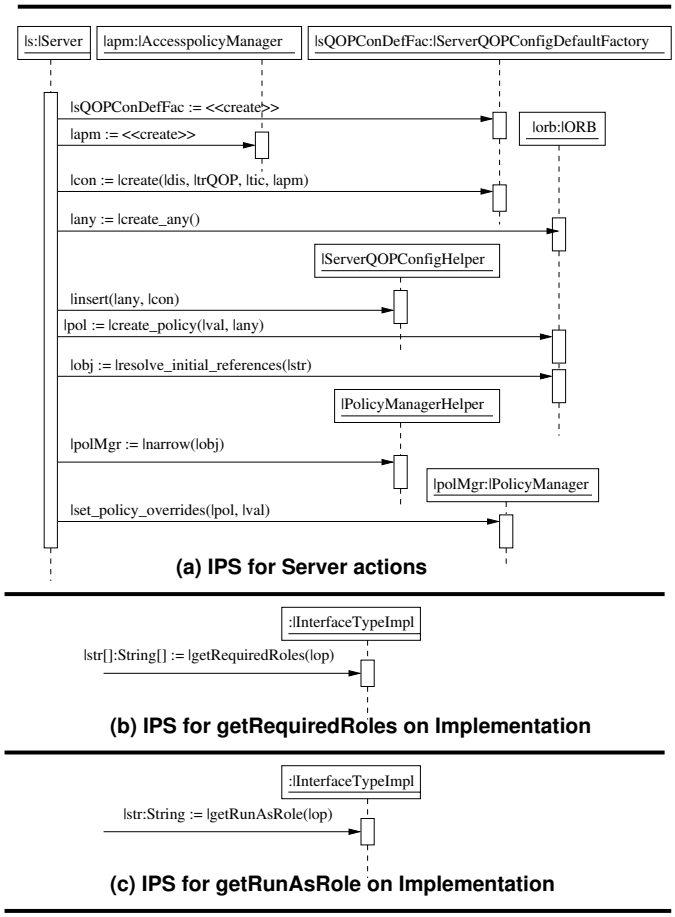


Figure 11. Authorization Aspect IPSs.

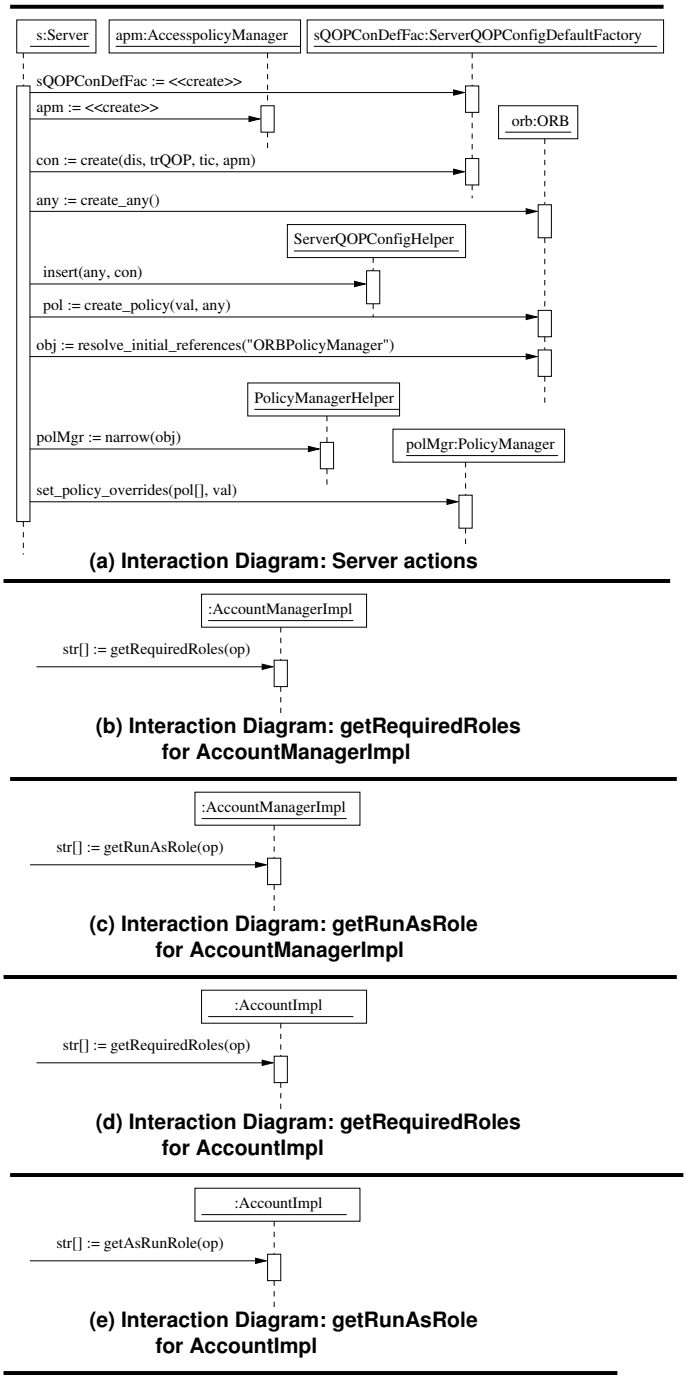


Figure 12. Authorization Interaction Diagrams.

of type `Any` into which it inserts the configuration instance using the *ServerQOPConfigHelper*. A policy object is created with the *Any* instance using the *ORB*. The IOR of the *PolicyManager* is obtained from the ORB instance and policy overrides are set with the newly created *Policy* instance.

Figures 11(b) and (c) show the authorization behavior in the role played by the implementation class. This class needs to implement the `getRequiredRoles` and `getRunAsRole` methods in the *ObjectAccessPolicy* interface. The `getRequiredRoles` behavior takes in an operation name and returns the application roles that are allowed access to the operation. The `getRunAsRole` behavior takes in an operation name and returns the role that can invoke the operation. OCL constraint templates (not shown) specify the return values.

Bindings shown in Table 2 were used to obtain the application specific class diagram shown in Figure 10. The context-specific interaction diagrams shown in Figure 12 are obtained by binding appropriate roles to the MTD elements. The message parameters are specified by the developer. Figure 12(a) shows the interaction diagram for the behavior for the *Server* instance. Figures 12(b) and (c) show the interactions corresponding to the `getRequiredRoles` and `getRunAsRole` behavior for the *Account-*

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

ManagerImpl class. Figures 12(d) and (e) show similar behavior for the *AccountImpl* class. OCL constraints specific to the application are generated from the templates. The `getRequiredRoles` and `getRunAsRole` operations rely on access control lists to obtain the developer specified mappings of operations and corresponding roles.

Table 2. Authorization bindings.

Role	Class diagram element	How to obtain
ORB	ORB	stamp out
create_any	create_any	stamp out
create_policy	create_policy	stamp out
resolve_initial_references	resolve_initial_references	stamp out
ObjectAccessPolicy	ObjectAccessPolicy	stamp out
getRequiredRoles	getRequiredRoles	stamp out
getRunAsRole	getRunAsRole	stamp out
AccessPolicyManager	AccessPolicyManager	stamp out
domain	domain	stamp out
getAccessPolicy	getAccessPolicy	stamp out
Policy	Policy	stamp out
ServerQOPConfigDefaultFactory	ServerQOPConfigDefaultFactory	stamp out
create	create	stamp out
ServerQOPConfigHelper	ServerQOPConfigHelper	stamp out
insert	insert	stamp out
ServerQOPConfig	ServerQOPConfig	stamp out
ServerQOPPolicy	ServerQOPPolicy	stamp out
PolicyManagerHelper	PolicyManagerHelper	stamp out
narrow	narrow	stamp out
PolicyManager	PolicyManager	stamp out
set_policy_overrides	set_policy_overrides	stamp out
InterfaceTypeImpl	AccountManagerImpl	bind
InterfaceTypeImpl	AccountImpl	bind
Server	Server	bind

5.2. Mapping Aspect Models to AspectJ

In Kamalakar and Ghosh [9] we described how the CORBA connectivity aspect models for clients and servers are mapped to code aspects in AspectJ. In this paper, we use connectivity aspects along with aspects for authentication or authorization. Thus, multiple aspects are woven with the same MTD implementation. However, we do not show the details of the connectivity aspect mappings for the bank application.

5.2.1 Authentication

The `login` method in the security context interface is used for authenticating clients to allow access to servers and vice versa. The *ORB* instance is created in the connectivity aspect using AspectJ introduction and "after" advice; the pointcut is the execution of the single argument constructor which takes the command line argument for *ORB* initialization as its parameter. An "after" advice is used in the authentication aspect to create a *Context* object from the *ORB* instance. The pointcut is the execution of the constructor. The *ORB* instance is accessed in the authentication aspect using a reference to the client object. This is made possible by declaring the authentication aspect as *privileged*. The `login` method is invoked on the *Context* object. After the login is performed, the rest of the connectivity aspect is executed in an "after" advice; the pointcut is the call reception of the single argument constructor.

Figure 13 shows the AspectJ code obtained by using the following steps to map the authentication aspect model to AspectJ code.

1. Import the appropriate packages (see aspect code).
2. Declare the aspect as privileged.
3. Declare the aspect precedence of the authentication aspect and the connectivity aspect to specify the order of execution of the advices since there are two after advices on the same pointcut.
4. Define a pointcut on the execution of the constructor and define the associated "after" advice which creates the security context object and invokes the `login` method on it.

```
import org.omg.CORBA.ORB;
import com.borland.security.Context;
import Bank.*;
public privileged aspect AuthenticationAspect {
   Context context;
   declare precedence: AuthenticationAspect, ClientConnectivityAspect
   pointcut getContext(Client c): execution(public Client.new(..)) && target(c);
   after(Client c) : getContext(c) {
      try {
          context = (com.borland.security.Context)
                  c.orb.resolve_initial_references("VBSecurityContext");
          if (context!=null) {
            context.login();
          }
       } catch (Exception e) {
          e.printStackTrace();
          System.exit(1);
       }
   }
}
```

Figure 13. AspectJ Code for Authentication.

5.2.2 Authorization

Since authorization requires authentication, we have the option of using an authentication aspect along with the authorization aspect. However, to keep the illustration simple, we use a password-based authentication mechanism by setting the `login` variable to `true` in the property file. Hence, we do not require the use of an authentication aspect for this illustration.

On the server-side, there are two sets of aspects. One is for the server class; the other is for the implementation class. For the server class, the authorization details are encapsulated in the *ServerSecurityAspect* (see Figure 14). The *ORB* instance required for authorization is created in an "after" advice in the connectivity aspect. The pointcut is the execution of the single argument constructor. The authorization steps for setting up an access policy manager and a policy manager for policy overrides are carried out in an "after" advice in the server authorization aspect. The pointcut is the execution of the constructor. The rest of the steps in the server connectivity aspect are executed in an "after" advice. The pointcut is the call reception of the single argument constructor.

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

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

1. Import the appropriate packages (see aspect code).
2. Declare the aspect as privileged.
3. Declare aspect precedence to specify the order of execution of the advices on the same pointcut.
4. Define a pointcut on the execution of the constructor and define the associated "after" advice which takes care of the authorization details.
5. The domain name for the `domain` method implementation is obtained from the `vbroker.security.authDomains` variable in the property file.

```
import java.io.*;
import org.omg.CORBA.Any;
import org.omg.CORBA.Policy;
import org.omg.CORBA.PolicyManager;
import org.omg.CORBA.PolicyManagerHelper;
import org.omg.CORBA.SetOverrideType;
import com.borland.security.csiv2.SERVER_QOP_CONFIG_TYPE;
import com.borland.security.csiv2.ServerQoPConfigDefaultFactory;
import com.borland.security.csiv2.ServerQoPConfig;
import com.borland.security.csiv2.ServerQoPConfigHelper;
import com.borland.security.csiv2.ServerQoPPolicy;
import com.borland.security.csiv2.AccessPolicyManager;
import com.borland.security.csiv2.ObjectAccessPolicy;
import org.omg.PortableServer.*;

public privileged aspect ServerSecurityAspect {
  declare precedence : ServerSecurityAspect, ServerConnectivityAspect;
  pointcut setPolicy(Server s):execution(public Server.new(..)) && this(s);
  after(Server s) : setPolicy(s) {
    try {
      ServerQoPConfig config =
         new ServerQoPConfigDefaultFactory().create(false,
               ServerQoPPolicy.ALL, true, new AccessPolicyManager() {
           public String domain () {
             return "bank";
           }
           public ObjectAccessPolicy getAccessPolicy (Servant servant,
                                      byte[] id,  byte[] adapter_id) {
             return (ObjectAccessPolicy) servant;
           }
         });
      Any any = s.orb.create_any();
      ServerQoPConfigHelper.insert(any, config);
      Policy qop = s.orb.create_policy(SERVER_QOP_CONFIG_TYPE.value, any);
      PolicyManager polmgr =
        PolicyManagerHelper.narrow(s.orb.resolve_initial_references
           ("ORBPolicyManager"));
        polmgr.set_policy_overrides(new Policy[] {qop},
            SetOverrideType.SET_OVERRIDE);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 14. AspectJ Code for Authorization (Server).

The access policies for the resources of the implementation objects are specified in the *AccountAuthorizationAspect* (see Figure 15). Only users playing the manager role can open accounts. Customers and tellers can query the balance.

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

1. Import the packages shown in the code aspect.
2. The implementation classes (*AccountManagerImpl* and *AccountImpl*) need to implement the *ObjectAccessPolicy* interface. Therefore, we include the declare parents clause.
3. The methods are implemented using the AspectJ introduction construct.

```
import com.borland.security.csiv2.ObjectAccessPolicy;
import org.omg.CORBA.BAD_OPERATION;
import org.omg.PortableServer.*;

public aspect AccountAuthorizationAspect {
  declare parents : AccountImpl implements ObjectAccessPolicy;
  declare parents : AccountManagerImpl implements ObjectAccessPolicy;

  public String[] AccountManagerImpl.getRequiredRoles(String op) {
    if (op.equals("open")) {
      return new String[] {"Manager"};
    }
    throw new BAD_OPERATION("No operation named " + op);
  }

  public String AccountManagerImpl.getRunAsRole(String op) {
    return null;
  }

  public String[] AccountImpl.getRequiredRoles(String op) {
    if (op.equals("balance")) {
      return new String[] {"Customer", "Teller"};
    }
    throw new BAD_OPERATION("No operation named " + op);
  }

  public String AccountImpl.getRunAsRole(String op) {
    return null;
  }
}
```

Figure 15. AspectJ Code for Authorization (AccountManagerImpl and AccountImpl).

5.3. Enhancing MTD Implementation and Weaving Aspects

Prior to weaving the MTD implementation with the code aspects, we need to enhance the MTD implementation to ensure CORBA compliance. CORBA requires the declaration of the service interfaces in the IDL format. Hence, the Java interface written by the developer needs to be converted to the CORBA IDL format. The developer has to specify 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 using the IDL compiler, the required CORBA files (AccountManager, AccountManagerPOA, AccountManagerHelper, Account, AccountPOA, and AccountHelper) are created.

We rename the service implementation classes *AccountManagerServer* and *AccountServer* provided by the application developer to *AccountManagerImpl* and *AccountImpl* respectively. The clause "`implements AccountManagerInterface`" in the *AccountManagerImpl* and "`implements AccountInterface`" in the *AccountImpl* must be deleted because they must now extend *AccountManagerPOA* and *AccountPOA* respectively. This extension is done in the connectivity aspects.

In the main method of the *Server* class, we need to create a server object using `new` because we have used the ex-

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

ecution of the single argument constructor as the pointcut for the "advice" specified in the aspects. For similar reasons, the client object is created in the *Client* class using the client constructor. Since we need to pass command line arguments to the connectivity aspect, we use a one argument constructor (with `args` as the parameter) in both the client and the server.

The enhanced MTD implementation is shown in Figure 16. Finally, the complete application is generated by weaving the client and server code with the respective aspects.

The authorization details in the *Server*, i.e., creating new QoP policies and setting the policy overrides using the *PolicyManager* are encapsulated in the authorization security aspect corresponding to the server. The roles that are allowed to access the resources are encapsulated in the authorization aspect corresponding to the implementation class.

```
// AccountImpl.java
public class AccountImpl {

  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
//_____________________________________________________________

// AccountManagerImpl.java
import java.util.*;
public class AccountManagerImpl {
  public synchronized Bank.Account open(String name) {
    Bank.Account account = (Bank.Account) _accounts.get(name);
    if(account == null) {
      account = createNewAccount(name);
      _accounts.put(name, account);
    }
    return account;
  }

  private Bank.Account createNewAccount(String name) {
    return null;
  }
  private Dictionary _accounts = new Hashtable();
}
//_____________________________________________________________

// Client.java
public class Client {
  public Client(String[] args) {  }

  public static void main(String[] args) {
    Client c = new Client(args);
    String managerId = "BankManager";
    Bank.AccountManager manager = c.getServiceObject(managerId);
    String name = "J. Doe";
    Bank.Account account = manager.open(name);
    float balance = account.balance();
    System.out.println("The balance in " + name +
      "'s account is $" + balance);
  }
  public Bank.AccountManager getServiceObject(String managerId) {
    return null;
  }
}
//_____________________________________________________________

// Server.java
public class Server {
  public Server(String[] args) {}

  public static void main(String[] args) {
    try {
      Server s = new Server(args);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 16. Enhanced MTD Implementation

5.4. SSL

We have applied the MTSD approach for incorporating SSL-based communication in our applications. We are unable to present the models and code owing to lack of space. Instead we present a high level description of how SSL can be aspectualized.

SSL [11] provides capabilities such as authentication of the entity at the other end of a TCP connection with the help of user certificates and allows for the encryption of messages sent over a TCP connection. Clients and servers identify themselves to the SSL layer using X.509 certificates chains. During communication, the client and server exchange certificates. For server authentication, the client verifies if the server's certificate is valid and checks if the certificate has been issued by a certificate authority (CA) listed in the client's list of trusted CAs. If the client obtains a certificate chain from the server, the certificate chain is searched for the presence of a CA certificate that is listed in the client's list of trusted CAs. Using the same techniques, the server authenticates the client. A cipher suite indicates the type of encryption to be performed on the messages that need to be sent between the entities involved in the communication. The client and server use an SSL *Current* object to obtain the certificate chains from each other. This object is used to determine the cipher suite that was negotiated between the client and the server.

In Visibroker, the Java Secure Socket Extension (JSSE) is used to provide mechanisms for supporting SSL. JSSE is a Java implementation of SSL and TLS protocols, and includes the functionality of data encryption, server and client authentication, and message integrity for enabling secure internet communications. To incorporate SSL into an application, we use connectivity aspects along with the SSL aspects. We use three SSL aspects for the *Client*, *Server* and *AccountManager*. The *Client-SSL* aspect takes care of two tasks:

1. Identify the client to (and thus, authenticate with) the SSL layer using X.509 certificates and an encrypted private key. This is done by invoking the `login` method on the security context object. A *wallet* object is constructed using the certificates and the private key, and passed as parameter to the *login* method for authentication.
2. Using an SSL current object, obtain the certificate chain of the server to verify if the server is trusted and also obtain the negotiated cipher.

On the server side, the *Server-SSL* aspect identifies the server to the SSL layer using the same steps as the *Client-SSL* aspect. The *AccountManager-SSL* aspect verifies the identity of the client by obtaining its certificate chain and checking for the trusted certificate. This is done by the

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

AccountManager-SSL aspect and not the *Server-SSL* aspect because each time a service method in the implementation class is invoked by the client, the implementation class needs to verify that the client is trusted. We have not considered the checking of the Certificate Revocation List to see if the peer certificate has been revoked before the scheduled expiration date. This will be addressed in future work.

6. Conclusions and Future Work

This paper presented the MTSD approach and applied it to develop secure CORBA applications. We incorporated CORBA security features to provide authentication, authorization, and SSL into an application that was designed free of any CORBA-specific design elements. The design aspects representing the security features 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. Our approach allows us to weave in multiple dependability aspects.

The MTSD approach can be used to develop fault tolerant applications as well. We are currently working on developing aspect models for fault tolerance mechanisms using object replication. The fault-tolerance aspects are responsible for maintaining replica consistency, replication transparency and failure transparency. There are two types of fault-tolerance mechanisms: those that are provided by the middleware infrastructure, and those that are provided by the application. In both cases, our approach can provide the advantages resulting from the decoupled design of business functionality and fault tolerance concerns. In infrastructure-controlled fault tolerance, the aspect implementations will utilize the API and services provided by the middleware platform vendor. In application-controlled fault tolerance, the aspects are implemented by the application developer.

Our next task is to apply MTSD to different application architectures and investigate the effect of using multiple potentially conflicting aspects with one primary model. 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.

There are several tasks in the MTSD approach that require inputs or enhancements from the application developer. We are working on specifying guidelines to perform the tasks systematically. We expect that different middleware platforms will require different sets of guidelines. We are also working on the development of a prototype tool for the automatable tasks. We plan on investigating verification and validation techniques to ensure that the woven code actually preserves the specified properties of the aspects. We will perform a cost-benefit analysis of using the approach and compare it with other approaches.

References

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

[2] Borland Software Corporation. *Security Guide, Enterprise Server 6*. Borland, 2004.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] B. Kamalakar and S. Ghosh. A Middleware Transparent Approach for Developing CORBA-based Distributed Applications. Technical Report 04-104: Available at: http://www.cs.colostate.edu/homes/ghosh/papers/csutechreport04104.pdf, Department of Computer Science, Colorado State University, Fort Collins, Colorado.

[10] 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.

[11] D. Pedrick, J. Weedon, J. Goldberg, and E. Bleifield. *Programming with Visibroker: A Developer's Guide to Visibroker for Java*. John Wiley and Sons, USA, 1998.

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

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

[14] 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.

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

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE