Component Models and Aspect-Oriented Programming

Mira Mezini[1], Klaus Ostermann[1,2], and Roman Pichler[2]

[1] Darmstadt University of Technology, D-64283 Darmstadt, Germany
`mezini@informatik.tu-darmstadt.de`
[2] Siemens AG, Corporate Technology, D-81730 Munich, Germany
{`Klaus.Ostermann,Roman.Pichler`}`@mchp.siemens.de`

Abstract. Server-side component models such as Enterprise JavaBean (EJB) add powerful abstractions to the bare "business objects" layer, in order to support a clean separation of server-side application logic from other concerns such as distribution, security, transaction management and persistence. An improved separation of concerns is also the main goal of aspect-oriented programming (AOP). This paper compares the two approaches and reasons about the possibility of substituting (parts of) component models using aspect-oriented programming mechanisms. We conclude that AOP is a promising approach to eliminate important shortcomings of the container-based component approach. However, our analysis of different concrete aspect-oriented languages shows that current AOP technology is not yet mature enough to supersede component models.

1 Introduction

This paper investigates infrastructural services such as security, transactions and persistence and their application to base objects. To make things more concrete, we concentrate on the authorization service of the Enterprise JavaBean (EJB) component model [4]. Our observations also apply to other services and other component models, such as the Corba Component Model (CCM) [2] or COM+ [**?**]. Sec. 2 emphasizes the merits and flaws of the container approach. Our working assumption is that aspect-oriented languages can at least partially take on the responsibilities fulfilled by a component's interceptor framework, i.e. the container. Sec. 3 investigates if this assumption is valid.

2 Enterprise JavaBean Container

2.1 Introduction

The Enterprise JavaBean (EJB) component model is Sun's server side component model. An EJB has to comply with certain idioms and programming restrictions and is hosted by a container. The container acts as the bean's runtime environment and provides the following infrastructural services to an EJB:

Resource and life cycle management, concurrency, security, persistence, transactions and remoting [4]. A deployment descriptor allows associating the infrastructural services with the beans hosted by the container in a declarative way at deployment time. By separating the business logic provided by an EJB form services like security or transactions, the EJB component model tries to encapsulate the infrastructural services (which are primary examples crosscutting concerns in AOP terminology) within the container.

2.2 Sample Application

Let's have a closer look how the EJB container encapsulates crosscutting concerns. Fig. 1 illustrates the association between an EJB and its container [1].

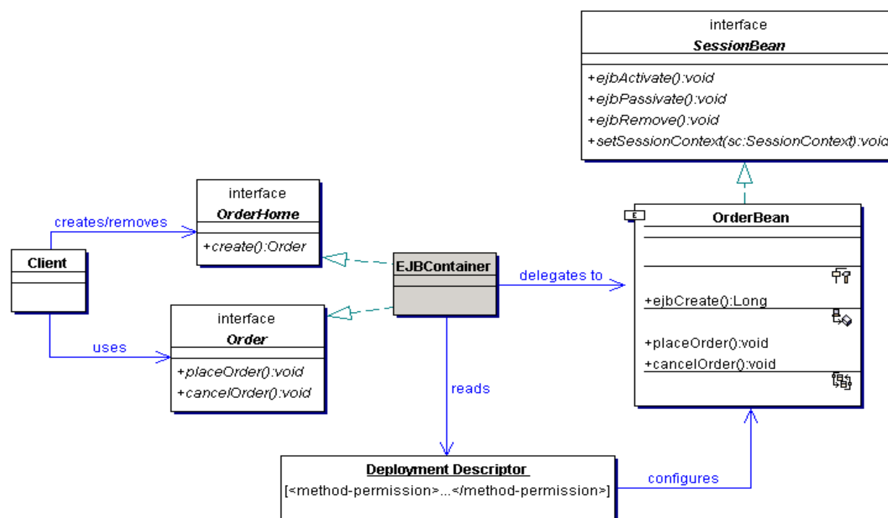

Fig. 1. EJB Sample Application

In the diagram above, the EJB `OrderBean` exposes two interfaces, `OrderHome` and `Order`. The interfaces allow the `Client` class to create an EJB instance and to call business methods. `OrderHome`, `Order` and `OrderBean` have to be created by the application developer. The class `EJBContainer` represents the EJB container. `EJBContainer` realizes the interfaces `OrderHome` and `Order`. This allows the container to intercept calls to the business methods `placeOrder()` and `cancelOrder()` of `OrderBean` and to execute authorization checks, for instance, before the method calls are passed on to `OrderBean`.

Infrastructural services provided by the EJB container are associated with `OrderBean` by editing the bean's deployment descriptor. A deployment descrip-

[1] For a more detailed account of the EJB container responsibilities, see [4]

tor is an XML file that makes it possible to configure EJBs declaratively. The EJB container reads the contents of the deployment descriptor at deployment time and applies the appropriate infrastructural services whenever the container intercepts a request to an EJB. For instance, we can specify authorization constraints for `OrderBean` by setting the appropriate values in the relevant section of the bean's deployment descriptor, cf.

```
<method-permission>
<role-name>admin</role-name>
<method>
  <ejb-name>Order</ejb-name>
  <method-name>*</method-name>
</method>
</method-permission>

<method-permission>
<role-name>customer</role-name>
<method>
  <ejb-name>Order</ejb-name>
  <method-name>placeOrder</method-name>
</method>
</method-permission>
```

Fig. 2. An EJB deployment descriptor fragment

The deployment descriptor fragment in Fig. 2 states that only the roles `admin` and `customer` are allowed to execute methods on `OrderBean`. `admin` is allowed to execute any method, `customer` is only allowed to access `placeOrder`. Whenever the container intercepts a call to one of the business methods of `OrderBean`, it checks the access rights specified in the deployment descriptor. If the client is authorized to access the method requested, the container executes the corresponding method on `OrderBean`. Otherwise, it aborts and throws an exception back to the client. Notice that the EJB component model only supports a role-based authorization model.

2.3 Crosscutting Concerns

The EJB container intercepts method calls to an EJB hosted by the container. It implements an interceptor framework and is a prime example of the application of the interceptor design pattern [3]. Notice that the interceptor framework is not standardized by the EJB specification [4]. The container vendor is responsible for the number of and the dependencies between the container classes.

The interceptor pattern allows to encapsulate services in dedicated classes and to add infrastructural services to a container without affecting existing code. The basic idea is that all direct calls to business objects are replaced by calls

to corresponding proxy objects that are provided (generated) by the container for each business object. These proxy objects inform the container about every call to the business object. The container has a list of *interceptors* that define additional actions that have to be performed on every call to a business object, e.g., a security interceptor performs an authorization check (based on the information in the bean's deployment descriptor) before the method call is actually dispatched to the business object. In order to ensure that this mechanism works, the bean developer has to obey a number of idioms. For example, object creation has to be delegated to the container (which returns a corresponding proxy object), and a bean must not attempt to pass `this` as an argument or method result but retrieve its corresponding proxy from the container first. Fig. 3 shows a simplified class diagram of the EJB interceptor framework used by the open source application server jBoss [1].

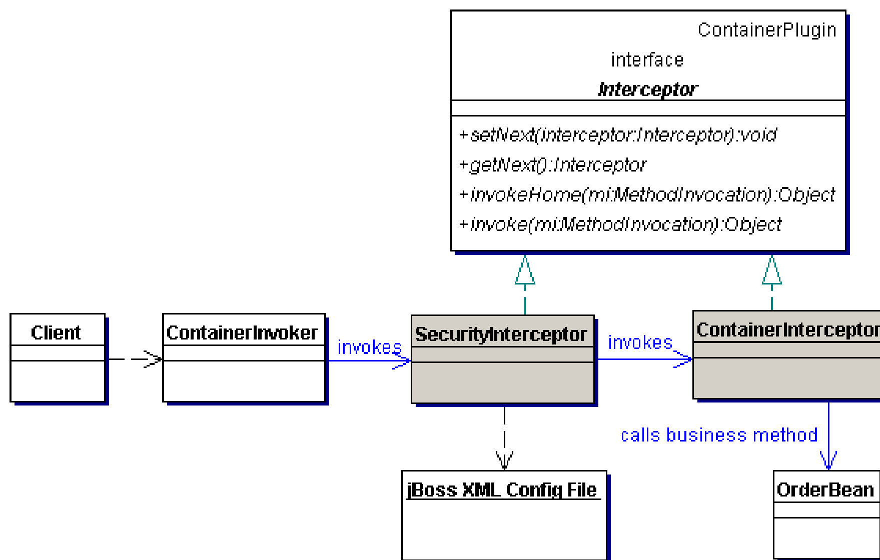

Fig. 3. Simplified Interceptor Framework of the Application Server jBoss

In Fig. 3, a client issues a request to `OrderBean`. Before the method call is dispatched to the EJB, the interceptors intercept the call and apply infrastructural services. For instance, `SecurityInterceptor` executes authorization checks (depending on the bean's deployment descriptor). jBoss requires that interceptors must implement the interface `Interceptor`. This allows extending the container by adding custom interceptors. Notice, however, that this solution is not standardized by the EJB specification.

Using the interceptor pattern enables the EJB container to transparently add infrastructural services such as authorization checks to beans such as `OrderBean`. Since infrastructural services are associated with an EJB via a deployment descriptor, the implementation of `OrderBean` does not have to be aware of any authorization checks or contain any authorization logic. This leads to a separation of business logic from infrastructural concerns such as authorization. The EJB container thus fulfills a similar role as an aspect in AOP [?,?].

Even though an EJB containers like the jBoss application server use interceptors, the EJB specification treats the container as a black box. [4] does not allow to configure interceptors or to add custom interceptors and extend the container functionality.[2] As a consequence, EJB technology is fairly easy to use but it also has some important limitations.

2.4 Advantages

The Enterprise JavaBean component model provides the following advantages: Every container compliant to the EJB specification offers the same set of infrastructural services. A standard set of services allows easy reuse and the integration of off-the-shelf components. An EJB can thus be easily deployed in various containers produced by different vendors - supposing the bean uses only standard interfaces.

Infrastructural services can be associated with EJBs via a declarative mechanism at deployment time. A dedicated entity, the deployment descriptor, associates the services with a bean. No source code is required - neither container nor EJB source code. In fact, EJBs are deployed as byte code. Since the deployment descriptor is an XML file, a deployer deploying an EJB and associating services with the bean does not have to be able to understand any Java.

2.5 Problems

The EJB component model's approach to modularize crosscutting concerns using a container has the following drawbacks:

Lack of Tailorability An EJB developer can either use the services provided by the EJB container or decide not to use them at all. There is no way to configure the container or any of its infrastructural services. It is also impossible to add new services to the container. We call this *lack of tailorability.* Suppose the sample application introduced in 2.2 is developed for a large corporation. It is likely that the company already uses an authorization service, which we would like to reuse. Employing EJB technology, we cannot tell the container to access an existing authorization service. We are completely dependent on the container

[2] Notice however that [6] does provide a standardized way to add new interceptors to the Servlet Engine, which acts as a web container, using the `javax.servlet.Filter` interface.

vendor's authorization solution. [3] The EJB component model assumes that the way the container encapsulates crosscutting concerns is completely transparent to a developer. If a service provided by the container does not fit to our needs, we either do not use EJB technology at all or we employ programmatic authorization within our EJBs.

Not being able to configure the container and its services leads to another drawback: Even if our application in 2.2 requires only security and transaction-related services, we get all the other container services as well. Again, there is no way to leave out crosscutting concerns that are not applicable to the application under consideration. As a consequence, container vendors offer and sell one package - the complete container. This makes the EJB component model fairly heavyweight and expensive.

Lack of checking and enforcement An EJB developer has to obey many design rules and idioms that are not enforced and cannot be checked by the compiler. For example, the EJB specification defines 17 programming restrictions [4, p. 494] for bean developers, such as "an EJB must not attempt to manage threads" or "an EJB must not use read/write static fields". In order to create an instance of a class or call methods of another EJB, the standard language mechanisms, namely constructor calls and message passing, are evaded. Instead, all actions have to be dispatched via the container. For example, a simple constructor call

```
MyBean bean = new MyBeanImpl();
```

has to be replaced by

```
Context initial = new InitialContext();
Object objref = initial.lookup("java:comp/env/ejb/MyBean");
MyBeanHome home = (MyBeanHome) PortableRemoteObject.narrow(objref,
                                            MyBeanHome.class);
MyBean bean = home.create();
```

In case a bean calls a method `foo()` of another bean `otherBean` and wants to pass `this` as an parameter to the other bean,

```
otherBean.foo(this);
```

, the `this` parameter has to be replaced by the corresponding proxy, which can be retrieved similar to

```
otherBean.foo( (MyBean) SessionContext.getEJBObject());
```

[3] Notice that Java does offer a straightforward way to reuse an authorization service via the Java Authentication and Authorization Service API (JAAS) [5]).

The decisive point is that these rules cannot be enforced at compile time. Even if the developer remembers to use the container idioms everywhere, the compiler cannot verify that these calls to the container are at least type safe, (cf. the type casts in the examples) because due to the use of universal methods (`getEJBObject()`) and strings instead of types, the type system is effectively discarded and all well-known benefits of static typing are lost.

Inconsistency Even though the EJB component model aims to free the application developer from having to worry about crosscutting concerns, it does allow developers to explicitly address authorization in their code, cf.

```
public class OrderBean implements SessionBean {

SAPConnector sap = new SAPConnector();

public void placeOrder() throws EJBException, SecurityException {
  if (EJBContext.getCallerPrincipal.getName.equals("Mr Jones")) {
      sap.placeOrder();
    }
    else {
      throw new SecurityException("Access denied. Unauthorized user");
    }
}
}
```

In the code fragment above, the Enterprise JavaBean `OrderBean` provides the method placeOrder(). Before an order is actually palced, a user-based authorization check is executed. OrderBean.placeOrder() uses the EJBContext interface provided by the EJB container to obtain the client's principal. Only if the client request is associated with the user Mr Jones, the order is placed and the appropriate method on SAPConnector is called.

In the code fragment, business and security logic is tangled, and the two distinct concerns are mixed. As a consequence, an application developer has to take care of authorization checks while writing business logic. What's particularly problematic is that security checks are spread over several places. Several methods on the EJB OrderBean implement the authorization checks. If there were a second Enterprise JavaBean that accessed SAPConnector, we would have to duplicate the authorization code in this bean (supposing we use programmatic security checks). This would lead to the distribution of authorization logic over several places. As a consequence, the security policy implementation of our application would be hard to understand and maintain.

Encapsulating infrastructural services in the EJB container and at the same time allowing developers to implement their own authorization mechanism seems to contradict the attempt to delegate all crosscutting concerns to the container. It shows that the EJB component model does not succeed in encapsulating crosscutting concerns in the container adequately. Making the container more flexible and configurable could be a potential solution. Allowing developers to mix authorization and business logic is a very bad idea.

Insufficiency The nice thing about an EJB container is that it encapsulates crosscutting concerns. That's great as long as we use code entities that can be hosted by the EJB container, i.e. Session and Entity beans. A real world e-business application will have to use ordinary Java classes in addition to EJBs, though. There are a number of reasons why container-hosted entities are not sufficient for a large J2EE application. One of the reasons is that EJBs are fairly constrained. They are not allowed to create threads or access the file system amongst other things (Sun 2001b). Another reason is that some of the application logic that resides on the application server may well be needed on the web server, too. In order to facilitate easy reuse, we could encapsulate this part of the application logic in normal Java classes and have EJBs delegate to these classes. A good example is database access from the web server for reads (fast and efficient) and database access from the application server for writes (using the transaction service provided by the EJB container). Employing ordinary Java classes and applying the Data Access Object pattern (Sun 2001e) allows us to do this. If we use ordinary Java classes in our application that live outside the context of an EJB we cannot take advantage of any container services. We thus face the problem of how to modularize crosscutting concerns and how to access infrastructural services. For instance, if we use ordinary Java classes (outside the context of an EJB), we are responsible to make sure that a client does have access rights to the appropriate methods. We may well use an authorization service to enforce a security policy. This way we end up with an authorization service provided by the container and an additional authorization service accessed by ordinary Java classes. As a consequence, we run into higher maintenance costs due to the overhead of two authorization services. In addition we face the problem of how to separate authorization as a crosscutting concern from our application code within the ordinary Java classes. Similarly, if we use ordinary Java classes that live within the context of an EJB, we generally cannot apply container services to them. There is no way, for instance, to use the container authorization mechanism to control access to a helper class from an Enterprise JavaBean.

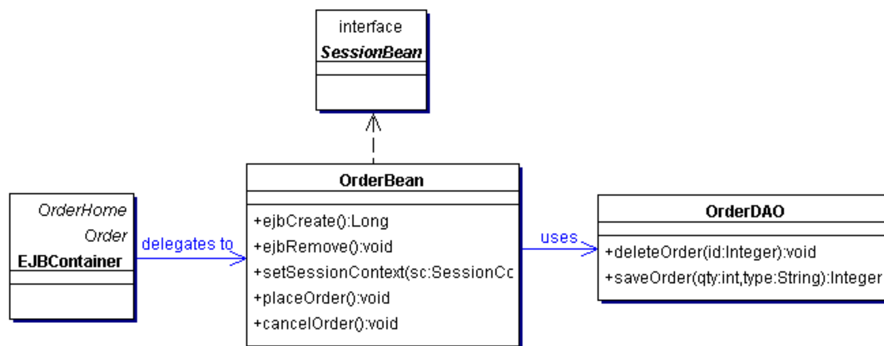

Fig. 4. EJB with helper class

In Fig. 4, the EJB OrderBean delegates the persisting of order information to the Data Access Object OrderDAO . The only way to use the container authorization service in order to control access to the methods exposed by OrderDAO is to model the class as an Enterprise JavaBean. This, however, is not in line with Sun's design pattern. It may have resource and performance drawbacks and has to be decided upfront at design time. The EJB container therefore intercepts only inter-EJB but not intra-EJB calls. The discussion above shows that the EJB container is insufficient to account for all crosscutting concerns in a complex J2EE application.

2.6 Summary

Using the EJB container to modularize crosscutting concerns works well as long as the following constraints hold:

- We do not need to configure any services provided by the container.
- We only use the deployment descriptor to associate an EJB with infrastructural services such as authorization. (No programmatic authorization checks.)
- We do not use any Java classes that live outside the context of an EJB.
- We do not want to make use of static type checking.

As shown, the criteria above constrain the development of a real world e-business application considerably. This, however, may not always be acceptable. A possible solution for some of the problems identified in 2.5 would be to enhance the EJB component model and allow developers to add custom interceptors to the EJB container along the lines proposed in Schmidt et al. (2000) and specified for the Servlet Engine in Sun (2001c). This would make the EJB container at least extensible. The latest EJB 2.0 specification (Sun 2001b) does not show any signs at all to add more flexibility to the component model, though. Adding flexibility to the EJB component model also comes at a cost. It would complicate the usage of the EJB technology and potentially endanger its acceptance and success as well as increase time-to-market and development costs.

3 AspectJ and Infrastructural Services

3.1 Introduction

This chapter briefly investigates the application of AspectJ (Kiczales et al. 2000) in order to associate infrastructural services that are usually provided by the Enterprise JavaBean container with base objects.

3.2 Sample Application

Let's assume we have the following simplified Order class as our base object:

```
public class Order {
int orderID;

public Order(int orderID) {
  this.orderID = orderID;
}

public void placeOrder() {
  //do some work
}

public void cancelOrder() {
    //do some work
}
}
```

The class above allows us to place and cancel orders. Suppose we now need to add authorization checks to Order. Suppose also we would like to check in particular if the requester has access rights to the methods placeOrder() and cancelOrder(). AspectJ (Kiczales et al. 2000) provides linguistic means to add authorization checks to Order in a transparent way.

3.3 AspectJ and Infrastructural Services

Authorization Aspect To add authorization to Order we can simply create a first authorization aspect in AspectJ that applies authorization checks to the methods of Order, cf.

```
aspect Authorization {

AuthorizationService as =
SecurityServiceFactory.getAuthorizationService
("StandardLogin");

pointcut authorizedMethods() : call(public void Order.*);

before() : authorizedMethods() {
  if (as.isCallerInRole("customer")) {
    return proceed();
  }
  else {
    throw new SecurityException("Access denied");
  }
}
}
```

The aspect above defines a pointcut that denotes those points in the execution of the program when instances of the Order class receive messages that correspond to public methods on Order with the return value void. The advice before() specifies what should happen before the methods referred to by

the poincut are executed. The advice uses role information provided by an instance of AuthorizationService and checks if the caller is in the appropriate role. The aspect therefore implements role-based authorization. AspectJ does a great job at allowing to add infrastructural services to base objects in an easy and straightforward way. Our Order class is now guarded by authorization checks. Every time a caller wants to execute the placeOrder() or cancelOrder() methods, role-based authorization is enforced.

Specifying Different Roles Let's assume we would like not only to execute authorization checks based on the role customer but rather to check for any role. And we also would like to specify role information without having to manually change the aspect code. This can be easily achieved using standard Java language mechanisms such as Property files, cf.

```
aspect Authorization {

AuthorizationService as = SecurityServiceFactory.getAuthorizationService("StandardLogin");

pointcut authorizedMethods() : call(public void Order.*);

before() : authorizedMethods() {
  Properties roles = getRoles();
  Enumeration enum = roles.propertyNames();

  while (enum.hasMoreElements() {
    if (as.isCallerInRole(roles.getProperty(enum.nextElement())) {
      return proceed();
      break;
    }
    else {
      throw new SecurityException("Access denied. Unauthorized role");
    }
  }
}

private Properties getRoles() {
  Properties authorizationProps = new Properties();
    FileInputStream authorizationStream = new FileInputStream("authorizationProperties");
    authorizationProps.load(authorizationStream);
    authorizationStream.close();
    return authorizationProps;
}
}
```

The code fragment above reads role values from a property file and checks if any of the roles has access rights to the relevant methods of Order. This roughly corresponds to multiple role entries for a public EJB method in an EJB deployment descriptor. If Order were an EJB the relevant section of the associated deployment descriptor would look like the following:

```
<method-permission>
<role-name>customer, admin</role-name>
<method>
  <ejb-name>Order</ejb-name>
  <method-name>*</method-name>
</method>
</method-permission>
```

The aspect Authorization allows us to add authorization to Order in a transparent way. It also offers us basically the same functionality as the interceptor framework of the EJB container regarding authorization without having to obey to any EJB programming restrictions. One of the great strengths of an EJB container, however, is that the container is easily reusable. We can drop any EJB into an EJB container and guard the bean's methods with authorization checks by editing the bean's deployment descriptor. How can we achieve reuse of the Authorization aspect?

Applying the Aspect to Further Classes Let's assume that our sample application consists not only of an Order class but also of a Product and a SAPConnector class. Let's also assume that the authorization checks applied to SAPConnector should be rather user-based than role based. In order to apply authorization checks to those two classes, we refactor our authorization aspect stated above and introduce two further aspects: An abstract aspect providing an abstract authorization pointcut and an aspect providing user-based authorization (see Fig. 5).

Our original authorization aspect has been refactored and is now called RoleBasedAuthorization. We made the aspect applicable to Product by manually editing the pointcut authorizedMethods(). The pointcut now denotes those points in the execution of the program when instances of the Product class receive messages in addition to the points in the execution of the program when instances of Order receive messages that correspond to public methods on Order with the return value void. The aspect RoleBasedAuthorization corresponds roughly to an EJB container plus the following two deployment descriptor fragments:

```
<method-permission>
<role-name>customer, admin</role-name>
<method>
  <ejb-name>Order</ejb-name>
  <method-name>*</method-name>
</method>
</method-permission>

<method-permission>
<role-name>customer, admin</role-name>
<method>
  <ejb-name>Product</ejb-name>
```

```
abstract aspect Authorization {

protected AuthorizationService as = SecurityServiceFactory.getAuthorizationService("StandardLogin");

abstract pointcut authorizedMethods();

protected void getRoles(String propertyFileName) {
  Properties props = new Properties();
    FileInputStream stream =
  new FileInputStream(propertyFileName);
    props.load(stream);
    stream.close();
  return props;
}
}

aspect RoleBasedAuthorization extends Authorization {

pointcut authorizedMethods() : call(public void Order.*) || call(public * Product.*);

before() : authorizedMethods() {
  Properties roles = getRoles("roleAuthorProperties");
  Enumeration enum = roles.propertyNames();
  while (enum.hasMoreElements() {
    if (as.isCallerInRole(
roles.getProperty(enum.nextElement())) {
      return proceed();
      break;
    }
    else {
      throw new SecurityException("Access denied. Unauthorized role");
    }
  }
}
}
aspect UserBasedAuthorization extends Authorization {

pointcut authorizedMethods() : call(public * SAPConnector.*);

before() : authorizedMethods() {
  Properties roles = getRoles("userAuthorProperties");
  Enumeration enum = roles.propertyNames();
  String userName = as.getUserPrincipal().getName();
  while (enum.hasMoreElements() {
    if (username.equals(
roles.getProperty(enum.nextElement())) {
      return proceed();
      break;
    }
    else {
      throw new SecurityException("Access denied. Unauthorized user ");
    }
  }
}
}
```

Fig. 5. ???

```
  <method-name>*</method-name>
</method>
</method-permission>
```

All we have to do to apply container services to an EJB is to deploy the bean together with its deployment descriptor. To change authorization checks we simply edit the deployment descriptor. There is no need to programmatically change any container or EJB code. In fact the container's interceptor framework is neither specified by Sun (2001b) nor is the source code of any commercial EJB container available. Let's take a look at our AspectJ-based solution. In our sample application, it is necessary to have the source code of the aspect available as well as being able to understand and modify AspectJ code. There is no other way in AspectJ to apply an aspect to additional classes apart from manually changing the aspect's pointcut.

Combining Different Aspects So far we have applied an authorization service to our three classes, Order, Product and SAPConnector. As mentioned in 3.1 EJB containers provide additional services such as persistence and transactions to EJBs. Suppose that we also want to provide a persistence and transaction service to Order in addition to the authorization aspect specified in 3.3.3. In order to do that we introduce a persistence and a transaction aspect, cf.

```
aspect Transaction {
pointcut transaction() : call(public void Order.*) ||
  call(public * Product.*);
before() : transaction() {
  //open transaction
}
after() : transaction() {
  //commit transaction
}
}
aspect Persistence {
pointcut persist() : call(public void Order.*) ||
call(public * Product.*);
after() : persist () {
  //persist the object to data store
}
}
```

The two aspect fragments above, Transaction and Persistence, apply additional services to Order and Product. Transaction opens and commits transactions for the appropriate methods on the two classes. Persistence persists the state of the object after the appropriate methods are executed on Order and Product. Neglecting the implementation details how a transaction service is accessed or provided (e.g. via the Java Transaction Service and the Java Transaction API) and how an object is persisted (e.g. via the Java Data Object or the Java Database Connectivity API), we would like to connect the authorization

aspect specified in 3.3.3 with the transaction and the persistence aspect. Let's assume that only authorized clients are allowed to access the base objects and that persisting the base objects is always done within a transaction. This leads to the following aspect chain: ¡RoleBasedAuthorization, Transaction, Persistence¿ There are two ways to chain aspects in AspectJ: aspect domination and inheritance. Using inheritance we could derive Transaction from RoleBasedAuthorization and Persistence from Transaction. This, however, would be an inadequate solution abusing inheritance to aggregate aspects. The aspect Transaction is after all no subtype and thus no special case of RoleBasedAuthorization. Therefore we do not discuss inheritance to chain aspects any further. Applying aspect domination means that the aspect RoleBasedAuthorization would explicitly declare that it dominates Transaction. Transaction in turn would explicitly state that it dominates Persistence, cf.

```
aspect RoleBasedAuthorization extends   Authorization
dominates Transaction { }

aspect Transaction dominates Persistence { }
```

Using domination leads to the following dependencies between the three aspects. RoleBasedAuthorization depends on Transaction, which itself depends on Persistence. Notice that each aspect also depends on the two base objects, Order and Product, cf. Fig. 6.

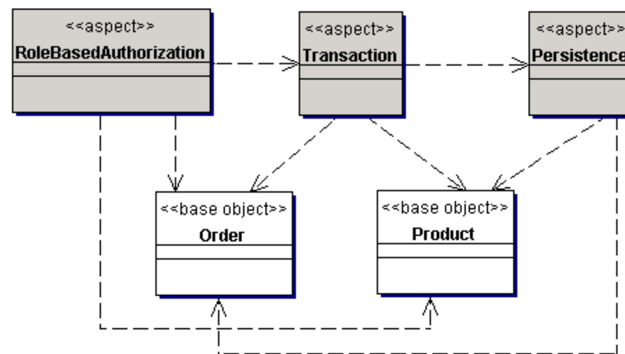

Fig. 6. ???

Domination works fine to chain together several aspects. It implicitly establishes an aspect framework, though, which is hard to adjust since the dependencies between the aspects are hard coded within the aspects themselves. If we wanted to change the domination relations we in the diagram above would have to manually change the AspectJ code. This is true for adding additional aspects, removing aspects and adopting aspects to a new application (reuse). Suppose

we change the persistence service used by the aspect Persistence to persist base objects. Suppose also that the new persistence service itself manages transactions for persisting data. The aspect Transaction is thus redundant and should be removed from the aspect chain. In order to achieve this we would have to manually change the code of RoleBasedAuthorization so that the aspect now directly dominates Persistence, cf.

```
aspect RoleBasedAuthorization extends   Authorization
dominates Persistence { }
```

In order to change RoleBasedAuthorization it is necessary to have the source code of the aspect available as well as being able to understand and change AspectJ code. Again, there is no other way in AspectJ to change an aspect chain apart from manually adjusting the aspects involved. Suppose we would like to reuse the aspect Transaction for another application. Suppose also that this application does not require Transaction to dominate any other aspect. In order to adopt Transaction to the new application we have to modify the aspect manually removing the dominates clause. Again it is necessary to have the source code of the aspect available as well as being able to understand and modify AspectJ code.

3.4 Summary

In this section we investigated the applicability of AspectJ to associate infrastructural services with base objects. The ultimate goal of this exercise is to replace the EJB container's interceptor framework by using AOP techniques. Employing AspectJ to apply authorization, transactions and persistence to base objects showed that AspectJ offers all the language mechanisms required to associate an infrastructural service with a base class. Using pointcuts we can apply an aspect to various base objects. Domination allows us to chain aspects together. There are, however, two prerequisites that make it very difficult to replace the EJB container's interceptor framework using AspectJ. In order to apply aspects in AspectJ to any base class, we need to have the source code of the aspect as well as the source code of the base class available. Also, to create or to modify an aspect chain based on domination, we have to have the aspect source code available. The two prerequisites are briefly discussed below. The prerequisite to have aspect and base object source code available seems to be impractical. If we imagine the next generation of EJB containers to be a set of aspects, no single vendor would be willing to ship any aspect source code. Similarly, third-party EJBs are available only as BYTE code. Having to ship the source code of base objects would make it impossible to sell base objects as off-the-shelf components. Secondly, in order to apply an aspect in AspectJ to any base object or to modify an aspect chain, we need someone who is able to read and modify AspectJ and Java code. Again, this prerequisite seems to be unrealistic if we consider the way Enterprise JavaBeans are configured today: A deployer simply adjusts the bean's deployment descriptor to associate certain infrastructural service such as authorization with the bean (Sun 2001). The deployer does not need to have

the source code available nor does he/she have to be able to understand any Java code. Not having to dive into source code reduces costs and deployment time and increases the acceptance of a technology. The prerequisites listed above make it unfeasible to replace the interceptor framework provided by EJB containers using current AspectJ language mechanisms. AspectJ however provides a good tool to quickly prototype the application of some (limited) infrastructural services to base objects via AOP techniques.

References

1. jBoss.org. jboss homepage, 2001. http://www.jboss.org.
2. Object Management Group. *CORBA Components Final Submission*. OMG TC Document orbos/99-02-05, 1999.
3. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Vol. 2*. Wiley, 2000.
4. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*. 2001.
5. Sun Microsystems. Java authentication and authorization service (jaas), reference guide for the java 2 sdk, standard edition, v 1.4, 2001. http://java.sun.com/products/jaas/index-14.html.
6. Sun Microsystems. *Java Servlet Specification, Version 2.3*. 2001.