SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–6 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

On aspectualizing component models

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

[1] *Siemens AG, Corporate Technology, D-81730 Munich, Germany*
[2] *Darmstadt University of Technology, D-64283 Darmstadt, Germany*

SUMMARY

Server-side component models such as Enterprise JavaBeans (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 concrete aspect-oriented languages shows that current AOP technology is not yet mature enough to supersede component models.

KEY WORDS: Aspect-Oriented Programming, Components, EJB

1. Introduction

Separation and modularization of concerns is an old and fundamental principle of software engineering, which has given rise to different forms of decomposition approaches at different stages in the history of programming languages and software engineering, with object-oriented decomposition being just the most recent mainstream approach. However, with object orientation finding its way into the construction of enterprise applications in the last decade, its restrictions especially with respect to modularizing infrastructural services such as security, transactions and persistence became evident.

In the terminology of aspect-oriented software development [9], infrastructural services as those mentioned above are called *crosscutting concerns* to indicate that given a good class-based modular structure of some base application, the implementation of these concerns cannot be encapsulated in a single module. Instead, the implementation is spread around several

*E-mail: roman.pichler@siemens.com
†E-mail: ostermann@informatik.tu-darmstadt.de
‡E-mail: mezini@informatik.tu-darmstadt.de

modules of the base application. Aspect-oriented software development is a new programming paradigm, which aims at providing explicit linguistic means for modularizing crosscutting concerns. In AspectJ [2], for instance, an aspect is a dedicated module that captures a crosscutting concern separate from the base objects by basically specifying what points in the execution of the base objects are affected by the crosscutting concern and how they should be affected. Given the base objects and aspect definitions, a weaver composes the different modules into a whole.

However, the aspect-oriented programming paradigm is not the only approach to modularize crosscutting concerns. Component models such as Enterprise JavaBean [16] try to achieve the same goal. The rationale underlying the programming model of the component technologies is to relieve the business logic developer from having to implement and manage infrastructural services. Component models achieve this by allowing to declaratively associate services with components at deployment time rather than to explicitly address them in the component implementation at design time.

Despite the commonalities of aspect-oriented programming and component models, an effort to put both trends into a common reference frame is still missing. This paper is a modest effort to fill this gap. To make things more concrete, we mainly focus on one aspect: authorization as an important security aspect. In a first step, we analyze how the Enterprise JavaBeans (EJB) technology [16] encapsulates infrastructural services and applies them to a base application, highlighting the strengths and the problems of Sun's server-side component model. Our observations also apply to other services and to other component models, such as the Corba Component Model (CCM) [14] or COM+ [6] due to the fact that all three component models are container-based.

In a second step we investigate how AspectJ as a representative of a linguistic AOP tool can be used to modularize the infrastructural services provided by the EJB component model. Later in the paper we explain to which extent our findings about AspectJ apply to other approaches of aspect-oriented software development available today. Our assumption is that aspect-oriented technology can at least partially take on the responsibilities fulfilled by a component model. We envision the next generation of component models to be a set of reusable aspects that can be easily attached to base objects viz. components. Our observations of the way AspectJ supports separation of crosscutting concerns shows that current AOP languages fail to adequately aspectualize component models. They also allow us to put forward some important conceptual prerequisites that next generation aspect-oriented languages should satisfy.

To summarize, the contribution of this paper is twofold. First, it provides an analysis of how component models and AOP languages help to modularize crosscutting concerns, highlighting the problems of both technologies and putting them in a common frame of reference. Second, we outline requirements to be met by future aspect-oriented languages if they want to be a key technology for developing the next generation of enterprise applications.

The remainder of this paper is organized as follows. Sec. 2 emphasizes the merits and flaws of the container approach in separating crosscutting infrastructural concerns. Sec. 3 outlines the merits and drawbacks of AspectJ as a representative of AOP languages. Sec. 4 puts forward requirements to be met by future aspect-oriented languages. Sec. 5 summarizes the work and outlines areas of future work.

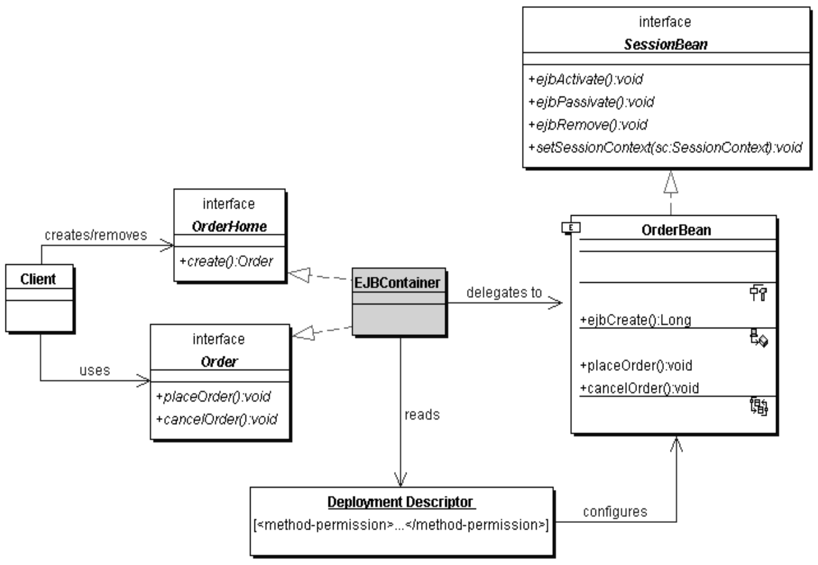

Figure 1. EJB Sample Application

2. Enterprise JavaBeans

2.1. Modularizing Crosscutting Concerns by the EJB Container

An EJB component is hosted by a container, which acts as the bean's runtime environment and provides the following infrastructural services to it: Resource and life cycle management, concurrency, security, persistence, transactions and remoting [16]. In order to be hosted by a container, a bean has to comply with certain idioms and programming restrictions. 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 from services like security or transactions, the EJB component model tries to encapsulate the infrastructural services (which are primary examples of crosscutting concerns) within the container.

Let us have a closer look how the EJB container encapsulates crosscutting concerns by means of the sample application in Fig. 1 that illustrates the association between an EJB and the container*. The EJB `OrderBean` provides order processing capabilities and exposes two

*For a more detailed account of the EJB container responsibilities, see [16].

```
<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>
```

Figure 2. An EJB Deployment Descriptor Fragment

interfaces to `Client`, `OrderHome` and `Order`, which allow the `Client` class to create an EJB instance, respectively to call business methods on it. `OrderHome`, `Order` and `OrderBean` have to be created by the application developer. The class `OrderBean` implements the business logic. It does not, however, implement the interfaces exposed to the client. The `OrderHome` and `Order` interfaces are implemented by the class `EJBContainer`, which schematically represents the EJB container.

When a client calls a business method on the component interface of an EJB, e.g., `placeOrder()` and `cancelOrder()` of `OrderBean`, the call is replaced by a call to the corresponding proxy object that is provided (generated) by the container. The proxy object informs the container about every call to the EJB. The container intercepts the method calls to the EJBs it hosts and applies additional actions that have to be performed, e.g., executing authorization checks, before the method calls are actually dispatched to the beans. 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.

Infrastructural services provided by the EJB container are associated with bean instances of type `Order` by editing the bean's deployment descriptor. A deployment descriptor is an XML file that makes it possible to configure EJBs declaratively. The EJB container reads the content of the deployment descriptor at deployment time and generates the code that applies the appropriate infrastructural services whenever the container intercepts a request to an EJB. For instance, we can specify authorization constraints for order beans by setting the appropriate values in the relevant section of the bean's deployment descriptor, cf. Fig. 2.

The deployment descriptor fragment in Fig. 2 states that only the roles `admin` and `customer` are allowed to execute methods on `Order` beans. The role `admin` is allowed to execute any

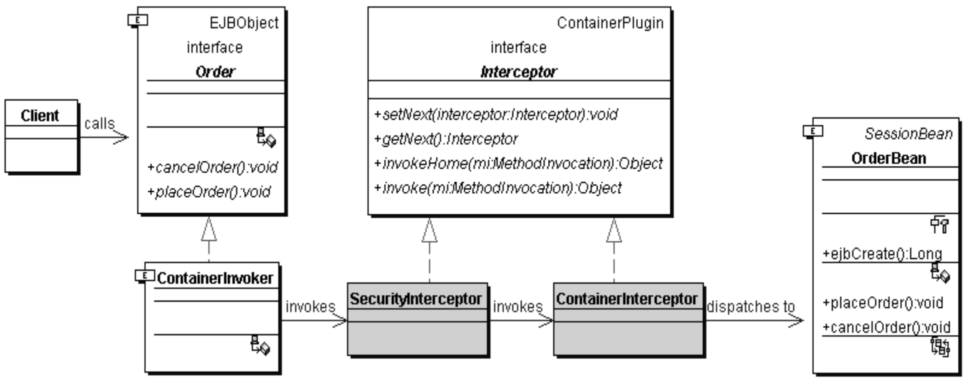

Figure 3. Simplified Interceptor Framework of the jBoss Container

method, while `customer` is only allowed to access `placeOrder`. Whenever the container intercepts a call to one of the business methods of the component interface `Order`, 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 the associated `OrderBean` instance. Otherwise, it aborts and throws an exception back to the client. Notice that the EJB deployment descriptor only supports a role-based authorization model.

Even though encapsulating services in dedicated classes and adding infrastructural services to a framework without affecting existing code is a prime example of the application of the interceptor pattern [15], the EJB specification [16] does not define any restrictions on the internal structure of the container. The container vendor is responsible for the number of and the dependencies between the container classes. Nevertheless, the design of some EJB containers does make use of the interceptor pattern. The open source application server jBoss [8], for instance, implements the interceptor framework shown in Fig. 3.

In Fig. 3, a client issues a request to `OrderBean`. Before the method call is dispatched to the EJB, a number of interceptors intercept the call and apply infrastructural services. For instance, `SecurityInterceptor` executes authorization checks (based on the information in the bean's deployment descriptor). jBoss requires that interceptors implement the interface `Interceptor`. This allows extending the container by adding custom interceptors. This solution, however, is not standardized by the EJB specification [16].

Using the interceptor pattern enables the EJB container to transparently add infrastructural services such as authorization checks to EJBs. Since infrastructural services are associated with an EJB via a deployment descriptor, the implementation of `OrderBean` in Fig. 3 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. The EJB container thus fulfills a similar role as an aspect in AOP [3, 7].

Even though an EJB container like the jBoss application server uses interceptors, the EJB specification treats the container as a black box. The specification does not allow configuring interceptors or adding custom interceptors and extending the container functionality.[†] The consequence of this approach is that the EJB technology is fairly easy to use, but the approach also imposes some important limitations.

2.2. 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.

2.3. Problems

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

2.3.1. Lack of Tailorability

The EJB specification [16] does not provide a mechanism to configure the container or any of its infrastructural services. The EJB specification also falls short to define how new services can be added to the container. As a consequence, a developer can either use the services provided by a commercial EJB container or decide not to use them at all. We call this *lack of tailorability.* The EJB component model regards the container as a black box and assumes that the container transparently encapsulates crosscutting concerns. The drawbacks of this approach are illustrated below.

Suppose we want to change the sample application introduced in Sec. 2.1 so that orders can only be placed by certain users. To meet the requirement, we have to employ user-based authorization, which is not provided by the EJB container. Since the EJB specification does not provide a way to extend the container and add a new authorization service, we have to explicitly address authorization in the EJB implementation (which, in fact, the EJB specification discourages to do).

[†]Notice that another J2EE specification, the Servlet specification 2.3 [17], 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. Developers are therefore able to extend the web but not the EJB container.

```
public class OrderBean implements SessionBean {
  public void placeOrder() throws EJBException, SecurityException {
    if (EJBContext.getCallerPrincipal.getName.equals("Mr. Jones"))
      //process the order
    else
      throw new SecurityException("Access denied. Unauthorized user");
  }
}
```

Figure 4. EJB and User-based Authorization

In the code fragment in Fig. 4, a user-based authorization check is executed in `OrderBean.placeOrder()`. The method 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. Authorization is thus hard coded in the bean.

Business and security logic are tangled in Fig. 4, and the two distinct concerns are mixed. As a consequence, an application developer has to take care of authorization checks while writing business logic. If other methods on `OrderBean` also need to employ user-based authorization, the security logic is spread over several places. This leads to poor understandability, maintainability and reusability of the security policy implementation.

Not being able to configure the container and its services leads to another drawback: Even if the application in Sec. 2.1 requires only security and transaction-related services, we get all the other container services as well. Since it is impossible to leave out crosscutting concerns that are not applicable to a specific application, container vendors offer and sell one package – the complete container.

Encapsulating infrastructural services in the EJB container and at the same time allowing developers to implement their own authorization (as well as transaction or persistence) 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. One could argue that if we need tailorability, we should use an EJB container such as jBoss that does allow adding custom interceptors and extending the container functionality. However, by doing so we commit to a non-standard solution and lose portability and reusability – two of the key rationales behind the component models.

2.3.2. Lack of Checking and Enforcement

An EJB developer has to obey many design rules and idioms that are not enforced and cannot be always checked by the compiler. For example, the EJB specification defines 17 programming restrictions [16, 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

```
Order order = new Order();
```

has to be replaced by

```
Context ctx = new InitialContext();
Object objref = ctx.lookup("Order");
OrderHome home =
  (OrderHome) PortableRemoteObject.narrow(objref, OrderHome.class);
Order order = home.create();
```

Although more than just a simple object creation happens in the container case, this is no justification for the loss of type information. A solution that were integrated into the programming language would let the programmer work with types instead of strings and do the necessary conversions behind the scenes, thereby preserving static type safety. In case `OrderBean` calls the method `calculate()` of another bean `DiscountBean` and wants to pass `this` as a parameter, it cannot simply pass on the self-reference as in

```
discount.calculate(this);
```

Rather, the `this` parameter has to be replaced by the corresponding proxy, which can be retrieved along the lines of

```
discount.calculate((OrderBean) SessionContext.getEJBObject());
```

Again, although the conversion above could be what *really* happens, an integrated solution should shield the programmer from such details and insert the necessary calls automatically.

The decisive point is that these EJB-specific 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 above) because, due to the use of universal methods, e.g., `getEJBObject()`, and strings instead of types, the type system is effectively discarded and all well-known benefits of static typing are lost.

2.3.3. Insufficiency

A real world application will have to use ordinary Java classes in addition to EJBs. Ordinary Java classes can profit from container services only if they live in the context of an EJB and even then only in a very limited way. 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 (see Lack of Checking and Enforcement). 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 using bean-managed persistence (BMP). 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

```
public class OrderBean implements SessionBean {
  DataAccessObject dao = new DataAccessObject();
  public void placeOrder() throws EJBException {
    dao.saveOrder();
  }
}
```

Figure 5. EJB and Ordinary Java Class

transparently by the EJB container). Employing ordinary Java classes and applying the *Data Access Object* pattern [1] allows us to do this. In the code fragment in Fig. 5, the Session bean `OrderBean` delegates persisting orders to the class `DataAccessObject`.

Suppose we would like to control the access to methods of `DataAccessObject` and grant only certain roles access rights. Suppose also that these roles may differ from the roles authorized to access `OrderBean`. Even though the EJB container does provide an authorization service, we cannot apply the service to `DataAccessObject` since it is an ordinary Java class and not an Enterprise JavaBean.[‡] Instead, we would have to provide an additional authorization service on the web server, e.g., by programmatically implementing role-based authorization within the `DataAccessObject` class mixing application with security logic.

We generally cannot associate services provided by the container with ordinary classes independent of the EJBs in whose context the Java classes live. Similarly, ordinary Java classes that live outside the context of an EJB (and thus outside an EJB container) cannot take advantage of any EJB container services. Therefore, we face the problem how crosscutting concerns can be modularized and how infrastructural services can be applied to ordinary classes.

2.4. 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 do not need to extend the container and add additional services.
- We only use the deployment descriptor to associate an EJB with infrastructural services such as authorization (i.e., no programmatic authorization checks).
- We do not use any Java classes that live outside the context of an EJB but require infrastructural service (for instance, authorization checks).
- We can afford to discard static type checking.

[‡]Please note that *local interfaces* as introduced by the EJB specification [16] are not applicable to `DataAccessObject` since `DataAccessObject` is an ordinary Java class and not an Enterprise JavaBean.

```
public class Order {
  public Order() {
  }
  public void placeOrder() {
    //place the order
  }
  public void cancelOrder() {
    //cancel the order
  }
}
```

Figure 6. Simplified `Order` Class

As shown, the criteria above constrain the development of a real world application considerably. A possible solution for some of the problems identified in Sec. 2.3 would be to enhance the EJB component model allowing to add custom interceptors to the EJB container along the lines proposed in [15], specified for the Servlet Engine in [17] and implemented by jBoss [8]. This would make the EJB container at least extensible. The latest EJB specification [16] does not show any signs to add more flexibility to the component model, though.

Enhancing the EJB component model and improving the flexibility of the EJB container also comes at a cost. It would complicate the usage of the EJB technology and potentially endanger its acceptance and success. Furthermore, while this would address the tailorability problem, the remaining two problems would still exist.

3. AspectJ and Infrastructural Services

3.1. Introduction

This section investigates the application of AspectJ 1.0 [2] to associate infrastructural services that are usually provided by the Enterprise JavaBeans container with base objects. The investigation is based on our vision of aspect languages and aspect-oriented frameworks replacing container frameworks by providing a set of reusable aspects that encapsulates infrastructural services and that can be applied to base objects on demand.

To make things more concrete, we base our discussion on a simplified order management application, whose EJB version was introduced in Sec. 2.1. We also focus on authorization as the core infrastructural service to be discussed. Let us assume we have the simplified `Order` class in Fig. 6 as our base object, which allows us to place and cancel orders.

Suppose that we have to make sure that only clients that have the appropriate access rights are allowed to invoke `placeOrder()` and `cancelOrder()`. To associate an authorization service with `Order` and to add authorization checks to the class in a transparent way, we simply create a first authorization aspect in AspectJ as in Fig. 7.

```
aspect RoleBasedAuthorization {
  AuthorizationService as =
    SecurityServiceFactory.getAuthorizationService();

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

  before() : authorizedMethods() {
    if (!as.isCallerInRole("customer"))
      throw new SecurityException("Access denied. Unauthorized role");
  }
}
```

Figure 7. Role-based Authorization in AspectJ

The aspect in Fig. 7 defines a pointcut that denotes those 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 advice `before()` specifies what should happen before the methods referred to by the pointcut are executed. The advice uses role information provided by an instance of `AuthorizationService` and checks if the caller is in the `customer` role. The aspect therefore uses linguistic means provided by AspectJ to associate an authorization service with `Order` and to implement role-based authorization. The `Order` class is now guarded with authorization checks. Every time a caller wants to execute the `placeOrder()` or `cancelOrder()` methods, role-based authorization is enforced. As a consequence, the aspect `RoleBasedAuthorization` fulfills a similar role in associating an authorization service with a base object as an EJB container does in applying authorization to an EJB.

3.2. Advantages

The example in Fig. 7 shows that AspectJ does a great job at allowing us to add infrastructural services to base objects in a straightforward and transparent way. It also offers us basically the same authorization functionality as the interceptor framework of an EJB container, cf. Sec. 2.1. In addition, we are not restricted to the services provided by the EJB container. We can, for instance, easily add an authorization aspect that applies user-based rather than role-based authorization to `Order` as shown in Fig. 8.

Unlike in the EJB example in Fig. 4 in Sec. 2.3.1, the implementation of `Order` in Fig. 8 is unaware of any authorization checks. Authorization is completely transparent to the base object. We could similarly add further infrastructural services, e.g., a persistence or QoS service, to existing base objects. The services can also be easily customized and exchanged by editing the appropriate advice. Additionally, no EJB-specific idioms and programming restrictions have to be obeyed. Base objects like `Order` in Fig. 6 do not have to implement specific interfaces such as `SessionBean` or `EntityBean` to be able to profit form infrastructural services. Unlike EJBs, they are also allowed to create threads and to use file I/O, cf. Sec. 2.3.2.

```
aspect UserBasedAuthorization {
  AuthorizationService as =
    SecurityServiceFactory.getAuthorizationService();

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

  before() : authorizedMethods() {
    if (!as.getUserPrincipal().getName().equals("Mr. Jones"))
      throw new SecurityException("Access denied. Unauthorized user");
  }
}
```

Figure 8. User-Based Authorization in AspectJ

Furthermore, `Order` is an ordinary Java class. AspectJ thus allows us to apply infrastructural services to any Java class using linguistic means instead of a combination of design patterns and standard object technology.

3.3. Problems

Even though AspectJ works fine to associate an authorization service with base objects, we encounter several problems when we try to substitute more container functionality using AspectJ language constructs.

3.3.1. Reusing Aspects

With respect to reusablity the strength of the Enterprise JavaBeans component model is twofold. First, EJB components are easily reusable in different application servers for different plattforms with eventually different implementations of the infrastructural services. 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 (supposing the container is compliant to the EJB specification [16]). The same is true for the implementation of infrastructural aspects encapsulated by an EJB application server, which are reusable with any EJB component deployed within the server. In this subsection we investigate how well reusability is supported in AspectJ. We use the `RoleBasedAuthorization` aspect introduced in Sec. 3.1 for illustration purposes.

Let us assume that our sample application consists not only of an `Order` class but also of a `Product` class. We would like to protect `Product` by authorization checks as well. To support better reusability of our aspects, we refactor `RoleBasedAuthorization` into an abstract aspect that provides an abstract pointcut definition. We also introduce a new aspect called `OrderProductAuthorization`, which extends `RoleBasedAuthorization` and applies the authorization checks to the two base objects, cf. Fig. 9.

```
abstract aspect RoleBasedAuthorization  {
  protected AuthorizationService as = ...;
  abstract pointcut authorizedMethods();
  protected before() : authorizedMethods() {
    //authorization check implementation goes here
  }
}
aspect OrderProductAuthorization extends RoleBasedAuthorization {
  pointcut authorizedMethods() :
    call(public void Order.*) || call(public * Product.*};
}
```

Figure 9. Aspect Reuse via Abstract Aspects

To apply authorization checks to `Order` and `Product` in Fig. 9, we don't change `RoleBasedAuthorization` but implement the pointcut of `OrderProductAuthorization` instead. The pointcut now also denotes those points in the execution of the program when instances of the `Product` class receive messages in addition to the points when instances of `Order` receive messages. `OrderProductAuthorization` thus fulfills a similar role as a *binding* in [12, 13].

Employing abstract aspects to increase aspect reusability in AspectJ has the following drawbacks: First, using concrete aspects as connectors may easily lead to a complex and bloated aspect hierarchy, which is hard to understand, extend and maintain. Second, when we employ an abstract aspect, it is still necessary to have the aspect source code available as well as being able to understand and modify AspectJ code in order to apply an authorization service to base objects. As pointed out in [13] and [5], the process of binding an abstract aspect to a concrete base object via inheritance requires sophisticated programming skills and is by no means straightforward. Third, applying abstract aspects is limited especially if aspects define aspect instance state, as discussed in detail in [13] and [5].

EJB technology makes it certainly much easier to apply infrastructural services to an Enterprise JavaBean. If `Order` and `Product` were EJBs, we would just deploy the two EJBs together with their deployment descriptor. The container would then automatically apply the appropriate services. To modify 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 the EJB specification [16] nor is the source code of any commercial EJB container available (see Sec. 2.1 for discussion).

3.3.2. Deploying Aspects

AspectJ requires that aspects are applied to base objects at design time (base object classes are hard coded in pointcuts and aspects are compiled together with the base object code). As a consequence, the application of infrastructural services to classes like `Order` and `Product` cannot be changed at deployment time. Using EJBs, however, infrastructural services are applied at deployment time via the deployment descriptor. Some J2EE application servers even allow deployment descriptor changes in production (*hot deployment*). Delaying the application of infrastructural services until deployment time increases the flexibility and reusability significantly. Besides the useful delay of the deployment to runtime, this also enables to choose the actual implementation of an aspect at runtime, i.e., we could choose the transaction- or persistence implementation at runtime. In AspectJ, however, there exists a static link between a particular aspect implementation and the base classes, and we have no means whatsoever to change this link without modifying code (which is bad), and even if we come to terms with code modification, this cannot be done after compile time.

3.4. Summary

Our investigation of using AspectJ to apply authorization, transaction and persistence services 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. Employing an aspect-oriented language rather than a container interceptor framework to transparently apply infrastructural services to base objects allows us to tailor the services according to our requirements. No idioms or programming restrictions are imposed and infrastructural services can be applied to any Java class. This clearly shows that AOP languages can provide infrastructural services and replace container technology while introducing additional benefits including tailorability and general applicability of the services.

There are, however, several issues that make it very difficult to replace the EJB container's interceptor framework using AspectJ. First, it is very hard to make aspects reusable. If we cannot reuse aspects for infrastructural services, the main benefit of using a container would be gone because we have to do all the work ourselves. Second, there is no notion of deployment in the AspectJ world. All we can do to integrate aspects is to hardwire them in the code, we cannot deploy aspects at runtime and choose a particular aspect implementation (or variant).

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 prototype the application of some (limited) infrastructural services to base objects using AOP techniques.

4. Towards a Marriage of Aspects and Components

4.1. Introduction

Summing up our discussion of the previous sections, the key observations are: Neither component models nor today's aspect-oriented languages can adequately meet the challenges of enterprise application development. The reliance on design patterns, idioms and programming restrictions are the Achilles' heel of component models. Current AOP languages are not capable of properly modularizing infrastructural service since they lack black-box reuse due to their code transformation centered approaches. They also fall short of providing appropriate deployment support. Based on these observations, this section puts forward a few essential requirements a model of software development with a strong commitment to a separation of crosscutting concerns should fulfill. The next generation AOP languages and containers will be characterized by the following properties:

1. Future AOP languages will turn conventions and idioms of today's component models into language features. For instance, instead of relying on conventions that components have to follow in order to enable containers to take control over their execution and attach infrastructural services to them in a transparent way, an intrinsic feature of the language concept of a reusable component will be that its operations are late bound depending on the runtime context of the component, i.e., on the available services in this context. We call the polymorphism resulting from this type of late binding *aspectual polymorphism*.

2. The next generation of AOP languages will allow aspectualizing component models by providing appropriate linguistic means to capture individual infrastructural services in separate modules. While today's AOP languages such as AspectJ basically support this property as indicated by the discussion in Sec. 3, the key difference is that the infrastructural service modules we envision will be (a) as easy to reuse within different applications as the infrastructural services provided by today's application servers, (b) easy to compose to build custom service infrastructures based on the concrete application needs, and (c) dynamically deployable to support the dynamic adaptability of the infrastructure depending on runtime conditions.

The remainder of this section first elaborates on the points stated above. We envision future AOP languages to enable developers to employ a reusable set of infrastructural aspects that can be easily attached to base objects. Our own language for aspect-oriented programming, Caesar [12, 13, 4], is based on these observations. After stating our general outline of future AOP languages/containers, we will very shortly discuss how (and to which degree) our requirements are met in Caesar.

4.2. Language Integration

In the history of programming languages, idioms and patterns of good style have been turned into rules imposed by language compilers. Next generation models for separating crosscutting concerns should follow this approach by turning the conventions of component frameworks into language features. The underlying rationale is that a software development model that is based on idioms and patterns is more difficult to apply and renders maintenance harder since it requires that developers carefully express concepts in a language that does not support the concepts in the first place.

While conceptually working with components, an EJB developer has to map the higher-level component concept into language means designed to express lower-level concepts such as objects (i.e., Java classes and interface). The resulting software is harder to understand since it encodes the mapping of a higher-level concept to lower-level language constructs rather than the concept itself. For instance, the sample code for creating a new `Order` object in Sec. 2.3.2

is extremely hard to understand for someone unfamiliar with remote object communication protocols. The code is prone to become invalid as soon as the remote communication protocols change and because of extra-language coding conventions, the benefits of the static type system are lost. Due to the idioms enforced by the EJB component model, the encapsulated crosscutting concerns show up at several places in the application code like the top of an iceberg.

One of the most important features that is realized via patterns and idioms in today's component models is some sort of *aspectual polymorphism.* An EJB component, for instance, must be prepared to live in a context where its functionality is extended with infrastructural services, e.g., it has to obey the rule not to pass an immediate reference to itself by means of `this` as a parameter, but rather the `EJBObject` instance associated with it.

Instead of having to follow certain conventions to simulate aspectual polymorphism, next generation languages with support for separating crosscutting concerns should integrate such polymorphism as a key concept into the language. For instance, the meaning of `this` during a method invocation should be late bound not only based on the concrete variant of an abstraction at hand, e.g., depending on whether `placeOrder()` is invoked on a standard or on an express order object, but also based on the infrastructural services available to the component's runtime system at the time of the call.

Today's AOP languages do not support aspectual polymorphism. The approach used by most of the current "mainstream" AOP languages to bind aspects is mainly based on some sort of either source-code or byte-code transformation that weaves aspect code (or meta-level code for triggering execution of the aspect code) into the right place of the base code. Compiling AspectJ code, for instance, happens in two steps: A preprocessor first transforms the Java base code integrating the aspect code. Second, the compilation of the merged Java code takes place. The current implementation of Hyper/J [18] basically follows the same approach. The only difference is that Hyper/J creates a new name space for the merged code instead of in-place modifications.

As an analogy, let us consider how subclass polymorphism is used to separate concerns that relate to different kinds of a data abstraction in object-oriented languages. In order to introduce, say, express orders by extending the existing implementation of a generic order abstraction in a base class `Order`, we do not use add-on language preprocessing tools to insert the specific code for express orders into the `Order` code. With subclass polymorphism of object-oriented languages, any abstraction that we write has the built-in potential of being *incrementally* extended with future variations.

An *incremental* extension of the base functionality by crosscutting concerns is missing in today's AOP languages. With AspectJ for instance, the extension happens in-place by modifying the code of the base functionality to integrate the extension. The modified code physically replaces the original code. The extension is not incremental in Hyper/J either: While the result of composing a crosscutting concern with an existing hyperslice is a new hyperslice, the latter is merely a copy of the former transformed to integrate the implementation of the crosscutting concern. It is by no mean related to the original hyperslice by subtyping.

With aspectual polymorphism, no transformation of a base object's or component's code at development and even deployment time is required. Hence, AOP technology with support for aspectual polymorphism will not rely on the availability of source code or on tools for byte-code transformation. This is an important prerequisite in the context of container-based software development. In order to protect their intellectual property, component providers are not willing to deliver the source code. Additionally, approaches based on byte-code transformations only work on languages with a well-structured byte code such as Java. They are not applicable to classes written in other languages such as C++. Furthermore, byte code transformation might render component testing more difficult.

To summarize, aspectual polymorphism is the key to bridge the gap between tailorability as provided by current AOP technology and black-box reuse as provided by container-based approaches.

4.3. Deploying Components into Aspectual Contexts

Besides the integration of component concepts and aspectual polymorphism, another key feature of the next generation AOP technology is the support for declarative deployment of base objects and components into any given aspectual context similar to today's container-based deployment. The term aspectual context is used to denote the composition of infrastructural services. The process of weaving aspects that encapsulate infrastructural services into application logic with next generation AOP technology should be at least as simple, declarative, and tool supported as the deployment of EJB components is today. Two key and interrelated issues need to be addressed with this respect: better reusability of aspects and postponing the deployment of beyond design time.

The discussion of reusability problems of aspects in Sec. 3.3 showed that aspects in AspectJ are associated with their base objects at design time. The association is hard coded in the aspects' pointcut. AspectJ does neither employ a dedicated module to associate aspects with base objects nor does it allow to specify the association at deployment time. To be able to easily apply aspects to different base objects, the aspects have to be reusable in the sense that they are not coupled to specific base objects. In Hyper/J, a hypermodule is not directly coupled to the code with which it is composed. Instead, the composition specification is outsourced into a *hypermodule specification*. However, the frequent use of so-called *merge-by-name* composition may lead to indirect coupling of different modules due to reliance on identical method and class names.

In the APPC [11] and Aspectual Components [10] approach, the aspect functionality is completely separated from the specification. An explicit connector construct associates the aspect code with the base code. However, similar to AspectJ and HyperJ, APPC and Aspectual Components also take a code transformation approach.

In container-based environments, infrastructural services are applied and configured at deployment time. This is essential for the separation of competence among the different EJB roles (EJB provider, application assembler, deployer, container provider etc. [16]). For example, the EJB provider is a domain expert and is responsible for the production of specific EJB components, whereas the deployer is responsible for deploying the components in a specific operational environment.

In most AOP approaches, the binding of aspects to base code happens at compile time, though. This renders the aforementioned separation of competence impossible. Therefore, we should strive for some kind of *late binding* of aspects, whereas "late" might refer to deployment or even runtime. Deploying aspects at runtime is necessary for *hot deployment* and enables the creation of aspects whose applicability depends on runtime conditions. For example, the applicability of an optimization aspect like instance pooling may depend on current system resources or system load.

4.4. First Steps Towards the Vision: The Caesar Model

Caesar is our own proposal to cope with the problems which were identified in the course of this paper. Details about Caesar can be found in other papers [12, 13] and on the project homepage [4]. In this section we want to discuss shortly how Caesar adresses the main problems identified with AspectJ: reusability, deployment and aspectual polymorphism.

4.4.1. Reusability

In Caesar, aspect *implementations* and aspect *bindings* into a base code are separated from each other. Aspect implementations play a similar role as the abstract aspect in Fig. 9, whereas bindings play a similar role as the concrete aspect in Fig. 9. However, there are a number of important differences as compared to the AspectJ approach. First, the aspect binding and implementation are defined in completely independent program units which are related to each other by means of a so-called *collaboration interface*, a specification of the common communication protocol. Hence, in contrast to Fig 9, the binding `OrderProductAuthorization` would not be hardwired to the aspect `RoleBasedAuthorization`. This separation enables us to compose different bindings with different implementations of the same collaboration interface, hence both implementations and bindings can be reused independently.

Second, our bindings are equipped with dedicated linguistic means for a hierarchy transformation in the sense that the data model (names, relations etc.) of the application can be transformed to the data model used by a particular family of aspects. Hence, the aspect can be written in terms of its own 'world' and is thus truly independent from a particular application.

Third, the structure of aspects in Caesar is that of a collaborating family of entities, whereby state and behavior can be applied independently to each entity. Hence, in contrast to AspectJ, it is easy to cope with aspect instance state (cf. the discussion in Sec. 3.3.1).

4.4.2. Deployment and Aspectual Polymorphism

Although Caesar has pointcuts and advices similar to those in AspectJ, there is an important difference: By default, pointcuts and advices in Caesar are passive, that is, their declaration and compilation alone does not have any computational effect on the remainder of the software – aspects have to be *deployed* before their pointcuts and advices become effective. For the deployment, there exists a special `deploy` clause in the language, with which it is possible to activate an aspect in the context of a method call (the aspect does not become active globally because that would lead to undesirable concurrency problems).

During this deployment, it is possible to make use of the subtype polymorphism of aspects in `Caesar`: All aspects are subtypes of their collaboration interface (cf. above), such that it is possible to choose among different implementations (i.e., different aspect variants) of the collaboration interface at runtime.

5. Summary and Future Work

In this paper, we compared two important approaches to modularize crosscutting concerns such as transaction, persistence and security: Server-side component models represented by the Enterprise JavaBeans (EJB) component model and aspect-oriented programming (AOP) represented by AspectJ. Our investigation showed that both approaches have their strengths as well as their drawbacks. Based on this observation, the paper reasoned about the possibility of combining the strength of both approaches into a future technology for separating crosscutting concerns, while avoiding their respective problems.

The paper put forward important requirements in order to achieve this goal. We first argued that patterns and idioms for separation of crosscutting concerns underlying the design of component frameworks should be integrated into languages. Broadly speaking, we strive to provide language constructs that directly support the concept of a component instead of requiring the application developer to obey rules and idioms for mapping the higher-level component concept onto the lower-level concepts of objects.

We also highlighted aspectual polymorphism as one of the key features that emerges when idioms and patterns are turned into language features. Similar to the definitions of a base class that has the built-in capability to late bound via subclass polymorphism, we promote the idea of having the built-in ability of component definitions to be late bound, based on the infrastructural services available in a deployment context. This feature will be the key to bridge the gap between the tailorability supported by current AOP technology and the black-box reuse promoted by container-based approaches. In addition, we proposed that next generation AOP languages should provide deployment constructs equivalent to the deployment descriptors of component models and that runtime deployment should be supported.

We gave a very short introduction to our model to cope with these problems, Caesar. Our future work will continue to focus on improving Caesar based on these requirements and applying it to the design of next generation component models.

REFERENCES

1. D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns. Best Practices and Design Strategies.* Prentice Hall, 2001.
2. AspectJ Homepage, 2001. http://aspectj.org.
3. Gregory Blank and Gene Vayngrib. Aspects of enterprise java beans. In *ECOOP Workshop on Aspect-Oriented Programming*, 1998.
4. Caesar Homepage. http://www.st.informatik.tu-darmstadt.de/pages/projects/caesar.
5. Siobhn Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.
6. Guy Eddon and Henry Eddon. *Inside COM+.* Microsoft Press, 1999.

7. Stephan Herrmann, Mira Mezini, and Klaus Ostermann. Joint efforts to dispel an approaching modularity crisis. In *Sixth International Workshop on Component-Oriented Programming at ECOOP*, 2001.
8. jBoss Homepage, 2001. http://www.jboss.org.
9. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, 1997. Springer-Verlag.
10. Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, March 1999.
11. Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.
12. Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA '02, Seattle, USA*, 2002.
13. Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proc. International Conference on Aspect-Oriented Software Development (AOSD '03), Boston, USA*, 2003.
14. Object Management Group. *CORBA Components Final Submission.* OMG TC Document orbos/99-02-05, 1999.
15. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Vol. 2.* Wiley, 2000.
16. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0.* 2001.
17. Sun Microsystems. *Java Servlet Specification, Version 2.3.* 2001.
18. Peri Tarr and Harold Ossher. Hyper/J user and installation manual, 1999. http://www.research.ibm.com/hyperspace.