Vrije Universiteit Brussel
Faculty of Science
Department of Computer Science
System and Software Engineering Lab

Aspect-Oriented Software Development for the Java 2 Platform, Enterprise Edition

Niels Joncheere

June 2004

Advisor:

Prof. Dr. Viviane Jonckers

A dissertation in partial fulfilment of the requirements of the degree of Licentiate in Computer Science.

Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica
System en Software Engineering Lab

Aspectgeoriënteerde Softwareontwikkeling voor het Java 2 Platform, Enterprise Edition

Niels Joncheere

Juni 2004

Promotor:

Prof. Dr. Viviane Jonckers

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica.

Abstract

Over the last decade, Component Based Software Development (CBSD) has become one of the major trends in software engineering. Another popular trend is Aspect-Oriented Software Development (AOSD). Based on the idea that an aspect-oriented approach for CBSD would be useful, the Vrije Universiteit Brussel's System and Software Engineering Lab (SSEL) developed an aspect-oriented programming language specifically tailored for CBSD, named *JAsCo*. This language allows an application developer to create aspect-oriented, component based applications using an "aspect-enabled" extension to the JavaBeans component model. In industrial applications, however, the JavaBeans component model is often discarded in favour of the Enterprise JavaBeans (EJB) model defined by the Java 2 Platform, Enterprise Edition (J2EE).

Although J2EE already provides applications with several services — such as security and transaction management — that are in fact crosscutting concerns, one cannot implement general-purpose aspects in J2EE. Hence, the integration of JAsCo into J2EE can make J2EE applications more comprehensible and reusable because of better modularisation of crosscutting concerns. On the other hand, JAsCo, too, can benefit from integration into J2EE, as it is interesting to demonstrate how the language can be applied to a full-fledged component model like the EJB model.

In this dissertation, we investigate how JAsCo can be fully integrated into J2EE. We propose a backward compatible extension to the Enterprise JavaBeans component model that introduces *aspect beans* into the model. Such aspect beans have all the characteristics of the aspect beans introduced by the original version of JAsCo. In our approach, EJBs are provided with built-in traps that are used to attach and detach aspect beans.

Just like in the original version of JAsCo, aspect beans do not have any reference to where they will be applied. In the original version of JAsCo, explicit *connectors* are responsible for linking aspect beans to a concrete context. We propose an extension to the EJB deployment descriptor that makes this deployment descriptor responsible for providing aspect beans with a concrete context: the deployment descriptor will link the EJBs to the aspect beans. We describe how the component model can allow runtime enabling and disabling of aspect-oriented functionality, and where the aspect-oriented functionality will be added to the main application. We also describe whether or not session, entity, and message-driven aspect beans are useful. Finally, we provide a limited Proof of Concept (PoC) implementation that implements our extension to the EJB deployment descriptor.

Samenvatting

Componentgebaseerde softwareontwikkeling (CBSD) is tijdens het laatste decennium uitgegroeid tot één van de belangrijkste trends in software engineering. Een andere populaire trend is aspectgeoriënteerde softwareontwikkeling (AOSD). Het System en Software Engineering Lab (SSEL) van de Vrije Universiteit Brussel heeft, gebaseerd op de idee dat een aspectgeoriënteerde aanpak voor CBSD nuttig zou zijn, een aspectgeoriënteerde programmeertaal ontwikkeld die specifiek op CBSD gericht is. Deze taal, *JAsCo*, laat een ontwikkelaar toe aspectgeoriënteerde, componentgebaseerde toepassingen te maken door gebruik van een "aspectgeactiveerde" uitbreiding van het JavaBeans componentmodel. In industriële toepassingen wordt het Enterprise JavaBeans (EJB) componentmodel, gedefiniëerd door het Java 2 Platform, Enterprise Edition (J2EE), echter vaak verkozen boven het JavaBeans componentmodel.

Hoewel J2EE toepassingen al van verschillende diensten voorziet — zoals veiligheids- en transactiebeheer — die in feite crosscutting concerns zijn, kan men geen aspecten voor algemene doeleinden schrijven in J2EE. Bijgevolg kan de integratie van JAsCo in J2EE toepassingen beter begrijpelijk en herbruikbaar maken omwille van een betere modularisatie van crosscutting concerns. Anderzijds kan ook JAsCo voordeel halen uit een integratie in J2EE, aangezien het interessant is om aan te tonen hoe de taal kan toegepast worden op een volwassen componentmodel als het EJB model.

In deze thesis onderzoeken we hoe JAsCo volledig geïntegreerd kan worden in J2EE. We stellen een achterwaarts compatibele uitbreiding voor van het Enterprise JavaBeans componentmodel die *aspect beans* in het model introduceert. Dergelijke aspect beans hebben alle eigenschappen van de aspect beans geïntroduceerd door de oorspronkelijke versie van JAsCo. In onze aanpak worden EJBs voorzien van ingebouwde vallen die gebruikt worden om aspect beans vast en los te maken.

Net als in de oorspronkelijke versie van JAsCo hebben aspect beans geen enkele verwijzing naar waar ze zullen toegepast worden. In de oorspronkelijke versie van JAsCo zijn expliciete *connectors* verantwoordelijk voor het linken van aspect beans aan een concrete context. Wij stellen een uitbreiding voor van de EJB deployment descriptor die deze deployment descriptor verantwoordelijk maakt van het voorzien van aspect beans van een concrete context: de deployment descriptor zal de EJBs linken aan de aspect beans. We beschrijven hoe het componentmodel runtime aan- en uitschakelen van aspectgeoriënteerde functionaliteit kan toelaten, en waar de aspectgeoriënteerde functionaliteit aan de hoofdtoepassing zal toegevoegd worden. We beschrijven ook of session, entity, en message-driven aspect beans nuttig zijn. Tenslotte geven we een beperkte Proof of Concept (PoC) implementatie die onze uitbreiding van de EJB deployment descriptor implementeert.

Acknowledgements

This dissertation would be far from finished if it wasn't for the support of several people. Therefore, I would like to thank the following persons:

Prof. Dr. Viviane Jonckers, my advisor, for her invaluable help in shaping the ideas presented in this dissertation, and for proofreading it.

Wim Vanderperren and Davy Suvée, the original authors of JAsCo, for helping me whenever I needed a minute of their time, and for proofreading this dissertation while it was being prepared.

Francky Desmet, my employer at the Delhaize Group's IT department, for giving me time off during the final phase of completing this dissertation, thus helping me to combine my full-time studies with a part-time job.

Françoise Hemelinckx, my mother, for providing me with the opportunity to pursue a university degree.

Table of Contents

List of Figures

List of Code Fragments

Chapter 1 Introduction

1.1 Problem Statement

A recurring problem throughout the history of computer science is the question of how one can manage software projects of increasing complexity: as the amount of available computing power increases, so does the complexity of programs and program requirements, making it difficult to finish projects on time and within budget. This problem has been called the *software crisis* [Dij72], and endures until today. The software crisis lies at the basis of the discipline of software engineering, and many software engineering techniques have claimed to provide a solution to it. By now, however, it is widely accepted that there is no single solution to the problem [Bro86].

An important contribution to a solution of the software crisis has been Object-Oriented Software Development, which decomposes an application into *objects* that encapsulate data and the operations on that data, thus enhancing maintainability and reusability. There are, however, some limitations to OOSD: reuse in object-oriented systems is often achieved through inheritance, resulting in a very strong coupling between a base class and its derived classes; a change to a base class could require changes to every derived class, or else the application would stop working [MS98].

Component Based Software Development attempts to improve software maintainability and reusability by defining the development of an application as the process of combining a number of black-box off-the-shelf *components*: a component is a unit of composition with explicit interfaces and dependencies [Szy97], which implements a certain service. The application developer does not need to know what the component does in order to achieve its functionality, and as they can only be accessed by their explicit interface, coupling is reduced to a minimum.

Both OOSD and CBSD, however, suffer from the *tyranny of the dominant decomposition* [OT99]: an application can only be modularized in one way at a time, and concerns that do not align with this modularization end up scattered across many modules and tangled with one another. Aspect-Oriented Software Development [KLM+97] attempts to increase the separation of concerns by allowing application developers to describe *crosscutting* concerns in separate modules — *aspects* — so that adding, modifying, or removing such concerns does not require changes to the rest of the system.

In the last few years, a multitude of aspect-oriented programming languages has been introduced; examples of well known languages are AspectJ [KHH+01], HyperJ [OT00], Adaptive Programming [LOO01], and Composition Filters [BAT01]. Most currently available aspect-oriented approaches, however, are focused on the object-oriented paradigm, but CBSD also suffers from the tyranny of the dominant decomposition: in an attempt to reduce coupling as much as possible, a lot of functionality is spread out and repeated among different components. Hence, an aspect-oriented approach focused on CBSD could significantly contribute to the component-based paradigm.

Therefore, Davy Suvée et al. developed the JAsCo language [Suv02, SVJ03, Van04]. JAsCo is an aspect-oriented programming language specifically tailored for component based software development. Based on the idea that in a component based situation, aspects should have much of the characteristics of components (i.e. they should be easily attached to or detached from other components, they should remain separate entities at runtime, etc.), JAsCo proposes an "aspect-enabled" extension to the JavaBeans component model, which provides components with built in traps that can be used to attach aspects.

The original implementation of JAsCo is based on the relatively simple JavaBeans component model [Ham97]. This made sense at the time JAsCo was first being developed: as the JavaBeans component model is both simple and widely used within the software engineering research community, it is ideal for research experiments. In industrial applications, however, the Java 2 Platform, Enterprise Edition's Enterprise JavaBeans model is the most widely used component model [DYK01, Sha03]. Although J2EE already provides applications with several crosscutting functionalities, an aspect-oriented extension that allows general-purpose aspects to be implemented could significantly contribute to the EJB component model. On the other hand, such an extension could prove useful to JAsCo, too, as it would be interesting to see whether the language can be applied to a full-fledged component model.

1.2 Goal, Context, and Approach

The goal of this dissertation is to investigate how JAsCo can be integrated into J2EE. We aim to propose an extension to the Enterprise JavaBeans component model that is "aspect-aware", in which JAsCo fits as well as possible. We also aim to provide a limited Proof of Concept (PoC) implementation that illustrates our ideas.

The context in which this research is conducted is the Java 2 Platform, Enterprise Edition introduced by Sun, and the JAsCo aspect-oriented programming language proposed by Davy Suvée et al.:

The Java 2 Platform, Enterprise Edition is one of the most widely used platforms for enterprise application development in existence. The J2EE 1.4 specification defines three important entities: JavaServer Pages, Java Servlets, and Enterprise JavaBeans. The latter is of most interest to us, as it implements J2EE's server-side component architecture. The EJB 2.0 specification defines three kinds of EJBs: session beans, entity beans, and message-driven beans. EJBs are *deployed* on a J2EE *application server*. When it is deployed, an EJB is wrapped by a *container* that provides the EJB with several services, such as security and transaction management. EJBs are always accompanied by a *deployment descriptor* that describes how the EJB should be deployed. Using several EJBs, one can create large, distributed, component-based applications; the EJB component model makes J2EE the de facto standard for component based software development.

JAsCo is an aspect-oriented programming language for the JavaBeans component model. It introduces two new concepts into the model: *aspect beans* define the crosscutting functionality that may be added to one or more components, and do not have any reference to where they will be applied; *connectors* provide one or more aspect beans with a concrete context, thus linking the aspect beans to the components.

Our first approach to the problem of integrating JAsCo into J2EE attempts to provide a working implementation of JAsCo for J2EE in a simple and straightforward manner, taking into account the particularities of J2EE as little as possible. Using this approach, we aim to achieve some valuable insight into the technicalities of extending JAsCo to J2EE; the approach can be a useful stepping stone towards a more extensive integration.

The approach is based on the idea that the original implementation of JAsCo, which was designed for the Java 2 Platform, Standard Edition (J2SE), should — in theory — also function correctly when applied to J2EE applications: as JAsCo uses Java byte-code manipulation to add its aspect-oriented functionality, and both J2SE and J2EE applications are compiled to byte-code, JAsCo must be able to add its aspect-oriented functionality to J2EE applications, too. We provide a slightly modified version of the original implementation that functions correctly for J2EE.

Our second approach to the problem attempts to fully integrate JAsCo into J2EE: we propose a new component model — based on the Enterprise JavaBeans component model — that is "aspect-aware". Our new "Enterprise JAsCoBeans" model is a backward compatible extension to the EJB model, which introduces a new kind of entity — called aspect beans — into the model. These aspect beans have the same characteristics as the aspect beans in the original version of JAsCo. Enterprise JAsCoBeans contain built-in traps that are used to attach and detach aspect beans.

Just like the aspect beans in the original version of JAsCo, the aspect beans in our new component model do not have any reference to where they will be applied. Our second approach makes the EJB deployment descriptor responsible for linking its EJBs to aspect beans, which thus provides these aspect beans with a concrete context.

We also discuss several other issues related to the integration of JAsCo into J2EE: we describe how one can allow application developers to enable and disable aspect-oriented functionality at runtime, we investigate whether the aspect-oriented functionality is best placed in the EJB containers or in the EJBs themselves, and we discuss whether or not session, entity, and message-driven aspect beans can be useful.

1.3 Outline of the Dissertation

In Chapter 2, we describe the dissertation's research context. We provide an introduction to component-based software development and discuss two common component models, i.e. the JavaBeans and Enterprise JavaBeans component models. We also provide an introduction to aspect-oriented software development, and discuss AspectJ, one of the most widely used aspect-oriented languages in existence. We conclude the chapter by discussing why it is useful to combine CBSD and AOSD, and introduce JAsCo as a practical example of this principle.

In Chapter 3, we describe a straightforward approach to implementing a version of JAsCo for J2EE, taking into account the particularities of J2EE as little as possible.

In Chapter 4, we describe a more elaborate approach to implementing a version of JAsCo for J2EE, which aims to integrate JAsCo into J2EE as well as possible. We propose a new component model, based on the EJB component model, which introduces aspect beans. We describe how these aspect beans are provided with a concrete context, and how aspect-oriented functionality can be enabled and disabled at runtime. We also investigate where this aspect-oriented functionality is best placed, and whether session, entity, and message-driven aspect beans can be useful.

Chapter 5 provides an overview of what aspect-oriented approaches are currently available for enterprise application development. We discuss JBossAOP [JBoa] — an aspect-oriented extension to the JBoss J2EE application server — and two frameworks that claim to be valid alternatives to J2EE based approaches: JAC [PDF+02] and Spring [Spr].

Chapter 6 states the conclusions of this dissertation. It summarizes our contributions, evaluates the applicability of our approach, and discusses possible future work.

Chapter 2 Research Context

2.1 Component-Based Software Development

2.1.1 Definition

As described by Clemens Szyperski [Szy97]:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Component-Based Software Development (CBSD) is a relatively new software development technique that focuses on reusing existing software components to build large applications. Based on the idea that a number of design patterns recur in many applications [GHJV94], CBSD arguments that software development should focus on combining existing components rather than writing new ones — an approach that may be summarized with the "write once, reuse everywhere" motto [WTM+].

Ideally, these components should be *black-box*: the developer should not know what the component does in order to provide its functionality. On the other hand, he should know exactly how to access the component, and what other components the component relies on in order to provide its functionality.

The advantages of CBSD are threefold:

- *Lower development costs;* as a component's development cost can be spread among many projects.
- *Shorter development time;* as developers should only assemble a number of components, instead of creating an entire project from scratch.
- *Increased development quality;* as defects in components will be discovered more quickly when they are used in many projects.

These advantages have already pushed several other engineering disciplines (such as the electronics or automobile industry) into the direction of component-based engineering; indeed, component-based engineering is a typical quality of any maturing engineering discipline.

CBSD shifts the development emphasis from programming software to composing software systems [Cle96]. Consequently, the process employed to create a component-based application is substantially different from existing software engineering processes. Four main activities characterize the component-based development approach [BW96]:

1. *Component qualification:* a number of *off-the-shelf* software components are tested in order to establish whether or not they are fit for use in the project at hand.

2. *Component adaptation:* components that do not fit into the project directly may need to be modified somewhat before they can be used.
3. *Component assembly:* the components are assembled into a complete system.
4. *System evolution:* the system is updated as bugs are discovered and software requirements change.

The developments in CBSD have given rise to a number of component models. A component model could be defined as follows [HC01]:

A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment. A component model also defines standards for an associated component model implementation, the dedicated set of executable software entities required to support the execution of components that conform to the model.

Examples of well known component models are the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [Cor04], Microsoft's Component Object Model (COM) [COM95], and Sun Microsystems' JavaBeans [Ham97] and Enterprise JavaBeans (EJB) [DYK01] models. The two former models will not be elaborated as they fall beyond the scope of this dissertation; the two latter models will be discussed in more detail below.

2.1.2 The JavaBeans Component Model

The JavaBeans component model [Ham97] is the simplest of Sun Microsystems' component models. Developed during the mid 1990s, its main goal was to define a component model for Java that enables third party Independent Software Vendors (ISVs) to create and ship Java components that can be composed together into applications by end users.

Since they were introduced, JavaBeans have become very popular, especially for the design of simple user interface applications. This is mostly due to the simplicity of the model and the ease-of-use of JavaBeans development tools such as Sun's Bean Builder [Bea]. For large applications, however, the JavaBeans component model is considered far too lightweight by most software developers.

The three most important features of a JavaBean are [Ham97]:

- *Properties* are named attributes associated with a bean that can be read or written by calling the bean's appropriate *getter* or *setter* method, respectively.
- *Methods* are normal Java methods which can be called by other components.
- *Events* provide a way for one component to notify other components that something interesting has happened: an *event listener* can be registered with an *event source*, which will notify all of its listeners when it detects that something interesting has happened. This is actually an implementation of the Observer pattern [GHJV94].

By composing a number of these JavaBeans in a graphical, plug-and-play fashion, a JavaBean developer can quickly and easily create simple component-based

applications. Figure 1 provides a screenshot of the design of such a simple application in Sun's Bean Builder:

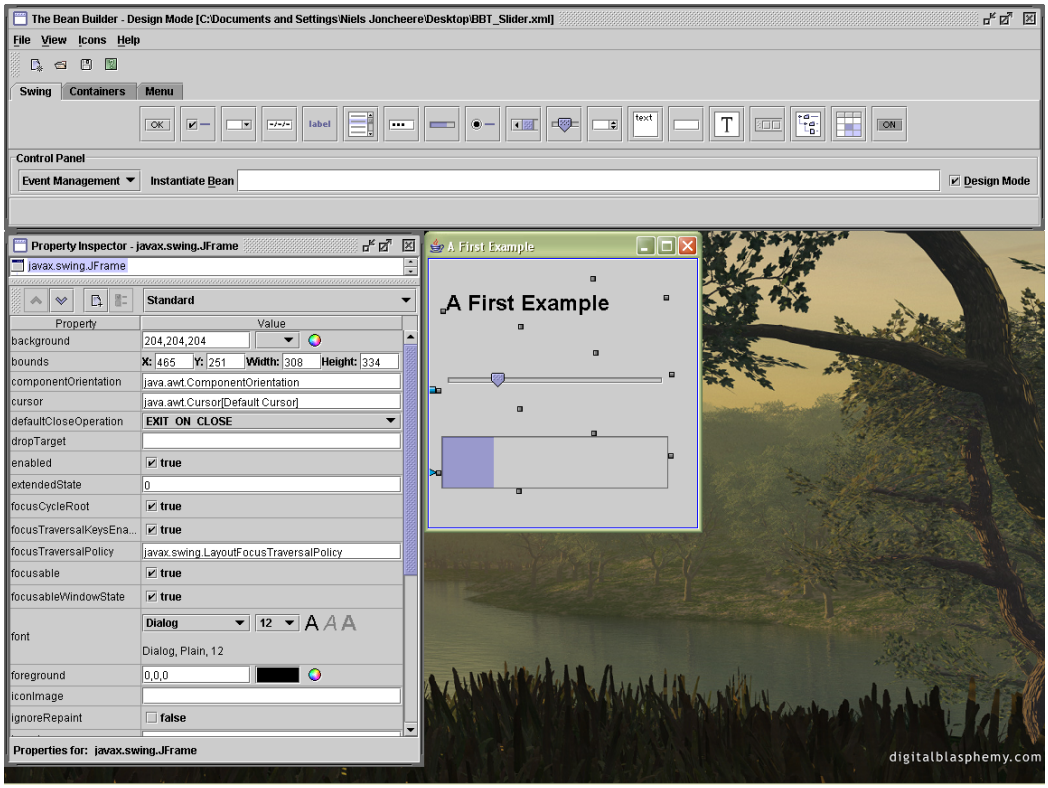

Figure 1: Designing a Simple Application in Sun's Bean Builder

2.1.3 The Java 2 Platform, Enterprise Edition

Overview

Sun Microsystems introduces the Java 2 Platform, Enterprise Edition (J2EE) as follows [J2E]:

The Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multi-tier enterprise applications. The J2EE platform simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behaviour automatically, without complex programming.

Today, J2EE is one of the most widely used platforms for enterprise applications in existence. The J2EE 1.4 specification [Sha03] defines three important entities:

- *JavaServer Pages* (JSPs) are HTML pages that contain pieces of Java code. This code is executed on the server when the page is requested, enabling the developer to create a dynamic website. JSP is very similar to other server-side scripting languages such as ASP(.NET) [ASP] and PHP [PHP]. Similar to these languages, JSP suffers from the "spaghetti code" problem: the source code is interleaved with the HTML code, making the pages difficult to read and expensive to maintain.
- *Java Servlets* provide a solution to the above problem. A servlet is a Java class that can be accessed through a URL. When it is accessed, this class has access to all the information in the corresponding HTTP request; it can

then handle the user's request itself or forward the request to another class. A JSP page is typically used to generate the response page, increasing the separation between functionality and layout. Servlets are very similar to CGI programs or .NET classes that implement the HttpHandler interface.

- The *Enterprise JavaBeans* (EJB) specification defines J2EE's server-side component architecture, and will be elaborated below.

Figure 2 illustrates how each of these entities fits into the J2EE enterprise application model [J2E]:

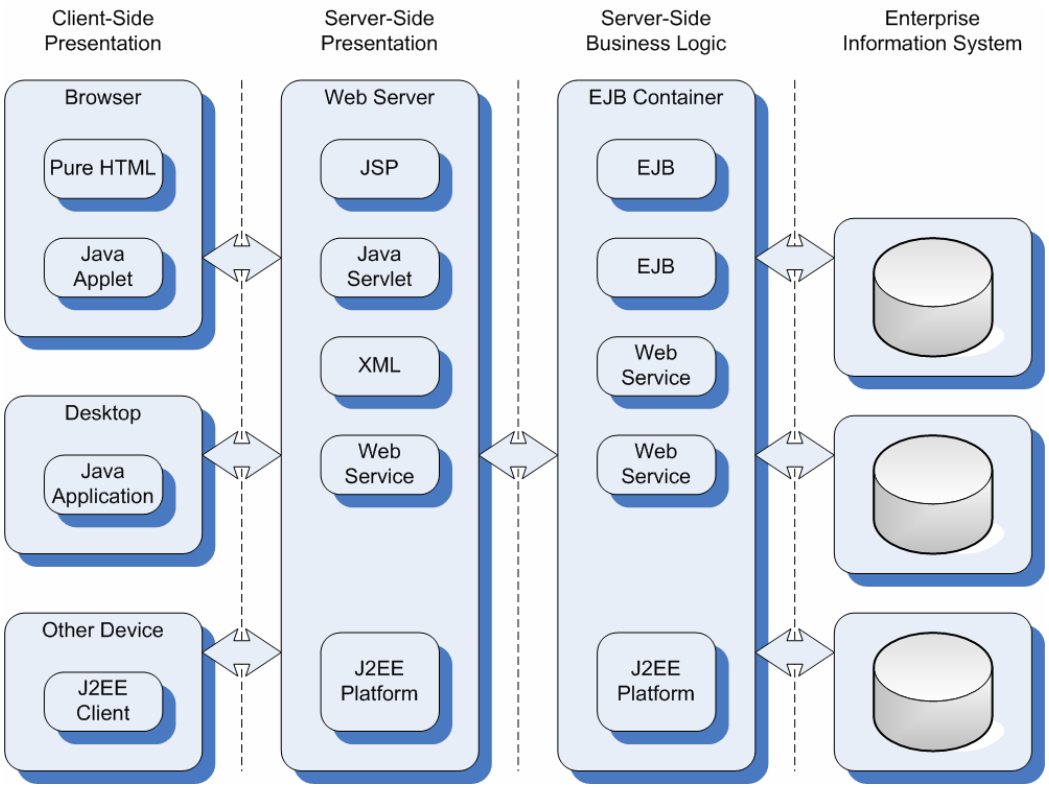

Figure 2: The J2EE Enterprise Application Model

Before it can be executed on a server, an entity's class files must be packaged in a Java Archive (JAR), together with a *deployment descriptor* — an XML file that describes how the entity should be *deployed* on the server. Among others, a deployment descriptor describes the entity's security policies.

J2EE entities are deployed on a J2EE *application server* (e.g. BEA WebLogic [Weba] or IBM WebSphere [Webb]). The application server wraps each entity in a *container* that is responsible for providing many of the platform's features, such as security, transaction management, etc.

Just as all of the other Java standards, J2EE is based on Sun's trademarked "write once, run anywhere" portability philosophy. In order to accomplish this, Java source code is compiled into platform independent *byte-code* — a stack based intermediate language that is interpreted by a Java Runtime Environment (JRE). This JRE has been ported to nearly every operating system imaginable.

The EJB Component Model

The EJB 2.0 specification [DYK01] defines three kinds of enterprise beans:

- *Session beans* represent behaviour associated with a client session; after the session is finished, the bean may be discarded. There are two kinds of session beans: stateless and stateful ones. Stateless session beans do not maintain a state between method invocations, making them ideal for grouping related methods in one place. Stateful session beans do maintain a state; a typical application of such a bean is a shopping cart in an online shopping application.
- *Entity beans* represent business objects that persist after a client session ends. Often, there is a one-to-one correspondence between an application's entity beans and the records in its relational database. Just like a database record, an entity bean can be uniquely identified by its primary key. A typical application of an entity bean is representing a customer of an online shopping application.
- *Message-driven beans* allow J2EE applications to process messages asynchronously. Normally, clients will invoke a bean method and then wait for a result. A message-driven bean can solve this problem by allowing a client to use the Java Messaging Service (JMS) to publish a request in the bean's queue. When the bean is ready to process the request, it retrieves the request from the queue, processes it and returns it to the client.

Typically, any of these beans will use one or more other beans to provide its functionality: by assembling several EJBs, one can create large, distributed, component-based applications.

Each of the above enterprise beans contains five parts:

- The *home interface* is used by a remote client to create or discard the bean.
- The *local home interface* is used by a local client to create or discard the bean; this interface was introduced into the EJB 2.0 specification to allow local clients to avoid the home interface and its network overhead.
- The *remote interface* is used by a remote client to invoke the bean's methods.
- The *local remote interface* is used by a local client to invoke the bean's methods; this interface was introduced into the EJB 2.0 specification to allow local clients to avoid the remote interface and its network overhead.
- The *implementation class* is where the bean's methods are implemented. The client cannot invoke these methods directly.

Before an EJB can be deployed, its home interface(s), remote interface(s), and implementation class need to be packaged in a JAR, together with a deployment descriptor that defines the bean's security policies, transaction policies, data source connections, etc.

Figure 3 illustrates the EJB component model by providing an example of a simple EJB application. A J2EE client uses EJB A's home and remote interfaces to create and use EJB A, respectively. The user cannot access EJB A directly: all requests need to pass through the EJB container that wraps the EJB, allowing this container to enforce, among others, security and transaction policies. EJB A, on its turn, will use EJB B's local home and local remote interfaces to create and use EJB B. Here, too, all requests need to pass through the EJB container:

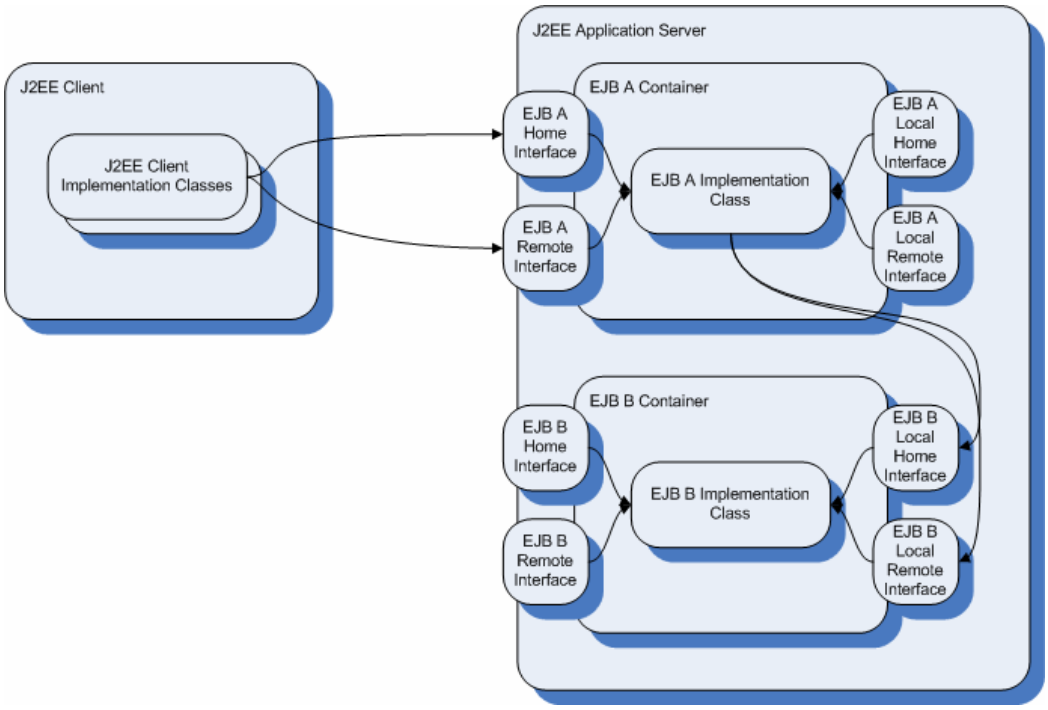

Figure 3: A Simple Enterprise JavaBeans Application

We will conclude our discussion of the EJB component model with a brief overview of the main differences between the JavaBeans and EJB component models:

- The JavaBeans component model is tailored for implementing simple, general-purpose components, while the EJB component model is tailored for implementing business objects and business logic.
- JavaBeans are typically executed on a client, while EJBs are always executed on a J2EE application server.
- J2EE provides EJB developers with powerful container services (such as security, transaction management, etc.), which are not available to JavaBeans developers.

2.2 Aspect-Oriented Software Development

2.2.1 Definition

Separation of concerns has been the holy grail of computer science for decades. Back in the early days, David L. Parnas associated the following benefits with it [Par72]:

(1) Managerial — development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility — it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility — it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

Indeed, throughout the history of computer science, these benefits have often caused programmers to migrate from simpler languages to languages that allowed them to decompose their problems more accurately.

Today, Object-Oriented Programming (OOP) is considered an effective instrument for implementing large applications. However, OOP suffers from the problems caused by the "tyranny of the dominant decomposition" [OT99]: the program can only be modularized in one way at a time, and concerns that do not align with this modularization end up scattered across many modules and tangled with one another.

Aspect-Oriented Programming (AOP) is a relatively new programming paradigm that originated around 1997, and aims at providing a solution to the above problem. Gregor Kiczales et al. [KLM+97] recognized that some concerns of an application cannot be modularized using current software development methods: the implementation of these concerns is completely spread out over the different modules of the system. The same logic is repeated among different modules, resulting in code duplication. This code duplication makes it very difficult to add, modify or remove these concerns [EFB01].

The goal of Aspect-Oriented Software Development (AOSD) is to achieve a better separation between these concerns. One should be able to describe *crosscutting* concerns in separate modules — *aspects* — so that adding, modifying, or removing these concerns does not require changes to the rest of the system. Typical examples of such aspects are logging and synchronization.

Typically, AOP allows a programmer to intercept method invocations and add functionality around them. In such a situation, the implementation of the logging concern could be extremely simplified by replacing it with an aspect that logs something before and after each method invocation. Of course, such an aspect should be added to the rest of the application at some time in order to be useful. This process is called *weaving*.

At the moment, these are the main approaches used to implement AOP:

- Compile-time approaches
  - *Source code weaving* — one could extend an existing programming language with special keywords that allow the programmer to express crosscutting concerns, and write a custom compiler that weaves these crosscutting concerns into the main program's source code.
  - *Byte-code weaving* — one could extend an existing programming language with special keywords that allow the programmer to express crosscutting concerns, and write a custom compiler that weaves these crosscutting concerns into the main program's byte-code.
- Runtime approaches
  - *Dynamic proxies* — dynamic proxies are a relatively new feature of J2SE that make it possible to add interfaces to an object at runtime. One could use this feature to insert "interceptors" between a method call and the corresponding method execution, allowing crosscutting behaviour to be executed at that point in the program execution.
  - *Byte-code manipulation* — one could enable the programmer to add aspect-oriented functionality to classes by modifying these classes' byte-code at runtime.

- *Debugging interface* — one could use the Java Virtual Machine Debugging Interface (JVMDI) to intercept method calls and add extra behaviour at that point in the program execution.

Almost every aspect-oriented language available today uses one or more of the above approaches in order to implement its functionality.

2.2.2 AspectJ

AspectJ [KHH+01] was developed during the late nineties at Xerox' Palo Alto Research Center (PARC). Nowadays, it is probably the most widely used aspect-oriented programming language on the market.

AspectJ extends Java with two kinds of crosscutting mechanisms:

- *Static crosscutting* enables the programmer to define new operations on existing types. Using static crosscutting, one can add interfaces, superclasses, methods, or fields to a class.
- *Dynamic crosscutting* enables the programmer to interfere at a number of well-defined points in the program execution.

In this section, we will concentrate our discussion on dynamic crosscutting; in order to implement this functionality, AspectJ introduces four concepts:

- A *join point* is a well-defined point in the execution of a program (e.g. the invocation of a certain method).
- A *pointcut* describes a collection of join points (e.g. the invocation of any of the methods of a certain class).
- An *advice* is attached to a pointcut and implements part of an aspect's behaviour (e.g. a *before* advice implements the behaviour that should be executed before the program enters a join point that belongs to the pointcut associated with the advice).
- *Aspects* encapsulate a number of pointcuts, and their associated advices.

A simple logging aspect, which will print a message every time any method of the Test class is executed, is given by Code Example 1:

```
public aspect LoggingAspect {

   pointcut exec():
      execution(* Test.*(..));

   before(): exec() {

      System.out.println("Method "
         + thisJoinPointStaticPart.getSignature()
         + " was invoked.");
   }
}
```

Code Example 1: A Simple Logging Aspect in AspectJ

An AspectJ program consists of two parts: (1) a number of 'regular' Java source files, implementing the program's main functionality, and (2) a number of AspectJ source files, implementing the program's crosscutting behaviour (i.e. its aspects). AspectJ will weave these aspects into the main program using a custom compiler. This

compiler employs a technique called *source code weaving*: it just inserts the program's aspects into the source code of the main program.

The fact that the AspectJ compiler employs source code weaving means that one can only add aspects to programs for which the source code is readily available. As this is often not the case, this was long considered one of AspectJ's main disadvantages. Recently, however, an updated version of the compiler has been proposed that employs byte-code weaving [HH04]. Another disadvantage of AspectJ is that it does not allow changes to the program's aspect-oriented functionality at runtime (e.g. disabling an aspect).

In 2002, PARC transferred AspectJ to the Eclipse project [Ecl] — an open-source development effort initiated by IBM, aimed to implement an open, extensible Integrated Development Environment (IDE).

2.3 Combining CBSD and AOSD

As described by Davy Suvée et al. [SVJ03]:

Currently available AOSD research mainly focuses on Object-Oriented Software Development (OOSD). CBSD, however, also suffers from the problems that arise with the tyranny of the dominant decomposition [OT99]. Similar to OOSD, aspects such as synchronization and logging are encountered, which crosscut several components from which the system is composed. Consequently, the ideas behind AOSD should also be integrated into CBSD. The other way around, namely the integration of CBSD within AOSD, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components; for example, it should be possible to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, would make aspects reusable and their deployment easy and flexible.

Indeed, combining CBSD and AOSD could prove interesting for both of these paradigms. Most currently available aspect-oriented approaches, however, are not very suitable for CBSD [SVJ03]:

- The deployment of aspects within a software system is at the moment rather static. In AspectJ, for instance, an aspect loses its identity at compile time, as it is woven into the main program's source code. In a component-based situation, an aspect should have more or less the same plug-and-play characteristics as components, allowing them to be connected and disconnected to components quickly and easily.
- Most aspect-oriented approaches describe their aspects with a specific context in mind. This makes it impossible to reuse these aspects. In a component-based situation, an aspect should not have any references to where it will be applied.

Therefore, the Vrije Universiteit Brussel (VUB)'s System and Software Development Lab developed an aspect-oriented approach that is specifically tailored for CBSD. This approach is called *JAsCo* [SVJ03].

2.3.1 JAsCo

In the late 1990s, Karl Lieberherr et al. [LLM99] described a problem with most aspect-oriented approaches: although aspects were described separately, they lost their

identity at compile time, as they were woven into the main program's source code. They proposed a solution to this problem, called *Aspectual Components*, which made sure that aspects were kept separate from the program's main behaviour.

JAsCo is based on roughly the same principle; Davy Suvée [Suv02] noticed that most aspect-oriented approaches required access to the main program's source code. In a component-based situation, however, where black-box, off-the-shelf components are being assembled into large applications, this source code is often not available. Hence, it should be possible to attach some kind of *aspect-oriented components* to certain points in the execution of a component. JAsCo — an acronym for "Java Aspect-Oriented Components" — provides a solution to this problem.

JAsCo's main characteristics are the following [Suv02]:

- The language is an extension to Java that stays as close to the original syntax as possible [GJSB00].
- The language is a general-purpose aspect-oriented language; it is not a domain-specific aspect-oriented language.
- One can attach aspects to black-box components; indeed, the language does not require access to the main program's source code.
- An aspect remains a separate entity at all times; the aspects' behaviour isn't simply copied into the components to which they are applied.
- Aspects can interact with one another.
- When multiple aspects are applicable at the same point in the program execution, it is possible to specify the order in which these aspects should be applied.
- It is possible to add aspects to or remove aspects from components in a plug-and-play fashion, similar to the way components are attached to each other.
- It is possible to add aspects to or remove aspects from components at runtime.

JAsCo introduces two concepts:

- *Aspect beans* define the crosscutting behaviour that has been extracted from the components. An aspect bean contains at least one *hook* that defines when the execution of a method should be interrupted, and what extra behaviour should be executed at that time. Aspect beans are completely contextless: they do not have any reference to where they will be applied.
- *Connectors* are used to connect components with aspect beans: they provide aspect beans with a concrete context by defining which methods should be connected with which hooks.

Figure 4 provides a graphical overview of JAsCo [Van04]:

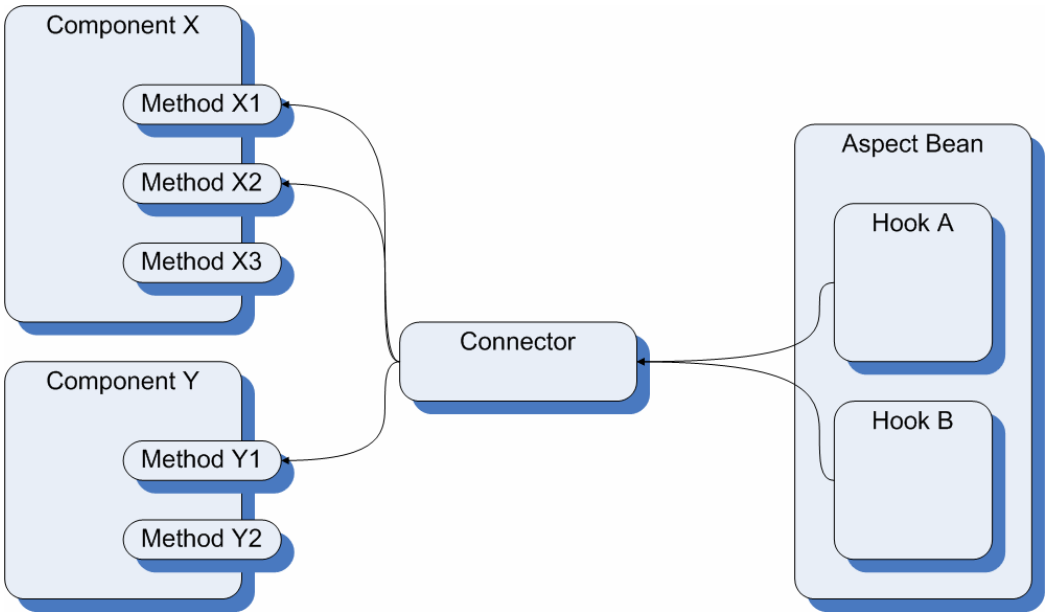

Figure 4: A Graphical Overview of JAsCo

A simple logging aspect and connector are given by Code Example 2 and Code Example 3, respectively. Together, they will make sure a message is printed each time any method of the Test class is invoked.

```
class LoggingAspect {

   hook exec {

      exec(method(..args)) {
         execute(method);
      }

      before() {

         System.out.println("Method "
            + calledmethod
            + " was invoked.");
      }
   }
}
```

Code Example 2: A Simple Logging Aspect in JAsCo

```
static connector LoggingConnector {

   LoggingAspect.exec exec = new LoggingAspect.exec(* Test.*(*));
   exec.before();
}
```

Code Example 3: A Simple JAsCo Connector

In order to make the JAsCo language operational, its authors introduce a new, "aspect-enabled" component model [Van04]:

The JAsCo Beans component model is a backward compatible extension of the Java Beans component model where traps are already built-in. These traps are used to attach and detach aspects. As a result, JAsCo beans do not require any adaptation whatsoever to be subject to aspect application. The JAsCo component model enables run-time aspect addition and removal. In addition, aspects remain first class entities at run-time as they are not weaved and spread into the target components.

In order to add this kind of functionality to a Java bean, it has to be *transformed* into a JAsCo bean. This transformation process modifies the Java bean's byte-code so that, when a method of the transformed bean is called, this method will look up all appropriate connectors in the *connector registry*, and invoke the corresponding behaviour [Suv02].

Chapter 3 JAsCo2EE: Straightforward Approach

3.1 Motivation

As we discussed in Chapter 2, the JAsCo AOP language boasts several features that greatly facilitate the combination of CBSD and AOSD. The current implementation of JAsCo is based on the JavaBeans component model; this design choice made perfect sense when the language was being developed: as the JavaBeans component model is both simple and widely used within the software engineering research community, it is ideal for research experiments [Van04].

In real-life applications, however, the JavaBeans component model is often discarded in favour of J2EE's EJB component model. Although J2EE already provides applications with several services — such as security and transaction management — that are in fact crosscutting concerns, J2EE does not allow application developers to describe general-purpose aspects. Hence, if we could integrate J2EE and JAsCo, which is an AOP language specifically tailored for expressing such general-purpose aspects in a component based context, separation of concerns in J2EE applications could be greatly improved. Indeed, integrating J2EE and JAsCo could prove very useful to J2EE.

On the other hand, integrating J2EE and JAsCo could provide interesting benefits to JAsCo, too: as J2EE is the major platform for the development of large commercial applications, JAsCo would be much more useful in real-life applications if it could be applied to J2EE. In this chapter, we will describe a first approach that illustrates how this new, improved version of JAsCo — which we will call "JAsCo2EE" — was implemented.

3.2 Approach

Our first approach is based on the idea that J2EE applications are in a lot of ways very similar to the "regular" J2SE applications that JAsCo works with; one might even consider J2EE to be J2SE extended with a number of special, enterprise-oriented APIs. Just as regular Java code, J2EE code is compiled into Java byte-code, and as JAsCo uses byte-code manipulation to weave its aspects into Java programs, it should be possible to use the existing implementation of JAsCo to add aspect-oriented functionality to J2EE with only minor modifications.

Hence, the goal of our first approach is to implement a version of JAsCo2EE in the most straightforward way possible, taking into account the particularities of J2EE as little as possible. While this approach might not be the most conceptually sound one — perhaps some functionality implemented in JAsCo is already available in J2EE in some form — it is bound to give us some valuable insight into the technicalities of extending J2EE with JAsCo functionality and can thus be an important stepping stone towards an improved version of JAsCo2EE.

3.3 Implementation

In order to compile a typical JAsCo application, consisting of a number of Java beans, a number of aspect beans, and a number of connectors, the following actions need to be performed:

1. Compile the aspect beans.
2. Compile the connectors.
3. Transform the Java beans into JAsCo beans.

When one executes such an application after these three steps have been performed, JAsCo is started automatically and the crosscutting functionality described in the application's aspect beans is added to the Java beans at the places described in the application's connectors.

Hence, one would expect that a typical JAsCo2EE application, consisting of a number of EJBs, a number of aspect beans, and a number of connectors, could be compiled as follows:

1. Compile the aspect beans.
2. Compile the connectors.
3. Transform the Enterprise JavaBeans into "Enterprise JAsCoBeans".

When one deploys such an application on a J2EE application server after these three steps have been performed, JAsCo is started automatically. The crosscutting functionality, however, is not executed, because JAsCo does not find the connectors that describe where the crosscutting functionality described in the aspect beans needs to be added.

The reason for this problem is that, in order to allow enabling and disabling connectors at runtime, JAsCo searches for JAsCo connectors in the Java classpath every five seconds. This makes it possible to add connectors to an application by simply copying them into the classpath, and remove them from the application by deleting them from the classpath.

In J2EE, however, each EJB — together with its associated connectors — lives in a world of its own, which is not included in the classpath. Hence, JAsCo cannot find these connectors, making it impossible to load them and invoke the aspect-oriented functionality declared by them. A solution to this problem is essential to implementing a working version of JAsCo2EE.

Our solution to this problem makes the EJBs themselves responsible for explicitly registering their connectors with JAsCo, so that JAsCo doesn't need to search for them any more. This can be accomplished by requiring application developers to declare the names of their applications' JAsCo connectors in these applications' deployment descriptors.

As the EJBs don't know what to do with the names of these connectors by themselves, we also need to modify the part of JAsCo that transforms the EJBs. More specifically, we need to make sure that, when a transformed EJB is being deployed, it goes looking for the names of its connectors in its deployment descriptor, and asks JAsCo to register these connectors. Finally, we need to modify the runtime part of JAsCo to allow EJBs to explicitly register their connectors like this.

The main comment one might make with regard to the solution we described above is that it makes it difficult, or even impossible to enable or disable an EJB's aspect-oriented functionality at runtime, and that this constitutes a noteworthy disadvantage when compared to the original version of JAsCo. Such comments, however, are somewhat exaggerated: one could easily devise a scheme in which one would extend a J2EE application server with an API that allows access to the server's JAsCo environment, and allows the application developer to add or remove connectors through a utility that uses this API.

Furthermore, it would be very counterintuitive from a J2EE perspective if one could add connectors to an EJB at runtime by merely copying them into the classpath; analogous to EJBs, which need to be deployed explicitly, connectors should only be deployed explicitly, too.

3.4 Usage

Using the implementation we described above, these are the steps one must perform in order to create a JAsCo2EE application and deploy it on a J2EE application server:

1. Write and compile a number of EJBs, just as one would do in order to create a regular J2EE application. These EJBs implement the application's main functionality.
2. Write and compile some JAsCo aspect beans, just as one would do in order to create a regular JAsCo application. These aspect beans implement the application's crosscutting functionality.
3. Write and compile some JAsCo connectors, just as one would do in order to create a regular JAsCo application. These connectors provide the aspect beans with a concrete context, i.e. they link the aspect beans to a number of the EJBs' methods.
4. Put the names of the connectors as an "environment entry" in the application's deployment descriptor. This allows the EJBs to explicitly register their connectors with JAsCo.
5. Transform the EJBs using our modified version of JAsCo. This makes sure the transformed EJBs read in the names of their connectors from the deployment descriptor and register these with JAsCo.
6. Package the transformed EJBs, the aspect beans, the connectors, and the deployment descriptor in a JAR, much like one would package a regular J2EE application.
7. Generate the EJBs' "stubs" and "skeletons" by using a utility provided by the application server on which the application will be deployed, just like one would do in order to create a regular J2EE application.
8. Deploy the application on the application server, just like one would do with a regular J2EE application.

It is worth noting that only one of these steps — step 4 — is different from the ones one would use to create a regular J2EE or JAsCo application. Hence, application developers familiar with these two technologies will not have much trouble with JAsCo2EE.

Once the JAsCo2EE application has been deployed on the J2EE application server, the following things happen:

1. When an EJB is created by its container, the functionality added by the JAsCo transformation process reads the names of the EJB's connectors from its deployment descriptor and asks JAsCo to load them.
2. JAsCo loads the connectors, adding them to the connector registry.
3. Every time a method is invoked, JAsCo looks into its connector registry for connectors that match the current method. If this is the case, the corresponding aspects' behaviour is invoked.

Note that JAsCo's class files must be available from within the EJB if these steps are to succeed; one could, for example, package JAsCo's class files in a JAR file and place this file in the application server's external library directory.

3.5 Conclusion

We may conclude that our first approach has accomplished its goal: it provides a simple, straightforward implementation of JAsCo2EE. There are, however, some things left to be desired:

As described by Davy Suvée et al. [SVJ03]:

The JAsCo language introduces two concepts: aspect beans and connectors. An aspect bean describes behaviour that interferes with the execution of a component by using a special kind of inner class, called a hook. The specification of a hook is context independent and therefore reusable. A connector on the other hand, is used for deploying one or more hooks within a specific context.

Hence, a JAsCo program can be divided into three main parts: (1) a 'regular' Java program to which the programmer wishes to add some aspect-oriented functionality, (2) one or more aspect beans that implement this functionality, and (3) one or more connectors that link the Java program to its aspect beans.

A J2EE program, on the other hand, can be divided into two main parts: (1) the program's implementation, i.e. one or more Enterprise JavaBeans, and (2) a number of deployment descriptors that describe how each of these Enterprise JavaBeans should be deployed.

By combining J2EE and JAsCo in the simplest way possible, our first approach has given rise to four main parts in each JAsCo2EE program: (1) one or more Enterprise JavaBeans to which the programmer wishes to add some aspect-oriented functionality, (2) a number of deployment descriptors that describe how each of these Enterprise JavaBeans should be deployed, (3) one or more aspect beans that implement the JAsCo2EE program's aspect-oriented functionality, and (4) one or more connectors that link the Enterprise JavaBeans to their aspect beans.

Figure 5 illustrates the modified EJB component model achieved through our first approach. It provides a simple EJB application — the one from Figure 3 — to which we have added some aspect-oriented functionality: EJB A has been linked to Aspect Bean X through the use of Connector K, and EJB B has been linked to Aspect Bean X and Aspect Bean Y through the use of Connector L:

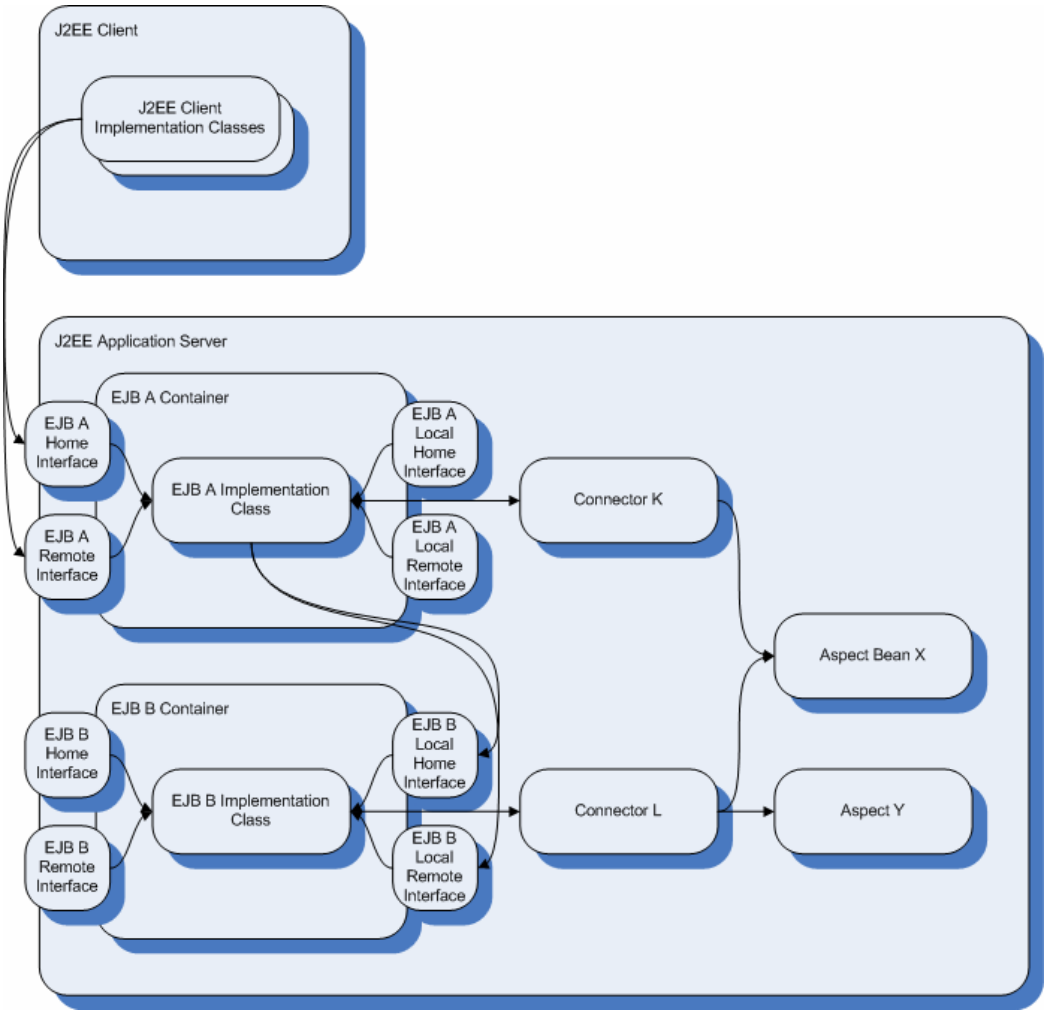

Figure 5: A Simple Enterprise JAsCoBeans Application

This is obviously not an optimal solution to the problem of integrating J2EE and JAsCo, as the deployment descriptors and the connectors perform more or less the same function: a deployment descriptor links an Enterprise JavaBean to its resources or to other Enterprise JavaBeans, while a connector links an Enterprise JavaBean to its aspect beans. Hence, JAsCo2EE would fit much better into the J2EE philosophy if the connectors' functionality could be implemented using the deployment descriptors.

As the EJB containers already implement a number of crosscutting concerns, it could also be useful to weave the crosscutting concerns described by the aspect beans into the EJB containers, instead of the EJB implementation classes.

These and other ideas will be explored further in Chapter 4.

Chapter 4 JAsCo2EE: Full Integration

4.1 Motivation

As we introduced in Chapter 3, the implementation obtained through our straightforward approach leaves a few things to be desired: JAsCo has not been completely integrated with J2EE, resulting in an unnecessarily cluttered component model, and reduced functionality.

4.2 Approach

We propose a new component model, based on the Enterprise JavaBeans component model, which will make J2EE "aspect-aware". Our new "Enterprise JAsCoBeans" model is a backward compatible extension to the EJB model, and has the following features:

- It introduces a new kind of entity — which we will call aspect beans — into the EJB model that enables application developers to describe crosscutting concerns separate from the EJBs to which they need to be applied.
- EJBs' implementation classes have built-in traps that are used to attach and detach aspects. Enterprise JAsCoBeans do not require any adaptation to be subject to aspect application [Van04].

Note that our JAsCo2EE aspect beans have the same characteristics as "regular" JAsCo aspect beans [Suv02]:

- One can attach aspects to black-box components; indeed, the language does not require access to the main program's source code.
- An aspect remains a separate entity at all times; the aspects' behaviour isn't simply copied into the components to which they are applied.
- Aspects can interact with one another.
- When multiple aspects are applicable at the same point in the program execution, it is possible to specify the order in which these aspects should be applied.
- It is possible to add aspects to or remove aspects from components in a plug-and-play fashion, similar to the way components are attached to each other.
- It is possible to add aspects to or remove aspects from components at runtime.

Figure 6 illustrates the new Enterprise JAsCoBeans component model. It provides a simple EJB application — the one from Figure 3 — to which we have added some aspect-oriented functionality: EJB A has been linked to Aspect Bean X, and EJB B has been linked to Aspect Bean X and Aspect Bean Y. Comparing Figure 6 to Figure

5, one can clearly see that our new component model is less complicated than the one achieved through our first approach:

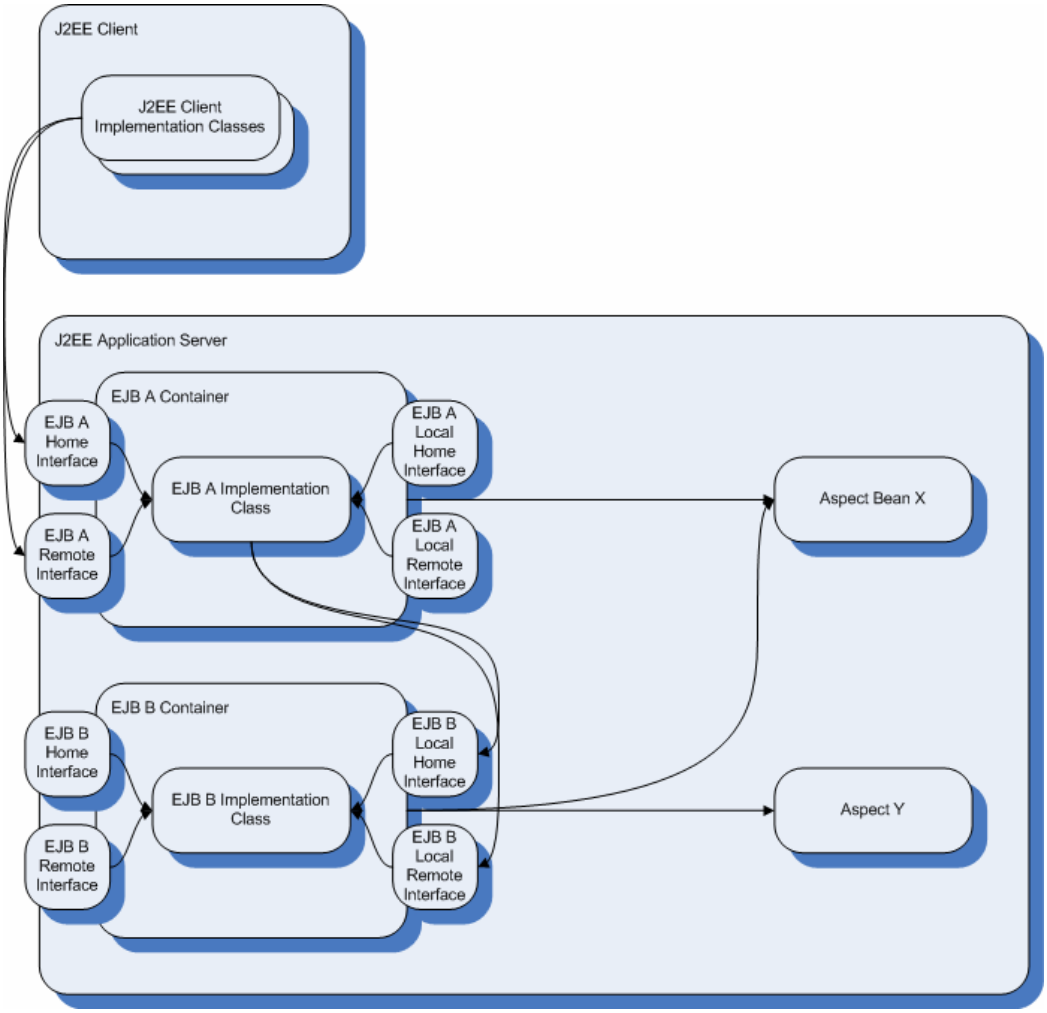

Figure 6: A Simple Enterprise JAsCoBeans Application

In order to support all of the characteristics of JAsCo2EE aspect beans we described above, we need to investigate the following topics:

- How will these aspect beans, which have no reference to where they will be applied, be provided with a concrete context?
- How will we allow aspects to be enabled and disabled at runtime?

Additionally, one may ask the following questions regarding the aspect beans themselves:

- Where will their behaviour be added to the EJBs: in the EJBs themselves, or in the EJB containers?
- EJBs come in three flavours: session beans, entity beans, and message-driven beans. Does this mean there will be an analogue distinction between aspect beans, too?

We will now discuss each of these topics in more detail.

4.2.1 Providing Aspect Beans with a Concrete Context

EJB deployment descriptors are text-based XML files that describe how an EJB module — a JAR containing one or more EJBs — is to be deployed on a J2EE application server. For each of its module's EJBs, a deployment descriptor declares — among others — the name, the home and remote interfaces, the implementation class, the initialisation parameters, the references to resources, and the references to other EJBs. It also declares the relationships among its module's EJBs, and defines the EJBs' security and transaction policies.

JAsCo connectors are Java source files with a special syntax. They provide aspect beans with a concrete context by linking JAsCo beans to them. A connector's responsibilities are:

- Declaring and instantiating a number of hooks.
- Defining the precedence among hook instantiations.
- Implementing abstract hook methods.
- Initialising hooks.
- Defining combination strategies.

Indeed, a deployment descriptor links an Enterprise JavaBean to its resources or to other Enterprise JavaBeans, while a connector links a JAsCo bean to its aspect beans. Hence, it comes very natural to wonder whether the responsibilities of a JAsCo application's connectors couldn't be expressed in an Enterprise JAsCoBeans application's deployment descriptor. Thus, one would achieve a JAsCo2EE component model that is much more streamlined than the one we achieved in Chapter 3. This is why our new component model does not contain any connectors: a JAsCo2EE application only consists of:

- A number of EJBs, which implement the application's main functionality.
- A number of aspect beans, which implement the application's aspect-oriented fnctionality.
- An extended deployment descriptor, which has all the responsibilities of a regular EJB deployment descriptor, but also describes the links between the application's EJBs and aspect beans.

Thus, JAsCo connectors are completely eliminated from the JAsCo2EE picture.

One of the main characteristics of our new component model is that it integrates the JAsCo connectors' responsibilities into the EJB deployment descriptors when this is appropriate, taking into account the J2EE philosophy as much as possible. Hence, we will need to carefully consider which of the JAsCo connectors' responsibilities actually belong in the Enterprise JAsCoBeans deployment descriptors. This is illustrated by Figure 7; a responsibility that belongs in the given entity is marked with a '+', a responsibility that belongs in the given entity, but with some reservations, is marked with a '±', and a responsibility that doesn't belong in the given entity is marked with a '-':

	Connectors	Deployment Descriptors
Declaring and Instantiating a Number of Hooks	+	+
Defining the Precedence among Hook Instantiations	+	+
Implementing Abstract Hook Methods	+	-
Initialising Hooks	+	±
Defining Combination Strategies	+	+

Figure 7: Comparison of Connector and Deployment Descriptor Responsibilities

In the following paragraphs, we will discuss each of these responsibilities in more detail.

Declaring and Instantiating a Number of Hooks

The most important responsibility of a JAsCo connector is declaring and instantiating a number of hooks; Code Example 4 provides an example of a connector that declares one hook [Van04]. As the declaration of a reference to a hook is very similar to the declaration of a reference to a resource or an EJB, which can already be expressed in the EJB deployment descriptor, this responsibility clearly belongs in the EJB deployment descriptor, too:

```
static connector PrintAccessControl {

   AccessManager.AccessControl control
      = new AccessManager.AccessControl(* Printer.*(*));
}
```

Code Example 4: A JAsCo Connector that Declares and Instantiates a Hook

Defining the Precedence among Hook Instantiations

Another important responsibility of a JAsCo connector is defining the precedence among hook instantiations when more than one hook is applicable at the same point. In regular JAsCo, such a precedence strategy is specified by calling one or more *hook behaviour methods* in a connector. The order in which these methods are called in the connector specifies the order in which they will be applied by JAsCo. Code Example 5 shows an example of a connector that defines a precedence strategy for the three hooks it declares [Van04]:

```
static connector PrintAccessControl {

   AccessManager.AccessControl control
      = new AccessManager.AccessControl(* Printer.*(*));

   LockingManager.LockControl lock
      = new LockingManager.LockControl(* Printer.*(*));
```

```
   Logger.FileLogger logger
      = new Logger.FileLogger(* Printer.*(*));

   logger.before();
   lock.before();
   control.replace();
   lock.after();
   logger.after();
}
```

Code Example 5: A JAsCo Connector that Defines a Precedence Strategy

The net result of this precedence strategy will be that, when the control, lock, and logger hooks are applicable at the same moment (i.e. every time one of the methods of the Printer class is invoked), the logger hook's before method will be executed before the lock hook's before method, the lock hook's before method will be executed before the control hook's replace method, and so forth.

It seems natural that, if one would define references to hooks in the EJB deployment descriptor, the precedence among these hooks would be defined in the same deployment descriptor. Hence, this responsibility belongs in the EJB deployment descriptor, too.

Implementing Abstract Hook Methods

In order to improve separation of concerns and facilitate reuse of aspect beans, JAsCo allows programmers to define *abstract hook methods*; these methods are declared in a hook, but aren't implemented yet: the implementation for these methods needs to be provided in a JAsCo connector.

This presents a problem: a deployment descriptor is an XML file used to declare information needed at deployment time; it should not be used to write Java code. Hence, we will not integrate this part of a JAsCo connector's responsibilities in the EJB deployment descriptor. This does not constitute a noteworthy disadvantage, as one could easily write a subclass for an aspect bean that implements an abstract hook method, instead of implementing the method from within the deployment descriptor.

Initialising Hooks

JAsCo allows programmers to specify regular Java code in a connector. This makes it possible to initialise hook variables using JAsCo connectors; making aspect beans even more reusable. Code Example 7 provides an example of a connector that declares two discount hooks, and initialises each of these hooks with a discount value [Van04]:

```
static connector DiscountConnector {

   Discounts.BirthdayDiscount birthday
      = new Discounts.BirthdayDiscount(* Shop.checkout(*));

   Discounts.FrequentDiscount frequent
      = new Discounts.FrequentDiscount(* Shop.checkout(*));

   birthday.setDiscount(0.02);
   frequent.setDiscount(0.20);
}
```

Code Example 6: Initialising Hooks from within a JAsCo Connector

It would be interesting if we could integrate this initialisation functionality in the EJB deployment descriptor, too. As a deployment descriptor should not be used for writing Java code, however, we will need to put some restrictions on the kind of initialisations one may perform. For example, we may decide to allow only invocations of hook methods with constant arguments here. While this kind of restrictions would make an EJB deployment descriptor somewhat less powerful than a JAsCo connector, this makes sense from a conceptual point of view: connectors should only be used for instantiating hooks and specifying how they cooperate; other, more complicated logic should be modularized in EJBs and aspect beans [Van04].

Defining Combination Strategies

A last responsibility of JAsCo connectors is defining a number of combination strategies. Combination strategies are a second solution to the *feature interaction problem* [PSC+01] — the problem of deciding which aspects should be executed when multiple aspects are applicable, and in what order. They are based on the idea that the precedence strategies we discussed above are interesting, but insufficient: these precedence strategies can, for example, not express "if hook A and B are both applicable, only hook A should be executed".

JAsCo combination strategies provide a solution to this problem by acting as a filter on the applicable hooks. A combination strategy is any Java class that implements the CombinationStrategy interface:

```
package jasco.runtime.connector;

public interface CombinationStrategy
{
   public HookList validateCombinations(HookList hookList);
}
```

Code Example 7: The JAsCo CombinationStrategy Interface

When multiple hooks are applicable at the same point in the execution of the program, the combination strategy's validateCombinations method will be called with the list of applicable hooks as its argument. The method can then filter the hooks that will be applied.

Code Example 8 provides an example of a connector that defines a combination strategy:

```
static connector DiscountConnector {

   Discounts.BirthdayDiscount birthday
      = new Discounts.BirthdayDiscount(* Shop.checkout(*));

   Discounts.FrequentDiscount frequent
      = new Discounts.FrequentDiscount(* Shop.checkout(*));

   birthday.replace();
   frequent.replace();

   addCombinationStrategy(
      new ExcludeCombinationStrategy(birthday, frequent)
   );
}
```

Code Example 8: Adding a Combination Strategy to a JAsCo Connector

It is clear that combination strategies are one of JAsCo's most distinguishing features when compared to other aspect-oriented approaches. As combination strategies attempt to provide a solution to the same problem as precedence strategies, combination strategies belong in our new deployment descriptor, too.

A Proof of Concept implementation for this extension of the EJB deployment descriptor is provided in the following section.

4.2.2 Enabling and Disabling Crosscutting Functionality at Runtime

One of the features of JAsCo is that it allows aspects to be added to and removed from components at runtime. In the regular version of JAsCo, this is accomplished by providing the user with an introspector tool, which allows application developers to examine which connectors are currently loaded, and provides them with the possibility to enable or disable these connectors. Especially for debugging purposes, such a function is very useful.

Currently, many J2EE application servers provide a web-based management interface, which, for example, allows application developers to load or unload EJB applications, or view details regarding the application. Using our approach, it should be possible to view which crosscutting concerns are currently added to such an application, and disable or enable these concerns at runtime through the web-based management interface.

4.2.3 Weaving Crosscutting Functionality in the Container

JavaBeans are separate components that are directly accessible to other components: all public methods of a JavaBean can be used by any other JavaBean. In the EJB component model, however, the J2EE application server wraps each Enterprise JavaBean with a container. Such a container will only allow other components to create or delete EJBs through the EJB's home interface, or to invoke methods on EJBs through the EJB's remote interface: public methods of an EJB's implementation class that are not declared in the home or remote interface are not accessible from the outside.

In regular JAsCo, aspects are woven into the JavaBeans components themselves. In J2EE, however, we have a choice of where we could weave our aspects: in the Enterprise JavaBeans themselves, or in their containers.

Weaving crosscutting functionality in the container is possible for all methods declared in the EJB's home and remote interfaces, i.e. the methods that are accessible from the outside. When such a method is invoked, the container already intercepts the method to check whether any security or transaction policies defined in the deployment descriptor needs to be applied. Hence, it sounds reasonable to check whether any aspect-oriented functionality described in the deployment descriptor needs to be added there, too.

There is, however, a disadvantage of this approach: as a container cannot intercept all methods defined by its EJBs' implementation classes, but only those declared in the EJBs' home and remote interfaces, one cannot add aspects to all of the implementation classes' methods. An approach that weaves aspect-oriented functionality in the containers is thus less expressive than one that weaves aspect-oriented functionality in the implementation classes themselves, as one method invocation intercepted by the container might give rise to several method invocations on the implementation class that cannot be intercepted by the container.

Our approach will weave its aspects in the container, as this fits better into the J2EE philosophy. A Proof of Concept implementation for the method that weaves the aspects in the implementation classes is provided in the following section.

4.2.4 Session, Entity, and Message-Driven Aspect Beans

The EJB 2.0 specification defines three kinds of Enterprise JavaBeans: session beans, entity beans, and message-driven beans. There are important differences between these types, especially with regard to state and persistence:

- Session beans are discarded when a session is finished; stateful session beans maintain a state between method invocations, stateless beans do not.
- Entity beans persist after a session is finished.

One might wonder whether it would be useful to make a similar distinction among aspect beans, for example, one might define stateful session aspect beans that persist until the end of the corresponding bean's session, and maintain state between method invocations, and entity aspect beans that persist infinitely. Thus, there would be a one-on-one mapping between EJBs and aspect beans.

The alternative to this method is the one used by the original version of JAsCo: there is one aspect bean for all instances of a certain EJB's implementation class. Our approach uses the former method, a Proof of Concept implementation for the latter method is provided in the following section.

4.3 Implementation

Before we can implement a version of JAsCo2EE, we must define the syntax of the modified deployment descriptor we introduced above. In this section, we define that syntax, and give an overview of how a version of JAsCo2EE that implements the approach we described above could be implemented. Finally, we give a concrete example of a Proof of Concept implementation that implements a part of this approach: the extended deployment descriptor.

4.3.1 Modified Deployment Descriptor Syntax

Introduction

As a deployment descriptor is an XML file, its syntax is unambiguously defined by a Document Type Definition (DTD); the EJB 2.0 deployment descriptor's syntax is defined by the ejb-jar_2_0.dtd file that was released by Sun as a part of the EJB 2.0 Specification [DYK01]. We will need to extend this DTD in order to define the syntax of our new Enterprise JAsCoBean deployment descriptor.

Code Example 9 shows an overview of the main elements of the EJB 2.0 deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
   <enterprise-beans>
      <session>
```

```
            ...
         </session>
         <entity>
            ...
         </entity>
         <message-driven>
            ...
         </message-driven>
      </enterprise-beans>
      <relationships>
         ...
      </relationships>
      <assembly-descriptor>
         ...
      </assembly-descriptor>
</ejb-jar>
```

Code Example 9: Overview of Main EJB 2.0 Deployment Descriptor Elements

These are the main characteristics of these elements [EJB]:

- The ejb-jar element is the root element of the EJB deployment descriptor. It has three important children: the enterprise-bean element, the relationships element, and the assembly-descriptor element. Each of these will be discussed below.
- The enterprise-beans element contains the declaration of one or more enterprise beans. It has three possible types of children: session elements, entity elements, or message-driven elements, representing session, entity, or message-driven EJB declarations, respectively.
- The session, entity, and message-driven elements declare a session, entity, or message-driven EJB, respectively: they declare, among others, an EJB's name, its home and remote interfaces, its implementation class, its references to resources, and its references to other EJBs.
- The relationships element describes the relationships in which entity beans with container managed persistence participate.
- The assembly-descriptor element contains application assembly information: the definition of security roles, the definition of method permissions, the definition of transaction attributes for enterprise beans with container-managed transaction demarcation, and a list of methods to be excluded from deployment.

In the following paragraphs, we will introduce four new elements into the EJB deployment descriptor. Code Example 10 provides an overview of where each of these elements fits into the deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
   <enterprise-beans>
      <session>
         ...
         <hook-ref>...</hook-ref>
         <hook-init>...</hook-init>
      </session>
```

```
      <entity>
         ...
         <hook-ref>...</hook-ref>
         <hook-init>...</hook-init>
      </entity>
      <message-driven>
         ...
         <hook-ref>...</hook-ref>
         <hook-init>...</hook-init>
      </message-driven>
   </enterprise-beans>
   <relationships>
      ...
   </relationships>
   <assembly-descriptor>
      ...
      <hook-prec>...</hook-prec>
      <hook-comb>...</hook-comb>
   </assembly-descriptor>
</ejb-jar>
```

Code Example 10: Overview of New Deployment Descriptor Elements

We are now ready to decide where each of the four responsibilities of a JAsCo connector we selected above fits into this EJB deployment descriptor.

Declaring and Instantiating a Number of Hooks

Conceptually, the declaration of a reference to a hook is very similar to the declaration of a reference to a resource or the declaration of a reference to an EJB, which are already available in the EJB deployment descriptor's session, entity, and message-driven elements. All of these declarations declare a part of the application's functionality that needs to be supported by an underlying system, in this case by the J2EE application server or by JAsCo. Hence, it comes very natural to declare references to hooks in the session, entity, and message-driven elements.

Therefore, we will introduce a new child — which we will name hook-ref to preserve the analogy with references to resources (resource-ref) and EJBs (ejb-ref) — into each of these three elements. Code Example 11 provides an example of such a hook-ref element, based on the JAsCo connector from Code Example 4:

```
<hook-ref>
   <description>
      Links the EJB to the AccessManager.AccessControl hook.
   </description>
   <hook-ref-name>control</hook-ref-name>
   <hook-ref-type>AccessManager.AccessControl</hook-ref-type>
   <hook-ref-param>* Printer.*(*)</hook-ref-param>
</hook-ref>
```

Code Example 11: An Example hook-ref Element

The hook-ref element contains the following child elements:

- An optional description element, which specifies a description for the hook reference. By allowing application developers to provide some comments for each hook reference, the description element greatly improves the readability and maintainability of the deployment descriptor.
- A mandatory hook-ref-name element, which specifies a name for the hook reference. Note that this name will only be used to refer to the hook

reference from within the deployment descriptor, for example to define a precedence strategy that includes the hook in question.

- A mandatory hook-ref-type element, which specifies a type for the hook reference. Analogous to other EJB elements, the type's package must be fully specified.
- Zero or more hook-ref-param elements, which specify the hook instantiation parameters that should be passed to the hook constructor.

A more extensive example that illustrates where this element fits into the deployment descriptor is provided in Appendix A.

Defining the Precedence among Hook Instantiations

The precedence strategies we discussed above do not seem to fit very well into the session, entity, or message-driven elements, as they express a totally different kind of functionality. As precedence strategies express how hook instantiations are combined by the J2EE application server at runtime, we express this responsibility in the assembly-descriptor element at the end of the EJB deployment descriptor.

We introduce a child — which we will call hook-prec — into the assembly-descriptor element that expresses the precedence of all of the deployment descriptor's EJBs' hooks. Code Example 12 provides an example of such a hook-prec element, based on the JAsCo connector from Code Example 5:

```
<hook-prec>
   <before>logger</before>
   <before>lock</before>
   <replace>control</replace>
   <after>lock</after>
   <after>logger</after>
</hook-prec>
```

Code Example 12: An Example hook-prec Element

The order in which the hook-prec element's before, replace, after, and default children are specified defines the order in which the corresponding hooks' before, replace, after, and default behaviour methods will be executed, respectively. In our example, this means that the logger hook's before method will be executed before the lock hook's before method, the lock hook's before method will be executed before the control hook's replace method, and so forth. Note that this example implies the existence of three hook references named logger, lock, and control.

A more extensive example that illustrates where this element fits into the deployment descriptor is provided in Appendix A.

Initialising Hooks

When an EJB is being deployed, JAsCo should perform the following actions:

1. Instantiate the EJB's hooks.
2. Load the hooks' abstract method implementations, if any.
3. Initialise the hooks, if necessary.

These responsibilities obviously belong together. Precedence and combination strategies, on the other hand, belong together, too, but in a different place.

Hence, we will introduce another element into the session, entity, and message-driven elements, which we will call hook-init. This element will allow the application developer to specify how a hook should be initialised. Code Example 13 provides an example of such a hook-init element, based on the connector from Code Example 6:

```
<hook-init>
   <hook-init-name>birthday</hook-init-name>
   <hook-init-method>setDiscount</hook-init-method>
   <hook-init-param>0.02</hook-init-param>
</hook-init>
```

Code Example 13: An Example hook-init Element

The hook-init element contains the following child elements:

- A mandatory hook-init-name element that specifies the name of the hook reference that should be initialised.
- A mandatory hook-init-method element that specifies the name of the initialisation method.
- Zero or more hook-init-param elements that specify the initialisation method's initialisation parameters.

Hence, the semantics of our example are as follows: when an EJB is being deployed, JAsCo will initialise the birthday hook by calling its setDiscount method with 0.02 as its only parameter.

A more extensive example that illustrates where this element fits into the deployment descriptor is provided in Appendix A.

Defining Combination Strategies

As precedence strategies and combination strategies attempt to provide a solution to the same problem (the feature interaction problem we introduced above), it is only logical to put these two responsibilities near each other in the EJB deployment descriptor.

Hence, we will introduce a second new child into the assembly-descriptor element, which we will call hook-comb. Code Example 14 provides an example of such a hook-comb element, based on the connector from Code Example 8:

```
<hook-comb>
   <hook-comb-type>ExcludeCombinationStrategy</hook-comb-type>
   <hook-comb-param>birthday</hook-comb-param>
   <hook-comb-param>frequent</hook-comb-param>
</hook-comb>
```

Code Example 14: An Example hook-comb Element

The hook-comb element contains the following children:

- A mandatory hook-comb-type element, which specifies the type of the combination strategy that should be added to the application. Analogous to other EJB elements, the type's package must be fully specified.
- Zero or more hook-comb-param elements, which specify the combination strategy's parameters; these parameters will be passed to the combination strategy's constructor.

For each combination strategy the application developer wishes to add to the application, he needs to add a hook-comb element to the EJB deployment descriptor.

A more extensive example that illustrates where this element fits into the deployment descriptor is provided in Appendix A.

Conclusion

In this subsection, we have given an informal overview of how our modified EJB deployment descriptor will incorporate the JAsCo connectors' responsibilities. A formal description of the modified deployment descriptor's syntax, under the form of an extension to the EJB 2.0 deployment descriptor DTD [EJB], is given in Appendix B.

4.3.2 Full Integration

An implementation of the full integration of JAsCo and J2EE we described above requires modifying a J2EE application server. In this subsection, we will give a brief overview of how such an implementation could be achieved.

When an Enterprise JavaBean application is being deployed on a J2EE application server, the server reads in the application's deployment descriptor in order to configure the application's container. As we introduce an extension to this deployment descriptor, the part of the application server that reads in the deployment descriptor will need to be modified. It must also make sure that the container stores the JAsCo-related information declared in the deployment descriptor (e.g. which hooks are attached to which EJBs, etc.).

The container must also be modified so that, each time a method is executed on an EJB, it checks whether that EJB has any hooks attached to it, and if this is the case, execute those hooks, in the order specified by the application's precedence strategy, and after applying any combination strategies. It is here, too, that the special behaviour of session and entity aspect beans must be implemented. The container must also expose an API that allows the application server's web management interface to retrieve diagnostic information concerning the application's aspect-oriented behaviour, and allows it to enable and disable aspects at runtime.

4.3.3 Proof of Concept

Introduction

The following paragraphs provide a Proof of Concept implementation for our extension to the EJB deployment descriptor. This implementation is a preprocessor that takes an Enterprise JavaBeans JAR file as its input, and returns an Enterprise JAsCoBeans JAR file as its output, using the information that is declared in the deployment descriptor.

There are several advantages to this approach:

- EJB application developers are already familiar with the notion of a pre-processor, as they must use a preprocessor provided by the J2EE application server to generate stubs and skeletons for their EJBs. This utility, too, uses the information declared in the deployment descriptor in order to provide its functionality.

- The approach is simple, as no modifications to the J2EE application itself are required. This makes it ideal for implementing a proof of concept that will mainly be used for research purposes.
- It is reusable: as the preprocessor produces J2EE compatible EJB JAR files, it can be used with any J2EE application server. This would obviously not be the case if we were to modify the application server.

On the other hand, the disadvantage of this approach is that, by modifying the J2EE application server, some interesting advanced functionality could be provided. All basic JAsCo functionality can be provided through this approach, though.

Overview

The input of our preprocessor is a JAR file that consists of:

- A number of compiled EJB home and remote interfaces.
- A number of compiled EJB implementation classes.
- A number of uncompiled JAsCo aspect beans.
- A number of compiled JAsCo combination strategies.
- An Enterprise JAsCoBeans deployment descriptor that describes the EJBs and links them to the aspect beans.

The output of our preprocessor will be a JAR file that consists of:

- A number of compiled EJB home and remote interfaces.
- A number of compiled EJB implementation classes that have been transformed to “aspect-enabled” Enterprise JAsCoBeans.
- A number of compiled JAsCo aspect beans.
- A number of compiled combination strategies.
- An Enterprise JavaBeans deployment descriptor that describes the EJBs.
- A number of compiled JAsCo connectors that was generated based on the information in the Enterprise JAsCoBeans deployment descriptor.

This JAR file can then be deployed just like we did using our first approach. The difference with the first approach is that the user is completely unaware of the fact that JAsCo connectors even exist: they are eliminated from the Enterprise JAsCoBeans component model, making it simpler and easier to grasp. The application developer also does not need to worry about the technicalities of transforming EJBs or compiling aspect beans as this is done automatically by the preprocessor.

Procedure

This is the procedure employed by the preprocessor to transform an input JAR to an output JAR:

1. Decompress the JAR file to a temporary directory. This directory will be used to transform the EJBs, compile the aspect beans, and generate and compile the connectors.

2. Compile all the aspect beans (i.e. any file whose name ends with '.asp') in the temporary directory.
3. Generate a number of connectors based on the information declared in the EJB deployment descriptor, and store the compiled connectors in the temporary directory.
4. Transform the Enterprise JavaBeans implementation classes (whose names can be retrieved from the EJB deployment descriptor) into Enterprise JAsCoBeans, storing the transformed EJBs in the temporary directory.
5. Compress all files in the temporary directory into a JAR file; this file is the output JAR.

The prototype implementation uses the Apache Xerces2 [Xer] XML parser for Java to process the deployment descriptor, and uses the original JAsCo implementation to compile aspect beans, compile connectors, and transform implementation classes.

4.4 Usage

Using the Proof of Concept implementation we described above, these are the steps one may use to create a JAsCo2EE application:

1. Write and compile a number of EJBs, just as one would do in order to create a regular J2EE application. These EJBs implement the application's main functionality.
2. Write some JAsCo aspect beans, just as one would do in order to create a regular JAsCo application. These aspect beans implement the application's crosscutting functionality. Note that these aspect beans do not need to be compiled: the JAsCo2EE preprocessor will do this automatically.
3. Write an Enterprise JAsCoBeans deployment descriptor that describes the EJBs and links them to the aspect beans. This deployment descriptor provides the aspect beans with a concrete context, i.e. it links the aspect beans to a number of the EJBs' methods.
4. Package the EJBs, the aspect beans, and the deployment descriptor in a JAR, much like one would package a regular J2EE application.
5. Run the JAsCo2EE preprocessor with the JAR file as its input: this compiles the aspect beans, generates and compiles a number of connectors, and transforms the EJBs.
6. Generate the transformed EJBs' stubs and skeletons by using a utility provided by the application server on which the application will be deployed, just like one would do in order to create a regular J2EE application.
7. Deploy the application on the application server, just like one would do with a regular J2EE application.

Indeed, the procedure to create a JAsCo2EE application has been simplified somewhat with regard to the implementation we introduced through our first approach. The main advantage, however, is the fact that the application does not need to worry about connectors any more, as aspect beans are provided with a concrete context using our new deployment descriptor.

4.5 Conclusion

We may conclude that our second approach has accomplished its goal: it provides an implementation of JAsCo2EE that integrates JAsCo and J2EE, resulting in a component model that is much more elegant, and fits into the philosophy of J2EE better.

Chapter 5 Related Work

Today, J2EE is one of the most widely used platforms for enterprise application development in existence. When it comes to AOP, however, not much research has been done with regard to J2EE. The only J2EE application server that currently supports some form of AOP is JBoss. There are, however, a number of other aspect-oriented frameworks — such as JAC and Spring — that claim to be a valid alternative to J2EE. We will discuss each of these approaches below.

5.1 JBoss

JBossAOP [JBoa] is an aspect-oriented extension to JBoss [FR03], an open-source J2EE application server that claims to have a 25 percent market share [JBob]. JBoss introduces a new concept, called an *interceptor*, which allows an application developer to intercept constructor calls, field accesses, and method calls, and add extra behaviour to them. An interceptor is any java class that implements the Interceptor interface given by Code Example 15 [JBoc]:

```
package org.jboss.aop.advice;

import org.jboss.aop.joinpoint.Invocation;

public interface Interceptor {

   public String getName();
   public Object invoke(Invocation invocation) throws Throwable;
}
```

Code Example 15: The JBossAOP Interceptor Interface

JBossAOP uses the "dynamic proxies" approach we introduced above: at runtime, an object implementing the Interceptor interface can be inserted between a method call and the corresponding method execution, allowing extra behaviour to be executed at that point. JBoss does not add any new language constructs to Java: all aspect-oriented functionality is implemented using this kind of special interfaces. Code Example 16 provides an example of a simple logging interceptor, which outputs a message each time a method is invoked:

```
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;
import org.jboss.aop.joinpoint.MethodInvocation;

public class LoggingInterceptor implements Interceptor {

   public String getName() {

      return "LoggingInterceptor";
   }

   public Object invoke(Invocation invocation) throws Throwable {

      if (invocation instanceof MethodInvocation) {
```

```
            MethodInvocation mi = (MethodInvocation) invocation;
            System.out.println("Method "
               + mi.method
               + " was invoked.");
         }
         return invocation.invokeNext();
      }
   }
```

Code Example 16: A Simple Logging Interceptor in JBossAOP

Interceptors are linked to a program through the use of a special kind of XML file, called jboss-aop.xml. This file defines a number of pointcuts and binds these to interceptors. An example of such an XML file is given by Code Example 17:

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>

   <bind pointcut="all(Test)">
      <interceptor class="LoggingInterceptor"/>
   </bind>

</aop>
```

Code Example 17: A Simple jboss-aop.xml File

An interceptor's invoke method will receive any invocation that matches the pointcuts to which it is bound, allowing it to execute extra behaviour at those points in the execution of the program. For our example, this means that the logging interceptor will receive any method invocation on the Test class.

Multiple interceptors can be grouped into aspects, for example when they are logically related. Before they can be deployed on the JBoss application server, these aspects, the interceptors, the jboss-aop.xml file, and the main program (e.g. an EJB application) must be packaged together in a JAR; indeed, the jboss-aop.xml file acts as a deployment descriptor for JBoss' aspect-oriented functionality.

Although JBossAOP was introduced as an extension to the JBoss J2EE application server, the framework can also be used separately.

5.2 JAC

JAC — an acronym for "Java Aspect Components" — claims to be a valid alternative to J2EE regarding the development of large enterprise applications. Renaud Pawlak et al. [PDF+02] noticed that all J2EE application servers — by definition — provide a number of non-functional container services, such as security and transaction management, that are in fact crosscutting concerns. As these concerns are wired deep into the component model, the model becomes cluttered. Therefore, the JAC component model replaces EJBs with Plain Old Java Objects (POJOs), and J2EE's "hardwired" services with loosely-coupled, dynamically pluggable *aspect components* [JAC].

A typical JAC application consists of three kinds of entities:

The application's main functionality is implemented by a number of *POJOs*. This part of the application is actually just a regular Java program; hence, the programmer does not need to learn a new API to begin programming in JAC (contrary to J2EE).

Wrappers are plain Java classes that extend the Wrapper class. They implement the application's crosscutting functionality by defining methods that will be called by JAC when a constructor or a method is invoked. Code Example 18 provides an example of a simple logging wrapper that prints a message to standard output each time a constructor or a method is invoked:

```
import org.objectweb.jac.core.Wrapper;
import org.objectweb.jac.core.AspectComponent;
import org.aopalliance.intercept.MethodInvocation;

public class LoggingWrapper extends Wrapper {

   public LoggingWrapper(AspectComponent ac) {

      super(ac);
   }

   public Object invoke(MethodInvocation mi) throws Throwable {

      System.out.println("Method "
         + mi.getMethod().getName()
         + " was invoked.");
      return proceed(mi);
   }
}
```

Code Example 18: A Simple Logging Wrapper in JAC

A wrapper does not have any reference to where it will be applied; as wrappers are independent of a concrete context, they are easily reusable.

Aspect components are plain Java classes that extend the AspectComponent class. They are used to apply a number of wrappers onto a concrete context. Code Example 19 provides an example of a simple aspect component that will link the logging wrapper from Code Example 18 to all of the application's constructor and method invocations.

```
import org.objectweb.jac.core.AspectComponent;

public class LoggingAspect extends AspectComponent {

   public LoggingAspect() {

      pointcut("ALL", "ALL", "ALL", "LoggingWrapper", null, false);
   }
}
```

Code Example 19: A Simple Aspect Component in JAC

The fact that JAC does not introduce any special constructs into Java for expressing aspect components and pointcuts makes it very easy to learn. On the other hand, such constructs could be more expressive, and could make aspect components more readable.

The biggest drawback of JAC is its incompatibility with J2EE. Although JAC is great for implementing custom non-functional services, the standard container services provided by J2EE are not available. JAC does provide a number of pre-defined aspect components that are meant as a replacement for these container services, but these are quite simple when compared to the container services offered by J2EE application servers. In addition, J2EE offers a lot more than these non-functional container services [Van04].

5.3 Spring

The Spring framework [Spr] is aimed to make J2EE easier to use and promote good programming practice [Joh03]. It attempts to provide an alternative to J2EE with regard to enterprise application development by providing a more lightweight application model than J2EE.

The core of Spring's design is the *Inversion of Control (IoC)* container. An IoC container is much more lightweight than a J2EE container, and is based on the *Hollywood principle*: "don't call us, we'll call you". In J2EE, an application must explicitly ask its container to retrieve the EJBs it wants to use, while a Spring IoC container will automatically figure out when an application needs a certain bean, and retrieve this bean by itself.

Unlike J2EE application servers, which implement a lot of container services by themselves, Spring does not implement services such as logging and transaction management; instead, it makes existing open-source technologies easier to use. Based on the idea that EJB programming may be too complicated for some applications, Spring also provides application developers with the choice whether to use EJBs or POJOs.

Spring applications are typically configured using XML "bean definition" files — Spring's equivalent for EJB deployment descriptors. These are used to declare beans, data sources, etc.

AOP in Spring is done using interceptors. An interceptor is any Java class that implements the AOP Alliance [AOP] MethodInterceptor interface given by Code Example 20:

```
package org.aopalliance.intercept;

public interface MethodInterceptor extends Interceptor {

   Object invoke(MethodInvocation invocation) throws Throwable;
}
```

Code Example 20: The AOP Alliance Method Interceptor Interface

Code Example 21 provides an example of a simple logging interceptor, which will print a message before each method invocation:

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LoggingInterceptor implements MethodInterceptor {

   public Object invoke(MethodInvocation invocation) throws Throwable {

      System.out.println("Method "
         + invocation.getMethod().getName()
         + " was invoked.");
      return invocation.proceed();
   }
}
```

Code Example 21: A Simple Logging Interceptor in Spring

An interceptor is linked to the rest of the program by declaring and referencing it in the application's bean definition file.

Although Spring introduces some interesting ideas, the fact that it is not completely compatible with the EJB component model could be considered a disadvantage: while Spring considers the heavyweight J2EE containers a drawback of the application model, these containers do provide a lot of functionality that greatly facilitates enterprise application development.

Chapter 6 Conclusion

6.1 Summary

The goal of this dissertation is to investigate how JAsCo and J2EE could be integrated with each other. This research is conducted in the context of the Java 2 Platform, Enterprise Edition (J2EE), and the JAsCo aspect-oriented programming language.

The Java 2 Platform, Enterprise Edition is a de facto standard for enterprise application development. Although J2EE already provides applications with several services — such as security and transaction management — that are in fact crosscutting concerns, it does not allow the application developer to describe general crosscutting concerns.

JAsCo is an aspect-oriented programming language specifically tailored for component based software development, but it is based on the relatively simple JavaBeans component model. Integrating JAsCo and J2EE could prove useful for both JAsCo and J2EE.

Our first approach to the problem of integrating JAsCo with J2EE provides an implementation of JAsCo for J2EE in the most straightforward manner possible. It is based on the idea that the method used by the original implementation of JAsCo to add its aspect-oriented functionality to components — byte-code manipulation — can be applied to J2EE, too. Hence, the original implementation of JAsCo can be applied to J2EE with minimal changes.

We explain which implementation details prevent the original implementation from functioning correctly with J2EE, and provide a modified implementation that does function correctly. However, as this approach does not take into account the particularities of J2EE, the resulting JAsCo2EE component model is not very streamlined: there is redundancy in the work that is being done and J2EE is not really "aspect-aware".

Our second approach to the problem attempts to integrate JAsCo and J2EE as well as possible. We propose a backward compatible extension to the Enterprise JavaBeans component model that introduces aspect beans into the model. These aspect beans have the same characteristics of the aspect beans of the original implementation of JAsCo. The model also provides EJBs with built-in traps that can be used to attach and detach aspect beans.

The aspect beans, which do not have any reference to the context in which they will be applied, are provided with a concrete context through the use of deployment descriptors: we propose an extension to the EJB deployment descriptor that allows the application developer to declare the aspect beans to which each EJB is linked. Our model also supports runtime enabling and disabling of connectors.

Finally, we discuss whether JAsCo2EE's aspect-oriented functionality should be woven into the containers that wrap EJB applications, or into the EJBs' implementation

classes themselves. We also discuss whether or not session, entity, and message-driven aspect beans could be useful.

6.2 Contributions

The main contributions of this dissertation are the following:

- We describe and implement an extension of the original version of JAsCo that allows JAsCo to be applied to J2EE applications much like it is currently being applied to regular Java applications.
- We propose a new component model that is backward compatible with the Enterprise JavaBeans component model, and aims to fully integrate JAsCo and J2EE. The component model adds a new kind of entity to the EJB model, i.e. aspect beans. These aspect beans have the same characteristics as the aspect beans introduced by the original version of JAsCo.
- We describe an extension to the EJB deployment descriptor that allows application developers to add aspect-oriented functionality to EJBs: the deployment descriptor provides aspect beans with a concrete context, thus linking the EJBs to the aspect beans.
- We describe how our new component model can support runtime enabling and disabling of aspect-oriented functionality.
- We discuss where the aspect-oriented functionality could be added to the EJBs: in the EJB containers or in the EJBs themselves. We describe how each of these methods could be implemented.
- We discuss whether session, entity, and message-driven aspect beans could be useful.
- We provide a limited Proof of Concept (PoC) implementation that illustrates an important part of our new component model, i.e. the extension of the EJB deployment descriptor.

6.3 Applicability

Aspect-Oriented Software Development is based on the idea that the tyranny of the dominant decomposition can be breached by describing an application's crosscutting concerns separate from the application's main functionality. As even relatively small applications contain crosscutting concerns that cannot be modularised using existing technologies, it is clear that this will certainly be the case for large J2EE applications. Although one might express some of these concerns using J2EE's built-in container services, one can imagine that many J2EE applications could benefit from the possibility to express general-purpose aspects. As this is exactly the functionality that JAsCo2EE provides, the technology could certainly prove to be useful.

6.4 Future Work

The most obvious and important continuation of this dissertation is providing a complete implementation for our full integration of JAsCo and J2EE. As JAsCo2EE introduces an extension to the Enterprise JavaBeans component model, it is very probable that providing a fully functional implementation would require modifying an existing J2EE application server.

Virtually all continuations of the “regular” version of JAsCo apply to JAsCo2EE, too [Van04].

Appendix A Deployment Descriptor Example

This appendix illustrates where each of the elements we introduced fit into the EJB deployment descriptor. It is based on the following EJB deployment descriptor, which declares one session bean:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
   <description>
      A simple EJB application that contains one session bean.
   </description>
   <display-name>Simple EJB Application</display-name>
   <enterprise-beans>
      <session>
         <ejb-name>SimpleEjb</ejb-name>
         <home>application.SimpleEjbHome</home>
         <remote>application.SimpleEjbRemote</remote>
         <ejb-class>application.SimpleEjbImpl</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>SimpleEjb</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

As an example, we will add two hooks to this session bean, initialise one of those hooks, declare a precedence strategy, and add a combination strategy. In regular JAsCo, this would be accomplished using the following connector:

```
static connector SimpleConnector {

   application.Aspect.HookA hookA
      = new application.Aspect.HookA(* *.*(*));

   application.Aspect.HookB hookB
      = new application.Aspect.HookB(* *.*(*));

   hookA.initialise(31337);

   hookA.default();
   hookB.default();

   addCombinationStrategy(
      new application.ExcludeCombinationStrategy(hookA, hookB)
```

```
   );
}
```

In JAsCo2EE, however, this can be accomplished by adding a number of elements to the deployment descriptor we introduced above:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
   <description>
      A simple EJB application that contains one session bean.
   </description>
   <display-name>Simple EJB Application</display-name>
   <enterprise-beans>
      <session>
         <ejb-name>SimpleEjb</ejb-name>
         <home>application.SimpleEjbHome</home>
         <remote>application.SimpleEjbRemote</remote>
         <ejb-class>application.SimpleEjbImpl</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
         <hook-ref>
            <description>
               Links the EJB to the application.Aspect.HookA hook.
            </description>
            <hook-ref-name>hookA</hook-ref-name>
            <hook-ref-type>application.Aspect.HookA</hook-ref-type>
            <hook-ref-param>* *.*(*)</hook-ref-param>
         </hook-ref>
         <hook-ref>
            <description>
               Links the EJB to the application.Aspect.HookB hook.
            </description>
            <hook-ref-name>hookB</hook-ref-name>
            <hook-ref-type>application.Aspect.HookB</hook-ref-type>
            <hook-ref-param>* *.*(*)</hook-ref-param>
         </hook-ref>
         <hook-init>
            <hook-init-name>hookA</hook-init-name>
            <hook-init-method>initialise</hook-init-method>
            <hook-init-param>31337</hook-init-param>
         </hook-init>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>SimpleEjb</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
      <hook-prec>
         <default>hookA</default>
         <default>hookB</default>
      </hook-prec>
      <hook-comb>
         <hook-comb-type>
            application.ExcludeCombinationStrategy
         </hook-comb-type>
         <hook-comb-param>hookA</hook-comb-param>
         <hook-comb-param>hookB</hook-comb-param>
      </hook-comb>
```

```
   </assembly-descriptor>
</ejb-jar>
```

Appendix B Modified Deployment Descriptor DTD

```
<!--
   The session, entity, and message-driven elements have each been
   extended with two children:
      - an optional declaration of the bean's hook references, and
      - an optional declaration of the bean's hook initialisations.
-->

<!ELEMENT session (description?, display-name?, small-icon?,
                   large-icon?, ejb-name, home?, remote?, local-home?,
                   local?, ejb-class, session-type, transaction-type,
                   env-entry*, ejb-ref*, ejb-local-ref*,
                   security-role-ref*, security-identity?,
                   resource-ref*, resource-env-ref*, hook-ref*,
                   hook-init*)>

<!ELEMENT entity (description?, display-name?, small-icon?, large-icon?,
                  ejb-name, home?, remote?, local-home?, local?,
                  ejb-class, persistence-type, prim-key-class,
                  reentrant, cmp-version?, abstract-schema-name?,
                  cmp-field*, primkey-field?, env-entry*, ejb-ref*,
                  ejb-local-ref*, security-role-ref*,
                  security-identity?, resource-ref*, resource-env-ref*,
                  query*, hook-ref*, hook-init*)>

<!ELEMENT message-driven (description?, display-name?, small-icon?,
                          large-icon?, ejb-name, ejb-class,
                          transaction-type, message-selector?,
                          acknowledge-mode?,
                          message-driven-destination?, env-entry*,
                          ejb-ref*, ejb-local-ref*, security-identity?,
                          resource-ref*, resource-env-ref*, hook-ref*,
                          hook-init*)>



<!--
   The hook-ref element declares a bean's hook reference.  It contains:
      - an optional description for the hook reference,
      - a mandatory name for the hook reference,
      - a mandatory type for the hook reference, and
      - an optional declaration of the hook reference's instantiation
        parameters.

   Used in: session, entity, message-driven
-->
<!ELEMENT hook-ref (description?, hook-ref-name, hook-ref-type,
                    hook-ref-param*)>

<!--
   The hook-ref-name element specifies the name for a hook reference;
   this name will be used to refer to the hook reference from within
   the deployment descriptor's other elements.  Note that this name may
   contain dots ('.') and forward slashes ('/').

   Used in: hook-ref
-->
<!ELEMENT hook-ref-name (#PCDATA)>
```

```
<!--
   The hook-ref-type element specifies the type for a hook reference.
   Note that the type's package must be fully specified.

   Used in: hook-ref
-->
<!ELEMENT hook-ref-type (#PCDATA)>

<!--
   The hook-ref-param element declares a hook reference's instantiation
   parameter.  These parameters will be passed to the hook's constructor
   when the hook is being instantiated.

   Used in: hook-ref
-->
<!ELEMENT hook-ref-param (#PCDATA)>



<!--
   The hook-init element declares a bean's hook initialisation.  It
   contains:
      - a mandatory name for the hook to be initialised,
      - a mandatory name of an initialisation method, and
      - an optional declaration of the initialisation method's
        parameters.

   Used in: session, entity, message-driven
-->
<!ELEMENT hook-init (hook-init-name, hook-init-method,
                     hook-init-param*)>

<!--
   The hook-init-name element specifies the name for a hook to be
   initialised.  This name must correspond to the name of a hook
   reference.

   Used in: hook-init
-->
<!ELEMENT hook-init-name (#PCDATA)>

<!--
   The hook-init-method element specifies the name of an initialisation
   method.  Note that this method must be defined by the hook whose name
   is specified by the hook-init-name element.

   Used in: hook-init
-->
<!ELEMENT hook-init-method (#PCDATA)>

<!--
   The hook-init-param element declares an initialisation method's
   parameter.  These parameters will be passed to the initialisation
   method whose name is specified by the hook-init-method element.

   Used in: hook-init
-->
<!ELEMENT hook-init-param (#PCDATA)>



<!--
   The assembly-descriptor element has been extended with two children:
      - an optional declaration of the application's precedence
        strategy, and
      - an optional declaration of the application's combination
        strategies.
-->
```

```
<!ELEMENT assembly-descriptor (security-role*, method-permission*,
                               container-transaction*, exclude-list?,
                               hook-prec?, hook-comb*)>


<!--
   The hook-prec element declares a precedence strategy.  The element
   contains a combination of zero or more default, before, replace, and
   after elements, which each specify the name of a hook reference.  The
   order in which the default, before, replace, and after elements are
   specified specifies the order in which the corresponding hooks'
   default, before, replace, and after behaviour methods will be
   invoked.

   Used in: assembly-descriptor
-->
<!ELEMENT hook-prec (default | before | replace | after)*>

<!ELEMENT default (#PCDATA)>
<!ELEMENT before  (#PCDATA)>
<!ELEMENT replace (#PCDATA)>
<!ELEMENT after   (#PCDATA)>



<!--
   The hook-comb element declares a combination strategy.
   It contains:
      - a mandatory type for the combination strategy, and
      - an optional declaration of the combination strategy's
        parameters.

   Used in: assembly-descriptor
-->
<!ELEMENT hook-comb (hook-comb-type, hook-comb-param*)>

<!--
   The hook-comb-type element specifies the type for a combination
   strategy.

   Used in: hook-comb
-->
<!ELEMENT hook-comb-type (#PCDATA)>

<!--
   The hook-comb-param element declares a combination strategy's
   parameter.

   Used in: hook-comb
-->
<!ELEMENT hook-comb-param (#PCDATA)>
```

Glossary

AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
ASP	Active Server Pages
CBSD	Component-Based Software Development
CGI	Common Gateway Interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DTD	Document Type Definition
EJB	Enterprise JavaBeans
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoC	Inversion of Control
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JAC	Java Aspect Components
JAR	Java Archive
JMS	Java Messaging Service
JRE	Java Runtime Environment
JSP	JavaServer Pages
.NET	Microsoft .NET
OMG	Object Management Group
OOP	Object-Oriented Programming
OOSD	Object-Oriented Software Development
PARC	Palo Alto Research Center
PoC	Proof of Concept
POJO	Plain Old Java Object
PHP	PHP: Hypertext Preprocessor
SSEL	System and Software Engineering Lab

URL	Uniform Resource Locator
VUB	Vrije Universiteit Brussel
XML	Extensible Markup Language

Bibliography

[AOP] The AOP Alliance. Available on the WWW at http://aopalliance.sourceforge.net.

[ASP] Microsoft Active Server Pages. Available on the WWW at http://msdn.microsoft.com/asp.

[BAT01] Lodewijk Bergmans, Mehmet Akşit, and Bedir Tekinerdoğan. Aspect Composition Using Composition Filters. In Mehmet Akşit, editor, *Software Architectures and Component Technology,* pages 357–382. Kluwer Academic Publishers, Boston, MA, USA, October 2001.

[Bea] The Bean Builder. Available on the WWW at http://java.sun.com/products/javabeans/beanbuilder/index.jsp.

[Bro86] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. In Hans-Jürgen Kugler, editor, Proceedings of the IFIP 10th World Computer Congress, pages 1069–1076, Amsterdam, Netherlands, September 1986. Elsevier Science Publishers.

[BW96] Alan W. Brown and Kurt C. Wallnan. Engineering of Component-Based Systems. In Alan W. Brown, editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute,* pages 7–15. IEEE Computer Society Press, Los Alamitos, CA, USA, September 1996.

[Cle96] Paul C. Clements. From Subroutines to Subsystems: Component-Based Software Development. In Alan W. Brown, editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute,* pages 3–6. IEEE Computer Society Press, Los Alamitos, CA, USA, September 1996.

[COM95] The Component Object Model Specification, Version 0.9. Specification Document, Microsoft Corporation, Redmond, WA, USA, October 1995.

[Cor04] Common Object Request Broker Architecture: Core Specification, Version 3.0.3. Specification Document, Object Management Group, Inc., Needham, MA, USA, March 2004.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM,* 15(10):859–866, October 1972.

[DYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise JavaBeans Specification, Version 2.0 (Final Release). Specification Document, Sun Microsystems, Inc., Palo Alto, CA, USA, August 2001.

[Ecl] The Eclipse Project. Available on the WWW at http://www.eclipse.org.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM,* 44(10):29–32, October 2001.

[EJB] The EJB 2.0 Deployment Descriptor DTD. Available on the WWW at http://java.sun.com/dtd/ejb-jar_2_0.dtd.

[FR03] Marc Fleury and Francisco Reverbel. The JBoss Extensible Server. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Middleware Conference (Middleware'03),* pages 344–373, Heidelberg, Germany, June 2003. Springer-Verlag.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Boston, MA, USA, October 1994.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. Addison-Wesley Professional, Boston, MA, USA, second edition, June 2000.

[Ham97] Graham Hamilton. JavaBeans Specification, Version 1.01-A. Specification Document, Sun Microsystems, Inc., Mountain View, CA, USA, August 1997.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Professional, Boston, MA, USA, June 2001.

[HH04] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In Gail Murphy and Karl Lieberherr, editors, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04),* pages 26–35, New York, NY, USA, March 2004. ACM Press.

[J2E] Java 2 Platform, Enterprise Edition Overview. Available on the WWW at http://java.sun.com/j2ee/overview.html.

[JAC] The JAC Project. Available on the WWW at http://jac.objectweb.org.

[JBoa] JBoss Aspect Oriented Programming. Available on the WWW at http://www.jboss.org/developers/projects/jboss/aop.

[JBob] JBoss Overview. Available on the WWW at http://www.jboss.org/overview.

[JBoc] JBossAOP Documentation. Available on the WWW at http://www.jboss.org/wiki/Wiki.jsp?page=JBossAOP.

[Joh03] Rod Johnson. Introducing the Spring Framework. *TheServerSide.COM,* October 2003. Available on the WWW at http://www.theserverside.com/articles/article.tss?l=SpringFramework.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01),* pages 327–353, Heidelberg, Germany, June 2001. Springer-Verlag.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97),* pages 220–242, Heidelberg, Germany, June 1997. Springer-Verlag.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, USA, March 1999.

[LOO01] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM,* 44(10):39–41, October 2001.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98),* pages 355–382, Heidelberg, Germany, July 1998. Springer-Verlag.

[OT99] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Technical Report RC 21452(96717)16APR99, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, April 1999.

[OT00] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development,* Boston, MA, USA, 2000. Kluwer Academic Publishers.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM,* 15(12):1053–1058, December 1972.

[PDF+02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. JAC: An Aspect-Based Distributed Dynamic Framework. December 2002.

[PHP] PHP: Hypertext Preprocessor. Available on the WWW at http://www.php.net.

[PSC+01] Elke Pulvermüller, Andreas Speck, James Coplien, Maja D'Hondt, and Wolfgang De Meuter. Feature Interaction in Composed Systems. Technical Report 2001-14, Universität Karlsruhe, Karlsruhe, Germany, September 2001.

[Sha03] Bill Shannon. Java 2 Platform, Enterprise Edition Specification, Version 1.4 (Proposed Final Draft 3). Specification Document, Sun Microsystems, Inc., Santa Clara, CA, USA, April 2003.

[Spr] The Spring Framework. Available on the WWW at http://www.springframework.org.

[Suv02] Davy Suvée. JAsCo: A General-Purpose Aspect-Oriented Component Language for Java. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2002.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component-Based Software Development. In William G. Griswold and Mehmet Akşit, editors, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03),* pages 21–29, New York, NY, USA, March 2003. ACM Press.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, Boston, MA, USA, December 1997.

[Van04] Wim Vanderperren. *Combining Aspect-Oriented and Component-Based Software Engineering.* PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2004.

[Weba] The BEA WebLogic Application Server. Available on the WWW at http://www.bea.com/products/weblogic/server.

[Webb] The IBM WebSphere Application Server. Available on the WWW at http://www.ibm.com/software/webservers/appserv.

[WTM+] Eric Wohlstadter, Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, and Premkumar Devanbu. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. Accepted at the 26th International Conference on Software Engineering (ICSE'04).

[Xer] The Xerces2 Java Parser. Available on the WWW at http://xml.apache.org/xerces2-j.