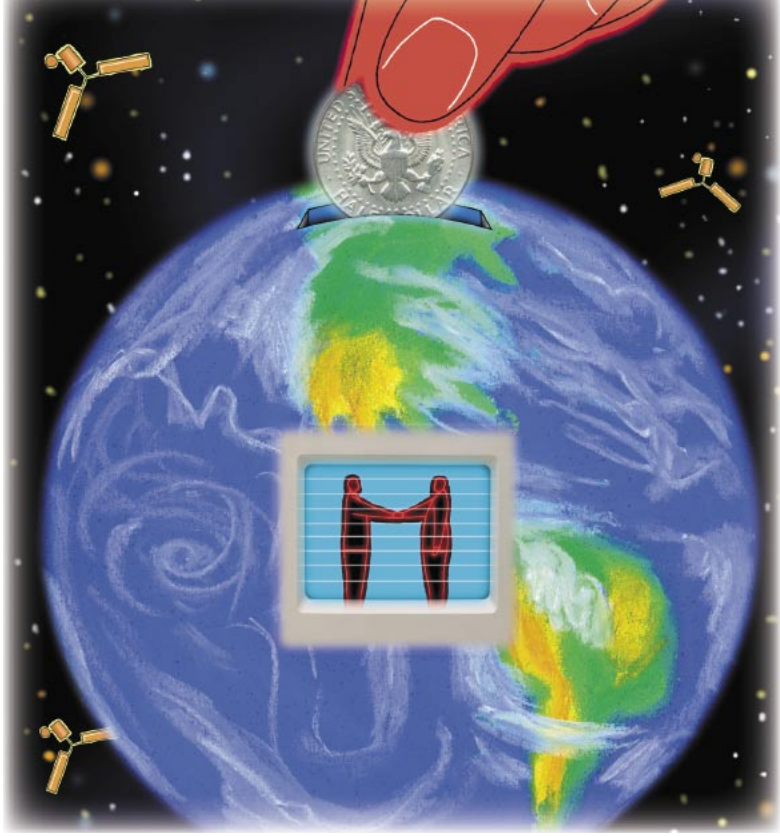

Das fehlende Glied

JBoss 4 und AOP

Dirk Mascher

Die neue Version 4 des freien Applikationsservers JBoss bringt die Aspektorientierung in die Welt der J2EE-Middleware. JBoss ermöglicht es dadurch, technische und fachliche Querschnittsbelange von J2EE-Enterprise-Anwendungen sauber zu kapseln und sogar beliebig zur Laufzeit zu aktivieren bzw. zu deaktivieren. JBoss 4 bringt jedoch nicht nur das eigens entwickelte AOP-Framework JBossAOP mit, sondern enthält auch gleich eine ganze Reihe von vordefinierten und teilweise völlig neuartigen Aspekten in Form der JBossAOP Servics. Der vorliegende Artikel gibt eine Einführung in JBossAOP und demonstriert anhand von Beispielen die Mächtigkeit dieses neuen JBoss-Features. Ein Blick hinter die Kulissen zeigt, wie in JBossAOP Aspekte durch die Manipulation von Bytecode in bestehende Klassen eingewoben werden können. Abgerundet wird der Artikel durch eine kurze Darstellung einzelner Aspekte der JBossAOP Services.

Neben der offiziellen J2EE 1.4-Zertifizierung (JBoss ist der erste Opensource-Applikationsserver, dem das gelingt!) und der damit einhergehenden Unterstützung der aktuellen Spezifikationen aller J2EE-Teiltechnologien (beispielsweise EJB 2.1) ist die wohl vielversprechendste Neuerung von JBoss 4 der Einzug der Aspektorientierung. Integraler Bestandteil von JBoss 4 ist das eigens entwickelte AOP-Framework *JBossAOP*, das einerseits direkt für die Entwicklung applikationsspezifischer Querschnittsbelange (Fachbegriff: „crosscutting concerns") verwendet werden kann, andererseits aber auch die Basis für verschiedene, neuartige Infrastrukturdienste bildet. JBoss Inc. bezeichnet die neue Version von JBoss aus diesem Grund auch als *Aspect-Oriented Middleware* (AOM).

In gewisser Weise konnte aber auch die Version 3.x von JBoss bereits als „aspektorientiert" bezeichnet werden. Schließlich ist es über das Konzept von clientseitigen und serverseitigen Interceptoren mit JBoss schon seit längerer Zeit möglich, so gut wie jeden „Aspekt" bei der Invokation von EJBs zu konfigurieren oder sogar zu verändern. Allerdings ist das Interceptor-Konzept von JBoss 3.x beschränkt auf EJBs und JMX MBeans. JBoss 4 dagegen wartet mit echter Aspektorientierung auf, die es erlaubt, Verhalten an beliebigen Stellen im JBoss Server oder einer in JBoss deployten Anwendung einzufügen oder zu verändern.

Terminologie

Wenn man sich mit JBossAOP beschäftigt, wird man mit verschiedenen Begriffen konfrontiert. Einige dieser Begriffe fallen in den allgemeinen Bereich der Aspektorientierung, andere sind eher JBoss-spezifisch. Der Vollständigkeit halber werden die wichtigsten Begriffe im Folgenden kurz aufgeführt:

- Ein *Joinpoint* ist eine bestimmte Stelle im Programm während dessen Ausführung.
- Ein *Advice* definiert ein Verhalten, das an einem Joinpoint ausgeführt werden soll.
- Ein *Aspect* ist eine Java-Klasse, die ein oder mehrere Advices kapselt.
- Ein *Interceptor* ist ein Aspect mit nur genau einem Advice.
- Ein *Pointcut* definiert eine Menge von Joinpoints.
- *Pointcut-Expressions* bilden eine Sprache zur Definition von Pointcuts. Pointcut-Expressions in JBossAOP entsprechen im Wesentlichen der bekannten Syntax von AspectJ. Als wichtigster Unterschied ist hier die Erweiterung um *Annotations* zu nennen (siehe weiter unten).

Obwohl der Begriff *POJO* mit AOP eigentlich direkt nichts zu tun hat, stößt man bei Themen rund um die Aspektorientierung doch öfter auf dieses Modewort, insbesondere im Zusammenhang mit JBossAOP. Aus diesem Grund macht es Sinn, auch auf diesen Begriff kurz einzugehen. Was ist also ein POJO? POJO steht für *plain old java object*. Damit sind „einfache" Java-Objekte gemeint, deren Klassen keine speziellen Interfaces implementieren bzw. nicht von speziellen Basis-Klassen einer komplexen API abgeleitet sind. Eine EJB beispielsweise ist demnach genau kein POJO.

Warum wird dieser Begriff nun öfter im Zusammenhang mit der Aspektorientierung verwendet? Der Grund, zumindest bezogen auf JBossAOP, ist der, dass die Aspektorientierung quasi als Schlüssel für die Vereinfachung von Middleware gesehen wird, mit dem Ziel Infrastrukturdienste für den Anwendungsentwickler völlig transparent zu machen. Idealerweise soll der Anwendungsentwickler also nur noch POJO-Klassen schreiben, d. h. sich zu 100 % auf die Implementierung von Anwendungslogik konzentrieren können. Aspekte wie Verteilung, Persistenz, Sicherheit, Transaktionen oder auch Clustering sollen zu einer späteren Phase im Entwicklungsprozess beliebig – je nach Bedarf der Anwendung – zuschaltbar sein.

JBossAOP

JBossAOP ist integraler Bestandteil von JBoss 4 und kann somit direkt in jeder JBoss 4-Umgebung (basierend auf der „all"- oder der neuen „standard"-Konfiguration) eingesetzt werden. Alternativ ist es aber auch möglich, JBossAOP außerhalb des JBoss-Applikationsservers, d. h. „standalone", zu verwenden. Einzige Voraussetzung hierfür ist die Verwendung eines JDK 1.4.x oder höher. JBoss 4 steht zum Zeitpunkt der Erstellung des Artikels in der finalen Version 4.0.0 auf der JBoss-Web-Seite zum Download bereit [JBOSS]. Die Standalone-Variante von JBossAOP kann in der Version 1.0 Release Candidate 2 (RC2) heruntergeladen werden.

JBossAOP ist eine reine Java-basierte AOP-Implementierung, d. h. im Gegensatz zu beispielsweise AspectJ definiert JBossAOP keine Java-Spracherweiterung. Konfiguriert werden Aspekte in JBossAOP vielmehr über XML. In der Datei „jboss-aop.xml" werden Pointcut-Expressions und Aspect-Bindings definiert. Ausgewertet werden diese von JBossAOP, indem der Bytecode der Klassen, die auf Pointcut-Expressions zutreffen, manipuliert wird. Diese Bytecode-Manipulation

kann dabei entweder zur Compile-Zeit oder zur Laufzeit angestoßen werden. Im ersten Fall muss ein spezieller AOP-Compiler (AOPC) verwendet werden. Glücklicherweise bringt JBossAOP für diesen AOPC auch gleich eine passende ANT-Task-Implementierung mit, sodass dieser Schritt relativ leicht in eine über ANT automatisierte Build-Umgebung integriert werden kann [ANT]. Alternativ kann die Bytecode-Manipulation aber auch erst zur Laufzeit stattfinden. Hierfür muss der Java System-Classloader durch einen speziellen JBossAOP-Classloader ersetzt werden. Dies ist durch Angabe des System-Properties **java.system.class.loader** wie in folgendem Beispiel möglich:

```
$ java -Djava.system.class.loader=
  org.jboss.aop.standalone.SystemClassLoader MyApplication
```

Bytecode-Manipulation – Ein Blick hinter die Kulissen

Was genau bedeutet nun aber eigentlich Bytecode-Manipulation? Um diese Frage zu beantworten, bietet es sich an, den durch AOPC modifizierten Bytecode einer Beispiel-Klasse unter Verwendung eines Decompilers, z. B. JAD, wieder zurück in lesbaren Java-Code zu transformieren [JAD].

Ausgangssituation für die folgende Betrachtung ist die Beispiel-Klasse **HelloWorld.java** in Listing 1.

```
public class HelloWorld {
  public static void sayHello() {
    System.out.println("hello world!");
  }
}
```

Listing 1: HelloWorld.java

Ein JBossAOP-Interceptor (es ist an dieser Stelle unerheblich, welchen Aspekt bzw. Advice der Interceptor implementiert) soll über die in Listing 2 gezeigte Pointcut-Expression in der Datei „jboss-aop.xml“ an den Execution-Joinpoint der statischen **sayHello**-Methode der Beispiel-Klasse gebunden werden.

```
<bind pointcut="execution(* HelloWorld->sayHello(..))">
  <interceptor class="SampleInterceptor"/>
</bind>
```

Listing 2: jboss-aop.xml

Spätestens an dieser Stelle ist es notwendig, kurz auf die JBossAOP-Syntax von Pointcut-Expressions einzugehen. Mit der Pointcut-Expression aus Listing 2 wird eine Menge von Execution-Joinpoints für **sayHello**-Methodenaufrufe der Klasse **HelloWorld** definiert, wobei der Return-Typ und die Parameter-Typen der **sayHello**-Methoden für die Auswahl nicht weiter relevant sein sollen und aus diesem Grund über Wildcards ausgedrückt werden. Die **sayHello**-Methode unserer Beispiel-Klasse passt also in jedem Fall auf diesen Ausdruck. Jetzt muss nur noch geklärt werden, was ein Execution-Joinpoint ist. Damit ist der Punkt zur Laufzeit unmittelbar vor der ersten Anweisung innerhalb einer aufgerufenen Methode gemeint, in unserem Beispiel also die Stelle direkt vor der **System.out.println**-Anweisung der **sayHello**-Methode in Listing 1.

Damit können wir uns jetzt anschauen, welchen Bytecode AOPC generiert. Der generierte Code entspricht in unserem Beispiel im Wesentlichen den in Listing 3 farblich hervorgehobenen Code-Passagen. Der Code ist allerdings aus Gründen der Übersichtlichkeit vereinfacht dargestellt. Tatsächlich per AOPC manipulierter und anschließend dekompilierter Java-Code würde u. a. noch Methoden zur Implementierung des für diese Betrachtung nicht weiter relevanten Advised-Interface beinhalten.

```
import org.jboss.aop.*;
import org.jboss.aop.joinpoint.InvocationBase;

public class HelloWorld implements Advised {
  public static void test$HelloWorld$sayHello$aop() {
    System.out.println("hello world!");
  }

  public static void sayHello {
    if (aop$classAdvisor$aop.doesHaveAspects) {
      // invoke chain of aspects and test$HelloWorld$sayHello$aop
    } else {
    test$HelloWorld$sayHello$aop();
    }
  }
}
```

Listing 3: Dekompilierte Version von HelloWorld.class nach AOPC Bytecode-Manipulation

Die durchgeführte Bytecode-Manipulation basiert auf dem Prinzip des „Method-Wrapping“. Der Methoden-Rumpf der **sayHello**-Methode der Ausgangsklasse wurde in die neu eingefügte Methode **test$HelloWorld$sayHello$aop** verschoben. Die **sayHello**-Methode der manipulierten Klasse enthält einen neuen Rumpf, der in einer einfachen If-Abfrage feststellt, ob für diesen Joinpoint Aspekte definiert sind. Falls ja, werden diese in einer Kette nacheinander aufgerufen. Am Ende der Kette wird schließlich der Rumpf der eigentlichen **sayHello**-Methode aufgerufen. Sind keine Aspekte definiert, wird direkt die **test$HelloWorld$sayHello$aop**-Methode aufgerufen, was exakt dem Verhalten eines Aufrufs der **sayHello**-Methode der nicht modifizierten Klasse entspricht.

Interessant ist folgende Beobachtung: Obwohl in der Datei „jboss-aop.xml“ ein konkreter Interceptor an die Ausführung der **sayHello**-Methode gebunden wurde, ist diese Information an keiner Stelle in dem dekompilierten Code zu finden. Der Grund hierfür: JBossAOP ermittelt erst zur Laufzeit durch (nochmalige) Auswertung der „jboss-aop.xml“, welche Aspekte an welchen Joinpoints eingebunden werden sollen. Die Grundvoraussetzung für dynamisches AOP in JBoss!

AOP - dynamisch

Die Fähigkeit, Aspekte erst zur Laufzeit an Joinpoints einzubinden und wieder zu entfernen bzw. Aspekte nur an bestimmte Instanzen einer Klasse zu binden, wird als dynamisches AOP bezeichnet. Wie wir oben gesehen haben, werden in JBossAOP selbst Aspekte, die schon vorab über die Datei „jboss-aop.xml“ definiert sind, erst zur Laufzeit gebunden. Mit Hilfe eines speziellen **prepare**-Tags ist es aber auch möglich, Klassen für den Einsatz von dynamischem AOP vorzubereiten, ohne dass in „jboss-aop.xml“ bereits konkrete Aspekte definiert werden müssen.

```
<prepare expr="execution(* HelloWorld->sayHello(..))" />
```

Listing 4: Beispiel für prepare-Tag in jboss-aop.xml

Listing 4 zeigt das **prepare**-Tag zur Vorbereitung der **sayHello**-Methode unserer Beispiel-Klasse. AOPC würde bei Auswertung dieses **prepare**-Tags dieselbe Bytecode-Manipulation durchführen, wie oben bereits gezeigt. Einziger Unterschied ist in diesem Fall, dass zur Laufzeit erst mal noch kein Aspekt an den Execution-Joinpoint der **sayHello**-Methode gebunden wäre. Was hierdurch allerdings erreicht wird, ist, dass die Klasse (bzw. alle **sayHello**-Methoden der Klasse) darauf vorbereitet wird, dynamisch zur Laufzeit mit Aspekten versehen zu werden. Wie dies genau funktioniert, wird in einem anderen Beispiel weiter unten verdeutlicht.

Pointcuts & Co

Neben den bereits erwähnten Execution-Pointcuts definiert die JBossAOP Pointcut-Expression Language auch die Möglichkeit Constructor-Pointcuts, Field-Pointcuts, Caller-Pointcuts und Introductions zu definieren. Für eine ausführliche Beschreibung der jeweiligen Syntax sei an dieser Stelle auf die JBossAOP-Dokumentation verwiesen [JBOSS].

Constructor-Pointcuts entsprechen im Wesentlichen Execution-Pointcuts, nur eben nicht für Methoden sondern für Konstruktoren. Field-Pointcuts erlauben es, sich in den Lese- oder Schreibzugriff auf Member-Variablen von Klassen zu hängen. Caller-Pointcuts ähneln Execution-Pointcuts, jedoch mit dem Unterschied, dass sie Joinpoints unmittelbar vor einem Methoden-Aufruf definieren. Mit Caller-Pointcuts ist es somit möglich, sich auch in Methoden-Aufrufe von System-Klassen zu hängen. Genau dies ist für Execution-Pointcuts nämlich verboten, da hierfür Bytecode von **java.***- bzw. **javax.***- oder **com.sun.***-Klassen modifiziert werden müsste.

Introductions sind ein weiteres interessantes Feature, das in diesem Rahmen allerdings nur angerissen werden kann. Über eine Introduction kann definiert werden, dass eine Klasse nach der Bytecode-Manipulation ein oder mehrere zusätzliche Java-Interfaces implementiert. Sieht man mal von reinen Marker-Interfaces wie beispielsweise **java.io.Serializable** ab, ist hierfür natürlich auch eine Implementierung dieser Interfaces nötig. Genau diese werden über so genannte Mixin-Klassen beigesteuert. Der Code der Mixin-Klassen wird per Bytecode-Manipulation in den Code der ursprünglichen Klasse eingefügt. Zur Laufzeit ist es somit möglich, ein derartig instrumentiertes Objekt auf eines der neuen Interfaces zu casten. So könnte man beispielsweise existierende Klassen um ein Logging-Interface erweitern, über das man zur Laufzeit pro Instanz verschiedene Log-Level setzen kann.

Annotations und Metadata

JBossAOP bietet als erstes AOP-Framework vollständige Unterstützung für Annotations, d. h. dass Aspekte aufgrund der Anreicherung von Code mit Metadaten an Joinpoints gebunden werden können (statisch und dynamisch). JBossAOP unterstützt hierbei nicht nur Annotations wie sie ab Java in der Version 5.0 ein Standard-Sprachbestandteil sind [JSR-175], sondern unterstützt dieses Feature auch für ältere Java-Versionen (1.4.x). Da ein Compiler für Java 1.4.x jedoch noch keine Annotations kennt, müssen in diesem Fall die Metadaten über javadoc-ähnliche Tags formuliert werden. Diese werden dann nicht vom Java-Compiler in Bytecode kompiliert, sondern von einem speziellen JBossAOP-Annotation-Compiler in XML, sodass auf die Metadaten zur Laufzeit zugegriffen werden kann. Wie für den AOPC-Compiler gibt es auch für den Annotation-Compiler eine passende Task-Implementierung für ANT, sodass auch dieser Schritt leicht über eine ANT-basierte Build-Umgebung automatisierbar ist.

Wie sehen nun aber Annotations in JBossAOP aus und was kann man damit machen? Das Beispiel aus dem JBossAOP-Tutorium (s. Listing 5) zeigt, wie Java-Code mit doclet-Tags, d. h. 1.4.x-Annotations, attributiert werden kann. In dem Beispiel wird eine Methode mit zwei Annotations versehen. Die **billable**-Annotation ist ein Beispiel für

```
<bind pointcut="execution(* POJO->@billable(..))">
  <interceptor class="BillingInterceptor"/>
</bind>

<bind pointcut="all(@trace)">
  <interceptor class="TraceInterceptor"/>
</bind>
```

Listing 6: Beispiel für die Verwendung von Annotations in Pointcut-Expressions

```
/**
 * @@billable (amount=0.05)
 * @@trace
 */
public void someMethod() {
  // ...
}
```

Listing 5: Beispiel für JBossAOP 1.4.x-Annotations

die Markierung einer fachlichen Querschnittsfunktionalität und soll ausdrücken, dass dem aktuellen Benutzer jeder Aufruf dieser Methode zu dem angegebenen Betrag in Rechnung gestellt werden soll. Die **trace**-Annotation markiert eine technische Querschnittsfunktionalität – nämlich, dass jeder Aufruf der Methode protokolliert werden soll.

Die Semantik der beiden Annotation-Tags wird wiederum über Pointcut-Expressions und Aspect-Bindings in der Datei „jboss-aop.xml“ (s. Listing 6) festgelegt. In dem Beispiel wird definiert, dass bei jedem Execution-Joinpoint einer mit der **billable**-Annotation markierten Methode innerhalb der Klasse **POJO** ein Interceptor der Klasse **BillingInterceptor** aufgerufen werden soll. Für das Tracing wird dagegen die **all**-Syntax der Pointcut-Expression-Language verwendet. Damit wird spezifiziert, dass alle Methoden, Konstruktoren und Felder, die mit der **trace**-Annotation markiert sind, mit Hilfe eines Interceptors der Klasse **TraceInterceptor** protokolliert werden sollen.

Natürlich ist es auch möglich die Datei „metadata-aop.xml“, die von dem Annotation-Compiler erzeugt wird, von Hand zu schreiben, d. h. Metadaten implizit zur Verfügung zu stellen. Das macht beispielsweise dann Sinn, wenn man Security-Metadaten definieren möchte. In diesem Fall wäre es sicher unklug, diese direkt im Quellcode zu hinterlegen. Die Mächtigkeit von AOP in Zusammenhang mit Annotations bzw. Metadaten wird in JBoss 4 eindrucksvoll durch einige der JBoss AOP Services demonstriert (siehe weiter unten).

Im Falle von Java 5.0-Annotations würde die attributierte Beispiel-Methode wie folgt aussehen:

```
@billable(amount=0.05) @trace
public void someMethod() {
  // ...
}
```

Die Semantik der Metadaten soll dabei dieselbe sein und wird durch dieselbe „jboss-aop.xml“ (s. Listing 6) definiert. Wie man erkennt, waren die JBoss-Entwickler sehr bemüht, die Syntax für Java 1.4.x doclet-basierte Metadaten an die Java 5.0-Annotation-Syntax anzulehnen.

JBossAOP in Aktion

Im Folgenden wird die Anwendung von JBossAOP anhand eines einfachen Beispiels demonstriert. Ausgangssituation ist eine J2EE-Enterprise-Anwendung, die an verschiedene Backend-Systeme angebunden ist. Die komplette Business-Logik des Systems ist in einem EJB-Layer gekapselt bzw. implementiert. Zugegriffen wird auf das EJB-Layer über eine Session-Facade aus der Web-Tier heraus. In einer relativ späten Phase des Projekts möchte man einen Profiling-Mechanismus für alle Methoden der Business-Logik einführen. Das Profiling soll dabei beliebig zur Laufzeit ein- und ausschaltbar sein.

Da es sich hier um eine typische Querschnittsfunktionalität handelt, haben wir eine perfekte Anwendung für AOP! Mittels AOP können wir die Profiling-Funktionalität an genau einer Stelle kapseln und dadurch verhindern, dass der existierende Code für die Fachlogik durch den Code für das Profiling „verschmutzt“ wird. Um den Profiling-Aspekt zu implementieren, müssen wir den Quellcode für die Fachlogik nicht anpassen. Gekapselt wird der Code in unserem Beispiel in der in Listing 7 angegebenen (zugegebenermaßen sehr einfachen) Interceptor-Klasse.

```
package profiling.aop;

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

public class SimpleProfilingInterceptor implements Interceptor {
  public String getName() { return "SimpleProfilingInterceptor"; }

  public Object invoke(Invocation invocation) throws Throwable {
    long before = -1L;
    String className = "";
    String methodName = "";

    if (invocation instanceof MethodInvocation) {
      MethodInvocation mi = (MethodInvocation)invocation;
      className = mi.getTargetObject.getClass().getName();
      methodName = mi.getMethod.getName();
    }

    try {
      System.out.println("profiling " + className + "->" + methodName);
      before = System.currentTimeMillis();
      return invocation.invokeNext();
    }
    finally {
      long after = System.currentTimeMillis();
      System.out.println("execution time: " + (after - before) + "ms");
    }
  }
}
```

Listing 7: SimpleProfilingInterceptor.java

Ein Interceptor in JBossAOP muss das Interceptor-Interface implementieren, d. h. er benötigt eine **getName**- und eine **invoke**-Methode. An dieser Stelle sei der Vollständigkeit halber erwähnt, dass man in JBossAOP auch Aspekte mit mehreren Advices definieren kann. In diesem Fall muss man lediglich eine Klasse schreiben, die pro Advice eine entsprechende Methode mit derselben Signatur wie die **invoke**-Methode des Interceptor-Interfaces enthält. Da in unserem Beispiel jedoch ein Aspekt mit einem Advice genügt, beschränken wir uns hier auf die Interceptor-Variante.

Die **invoke**-Methode hat als einzigen Parameter ein Objekt vom Typ **Invocation**. Ein **Invocation**-Objekt in JBossAOP entspricht im Prinzip einem Joinpoint. Demzufolge gibt es für verschiedenartige Joinpoints auch verschiedene Ausprägungen des **Invocation**-Objekts. Das **Invocation**-Objekt für einen Execution-Joinpoint ist vom Typ **MethodInvocation** und enthält neben einer Referenz auf das Ziel-Objekt alle weiteren Informationen über den Execution-Joinpoint, d. h. beispielsweise ein **java.lang.reflect.Method**-Objekt, das die abgefangene Methode beschreibt, und ein Array von **java.lang.Object**-Objekten, die die Werte der Aufrufparameter der abgefangenen Methode enthalten.

Interceptoren bilden in JBossAOP immer eine Kette, d. h. ein Interceptor sollte über die **invokeNext**-Methode des **Invocation**-Objekts den nächsten Interceptor aufrufen. Am Ende der Kette wird dann die eigentliche, abgefangene Methode des Ziel-Objekts aufgerufen. Unser **SimpleProfilingInterceptor** merkt sich zuerst den aktuellen Zeitstempel, gibt auf **Stdout** aus, welche Methode gemessen wird, und ruft dann den nächsten Interceptor in der Kette auf, was letztendlich zum Aufruf der zu messenden Methode führt. Im **finally**-Block wird ein neuer Zeitstempel geholt und schließlich die Differenz der beiden Zeitstempel auf **Stdout** ausgegeben. Natürlich ist ein derartiger Profiling-Interceptor nur dann sinnvoll, wenn sichergestellt ist, dass er als letzter Interceptor der Kette aufgerufen wird, da er sonst nicht nur die Laufzeit der abgefangenen Methode, sondern auch die Laufzeit der folgenden Interceptoren in der Kette messen würde. Gehen wir aber einfach mal davon aus, dass dies erfüllt ist.

Als nächstes muss eine „jboss-aop.xml"-Datei (s. Listing 8) bereitgestellt werden. Diese wird benötigt, um den Bytecode des Business-Layers unserer Anwendung mittels AOPC darauf vorzubereiten (d. h. zu manipulieren), dass alle Business-Methoden zur Laufzeit mit dem Profiling-Aspekt versehen werden können. Die hier angegebene Pointcut-Expression ist ein Beispiel für einen etwas komplexeren Ausdruck, der die Execution-Joinpoints aller öffentlichen Methoden aller Session-Bean-Implementierungsklassen definiert. Über die **instanceof**-Syntax werden alle Objekte gefiltert, die das Interface **javax.ejb.SessionBean** implementieren. Über logische Verknüpfungen wird erreicht, dass nur die vermeintlichen Business-Methoden und nicht die für das Profiling uninteressanten EJB Lifecycle-Methoden (**ejbCreate**, **setSessionContext**, ...) präpariert werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <prepare
   expr="execution(public * $instanceof{javax.ejb.SessionBean}->*(..)) AND
     !(execution(public * *->ejb*(..)) OR
       execution(public * *->setSessionContext(..)))"
  />
</aop>
```

Listing 8: jboss-aop.xml

So weit so gut! Stellt sich nur noch die Frage, wie zur Laufzeit unser Profiling-Interceptor aktiviert werden kann. Hierfür drängt sich in JBoss die Verwendung von JMX auf. Wir schreiben also eine XMBean (eine JBoss-spezifische Erweiterung von MBeans), die die nötigen Operationen zum Ein- und Ausschalten des Profiling-Aspekts bereitstellt. Mittels der Browser-basierten JMX-Console von JBoss bekommen wir auch gleich die passende Benutzeroberfläche zum Aufrufen der Operationen geschenkt.

Der Code für die XMBean ist in Listings 9 angegeben, alle JBoss-AOP-relevanten Passagen sind farblich hervorgehoben. In der **start**-Methode der XMBean wird ein neues **AdviceBinding**-Objekt angelegt, das als Parameter eine Pointcut-Expression bekommt. Die übergebene Pointcut-Expression definiert ganz offensichtlich mehr potenzielle Joinpoints als die Pointcut-Expression aus unserer „jboss-aop.xml"-Datei. Das macht in unserem Fall aber nichts, da nur maximal die Methoden abgefangen werden können, die auch mittels AOPC vorbereitet wurden. Anschließend wird an das erzeugte **AdviceBinding**-Objekt das **Class**-Objekt unseres Profiling-Interceptors gehängt. Die Implementierungen der Methoden **activateProfiling** und **deactivateProfiling** verwenden lediglich das JBossAOP-**AspectManager**-Singleton, um unser **AdviceBinding**-Objekt hinzuzufügen bzw. zu entfernen. Der Vollständigkeit halber werden in den Listings 10 und 11 noch die für das Deployment der vorgestellten XMBean notwendigen XML-Deskriptoren angegeben.

Erweitert man die vorgestellte MBean noch um ein Attribut zum Setzen der Pointcut-Expression und um Methoden zum Hinzufügen und Entfernen von Klassennamen für Interceptoren, bekommt man mit geringem Aufwand bereits eine relativ generische Aspect-Manager-MBean, mit der sich beliebige Aspekte an beliebigen Stellen im Code einer Anwendung dynamisch ein- und ausschalten lassen.

Ein Hinweis an dieser Stelle zu JBossAOP in Verbindung mit CMP Entity-Beans. Ab EJB 2.0 werden die zu persistierenden Felder einer Entity-Bean ja nur noch implizit über abstrakte Methoden definiert. JBoss 4 stellt für diese Methoden erst zur Laufzeit die Implementierung zur Verfügung. Aus diesem Grund ist es nicht möglich, mit JBossAOP Querschnittsfunktionalitäten an den Execution-Joinpoints für getter- und setter-Methoden von CMP 2.0 Entity-Beans einzufügen. Über das klassische Interceptor-Konzept von JBoss ist es aber trotzdem - auch ohne AOP-Mittel - möglich, sich in die Aufrufe aller

```
package profiling.jmx;

import org.jboss.aop.AspectManager;
import org.jboss.aop.advice.AdviceBinding;

public class SimpleProfilingAspectMBean {
  public static final String POINTCUT_EXPR =
    "execution(public * $instanceof{javax.ejb.SessionBean}->*(..))";

  private AdviceBinding binding = null;
  private boolean isActive = false;

  public SimpleProfilingAspectMBean () { }

  public String getName() {
    return "SimpleProfilingAspectMBean";
  }
  public void start() throws Exception {
    binding = new AdviceBinding(POINTCUT_EXPR, null);
    binding.addInterceptor(
      Class.forName("profiling.aop.SimpleProfilingInterceptor"));
    isActive = false;
  }

  public void stop() {
    deactivateProfiling();
    binding = null;
  }

  public boolean isProfilingActive() {
    return isActive;
  }

  public void activateProfiling() {
    if (isActive) return;
    AspectManager.instance().addBinding(binding);
    isActive = true;
  }

  public void deactivateProfiling() {
    if (!isActive) return;
    AspectManager.instance().removeBinding(binding.getName());
    isActive = false;
  }
}
```

Listing 9: SimpleProfilingAspectMBean.java

Methoden des Remote- bzw. Local-Interface einer EJB zu hängen, sodass diese Limitierung zu vernachlässigen ist.

Ergänzend zu dem Beispiel noch eine kurze Betrachtung des Laufzeit-Overheads, der durch das Vorbereiten von Klassen für dynamisches AOP entsteht. In den Beispielen von JBossAOP gibt es hierzu ein Bechmark-Programm, das u. a. jeweils die Laufzeit von 20 Millionen Aufrufen einer noop-Methode eines nicht-präparierten und eines präparierten POJOs misst. Auf einem Test-PC (Intel Pentium IV 1,3 GHz und 1 GB Hauptspeicher) ergaben sich hierbei unter Verwendung der JRE 1.5.0 b64 Laufzeiten von ca. 90 ms (nicht präpariert) und 680 ms (präpariert).

Der Aufruf einer für dynamisches AOP vorbereiteten Methode erzeugt also einen geringen Overhead, der jedoch bei einer Real-World Enterprise-Anwendung kaum ins Gewicht fallen dürfte, da hier die Laufzeiten maßgeblich durch Datenbank-Zugriffe bzw. Backend-Zugriffe geprägt werden. Es scheint also durchaus ein interessanter Ansatz zu sein, generell zentrale Klassen im Business-Layer einer Enterprise-Anwendung für dynamisches AOP vorzubereiten, sodass ihr Verhalten bei Bedarf zur Laufzeit mit verschiedenen Aspekten erweitert werden kann.

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

<!DOCTYPE mbean PUBLIC
  "-//JBoss//DTD JBOSS XMBEAN 1.0//EN"
  "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">

<mbean>
 <description>Simple Profiling-Aspect XMBean</description>
 <descriptors>
  <persistence persistPolicy="Never" />
  <currencyTimeLimit value="10"/>
  <state-action-on-update value="keep-running" />
 </descriptors>
 <class>profiling.jmx.SimpleProfilingAspectMBean</class>
 <constructor>
  <description>no-arg constructor</description>
  <name>profiling.jmx.SimpleProfilingAspectMBean</name>
 </constructor>
 <attribute access="read-only" getMethod="isProfilingActive" >
  <description>binding status of profiling-interceptor</description>
  <name>ProfilingActive</name>
  <type>boolean</type>
 </attribute>
 <operation>
  <description>start</description>
  <name>start</name>
  <return-type>void</return-type>
 </operation>
 <operation>
  <description>stop</description>
  <name>stop</name>
  <return-type>void</return-type>
 </operation>
 <operation>
  <description>bind profiling-interceptor</description>
  <name>activateProfiling</name>
 </operation>
 <operation>
  <description>unbind profiling-interceptor</description>
  <name>deactivateProfiling</name>
 </operation>
</mbean>
```

Listing 10: profiling-xmbean.xml

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

<!DOCTYPE server PUBLIC
  "-//JBoss//DTD MBean Service 3.2//EN"
  "http://www.jboss.org/j2ee/dtd/jboss-service_3_2.dtd">

<server>
 <mbean code="profiling.jmx.SimpleProfilingAspectMBean"
      xmbean-dd="META-INF/profiling-xmbean.xml"
      name="AOP:name=SimpleProfilingAspectMBean,type=XMBean"/>
</server>
```

Listing 11: jboss-service.xml

JBoss 4 Aspekte - AOP Services

JBoss 4 beinhaltet eine Reihe teilweise neuartiger Dienste, die auf JBossAOP aufsetzen. Darunter sind im Wesentlichen vordefinierte Aspekte zu verstehen, die über Annotations und Metadaten verwendet werden können. Im Folgenden werden die wichtigsten dieser neuen Aspekte kurz dargestellt.

JBoss AOP Transaction Demarcation überträgt die von EJBs bekannte Semantik bzgl. Transaktionen auf ganz normale Java-Klassen. Über Annotations können für Methoden und sogar für Felder von POJOs mit den bekannten J2EE-Schlüsselwörtern („Required“, „RequiresNew“, „Supports“, …) Transaktionssemantiken definiert werden. Eine

entsprechende Attributierung der Methode einer POJO-Klasse würde beispielsweise wie in Listing 12 aussehen.

```
public class POJO {
  /**
   * @@org.jboss.aspects.tx.Tx (org.jboss.aspects.tx.TxType.REQUIRED)
   */
  public void doSomething() {
    // ...
  }
}
```

Listing 12: Beispiel für explizite JBoss AOP Transaction Demarcation über Annotations für Java 1.4.x

Alternativ kann die Definition der Transaktions-Attribute auch implizit über die Datei „metadata-aop.xml" (s. Listing 13) erfolgen. Die Syntax hierfür wurde ganz bewusst an die Syntax der entsprechenden Definition von EJB-Transaktions-Attributen im „ejb-jar.xml"-Deployment-Deskriptor angelehnt.

```
<annotation tag="transaction" class=" POJO">
  <default>
    <trans-attribute>Required</trans-attribute>
  </default>
  <method name="doSomething">
    <trans-attribute>RequiresNew</trans-attribute>
  </method>
</annotation>
```

Listing 13: Beispiel für implizite JBoss AOP Transaction Demarcation über metadata-aop.xml

Was muss man sich aber unter einer Transaktionssemantik für POJOs vorstellen? Betrachten wir zur Klärung dieser Frage ein Java-Programm, in dem POJOs von mehreren Threads bearbeitet werden. Thread A startet durch Aufruf einer POJO-Methode implizit eine Transaktion. In dieser Transaktion werden mehrere Felder des POJOs nacheinander verändert. Versucht nun Thread B auf die Felder dieses POJOs lesend zuzugreifen, wird er – solange die Transaktion von Thread A noch nicht erfolgreich beendet wurde – nur die alten Werte der Felder lesen können (Isolation). Erst nach vollständiger Beendigung der Transaktion von Thread A sind alle Veränderungen an den Feldern des POJOs auch für andere Threads sichtbar (Atomicity). Wird während der Ausführung der Transaktion von Thread A eine Exception geworfen, wird die Transaktion automatisch zurück gerollt. Transaktionale POJOs haben also in multithreaded Umgebungen immer einen konsistenten Zustand.

Ganz ähnlich funktioniert auch *JBoss AOP Security*. Hier wird versucht, das von J2EE bzw. EJB bekannte deklarative, Rollen-basierte Sicherheitskonzept auf POJOs zu übertragen. Auch hier hat man sich weitestgehend an die bekannte „ejb-jar.xml"-Sytnax angelehnt.

Eine weitere sehr interessante Anwendung von JBossAOP stellt der *JBossCache* dar. Hier handelt es sich um einen hierarchischen, transaktionalen und verteilten Cache, der in JBoss 4 an verschiedenen Stellen intern verwendet wird. JBossCache kann aber auch direkt vom Anwendungsentwickler verwendet werden, um beispielsweise gecachte Anwendungsdaten im Cluster synchron bzw. konsistent zu halten. Für die AOP-basierte Version von JBossCache (TreeCacheAOP) müssen hierfür lediglich die Klassen der zu cachenden POJOs mittels AOPC vorbereitet werden. Liegt ein POJO erst einmal im Cache, werden alle Veränderungen an dem Objekt (über setter-Methoden oder über direkten Zugriff auf Member-Variablen) im Cluster propagiert. Will man das Propagieren von inkonsistenten Zuständen eines POJOs verhindern, kann man dies mittels Transaktions-Klammern erreichen. In diesem Fall wird der neue Zustand des POJOs erst nach einem Commit für andere Clients des Cache sichtbar.

Die JBossAOP-Aspect-Bibliothek enthält noch weitere interessante Aspekt-Implementierungen, auf die in diesem Rahmen jedoch nicht näher eingegangen werden kann, beispielsweise:

- JBoss AOP Transactional Locking,
- JBoss AOP Remoting bzw. Clustered Remoting,
- JBoss AOP Asynchronous Method-Invocations.

Alle genannten Aspekte haben eins gemeinsam. Sie übertragen Infrastrukturfunktionalität der Middleware auf einfache Java-Klassen, ohne dass der Anwendungsentwickler hierfür spezielle Interfaces implementieren bzw. komplexe APIs lernen muss. Die einzelnen Aspekte können später jederzeit – bei Bedarf – hinzugefügt oder entfernt werden, ohne dass der existierende Code einer Anwendung angepasst werden muss. Dies ist die Grundidee von „POJO-based Middleware"!

Fazit

Middleware stellt Infrastrukturdienste zur Verfügung. Diese stellen aus Anwendungssicht Querschnittsfunktionalität dar. Aus diesem Blickwinkel betrachtet, scheint der Einzug der Aspektorientierung in Middleware nur der nächste logische Schritt zu sein. JBoss Inc. geht mit JBoss 4 und JBossAOP genau in diese Richtung und bezeichnet die neue Version des freien Applikationsservers folgerichtig auch als Aspect-Oriented-Middleware. Mit JBossAOP lassen sich fachliche und technische Querschnittsbelange von J2EE-Enterprise-Anwendungen sauber kapseln. Wie wir gesehen haben, lässt sich mit relativ geringem Aufwand auch eine passende MBean zur Verwaltung eigener Aspekte schreiben.

JBossAOP stellt aber zugleich auch die Basis für neuartige Infrastrukturdienste dar, die auf ganz normale Java-Klassen (Stichwort: POJOs) angewendet werden können. Zielsetzung dieser Dienste ist im Wesentlichen die dahinter steckende Komplexität völlig transparent zu halten, sodass sich Anwendungsentwickler zu 100 % auf die Implementierung der Fachlogik konzentrieren können. Nach der Vision von JBoss Inc. sollte diese im Idealfall nur noch mit POJOs implementiert werden. Der Zugriff auf die für eine Enterprise-Anwendung relevanten Infrastrukturdienste soll ausschließlich über zuschaltbare Aspekte erfolgen. JBoss 4 ist der erste Schritt auf dem Weg zu dieser Vision von POJO-based Middleware. Man darf gespannt sein, wie die nächsten Schritte in diese Richtung aussehen werden.

Links

[ANT] http://ant.apache.org/
[JAD] Decompiler JAD, http://www.kpdus.com/jad.html
[JBOSS] JBoss-Web-Seite, http://www.jboss.org
[JSR-175] Java Specification Request 175,
http://www.jcp.org/aboutJava/communityprocess/review/jsr175/

Dirk Mascher ist Geschäftsführer der Accelsis Technologies GmbH und seit Jahren spezialisiert auf Architekturen verteilter, mehrschichtiger Systeme, insbesondere im J2EE-Umfeld. Als Software-Architekt und Projektleiter hat er mit seinen Teams verschiedenste Enterprise-Anwendungen für Großunternehmen und Behörden auf Basis von J2EE erfolgreich entworfen und umgesetzt. Dirk Mascher ist zertifizierter JBoss Consultant.
E-Mail: dirk.mascher@accelsis.de.