

## 8 Aspektorientierte Programmierung

Viele Programme vermischen die eigentliche Geschäftslogik mit eher technischem Code, wie beispielsweise Logging, Persistenz oder Aufrufe von entfernten Methoden.

4.0

Beim Paradigma der aspektorientierten Programmierung (AOP) wird versucht, diese Vermischung in ihre unterschiedlichen Aspekte (Geschäftslogik, Persistenz, Remote-Aufrufe oder Logging) zu zerlegen, und diese einzelnen Aspekte dann gesondert zu lösen. Der Entwickler schreibt im Idealfall nur noch die Geschäftslogik, während die technischen Aspekte dann damit verwoben werden. Mit diesem Vorgehen wird unter anderem versucht, die Wiederverwendbarkeit der Komponenten zu erhöhen.

Dieser Abschnitt soll insbesondere das Konzept grob erklären und Lust auf Mehr machen. Das Thema AOP ist, obwohl es immer noch in den Kinderschuhen steckt, sehr umfangreich und kann mehrere Bücher füllen (z.B. [Böh05]).

### 8.1 Was ist AOP?

Bill Burke erklärt AOP im Java Developers Journal (Dezember 2003) wie folgt:

*What is an Aspect? An Aspect is a common functionality that's scattered across methods, classes, object hierarchies, or object models. Functionality that your class or object model shouldn't be concerned about, functionality that doesn't belong as it's not what the object is all about. The AOP-ites like to call this type of functionality crosscutting concerns, as the behavior is cutting across multiple points in your object models, and yet is distinctly different from the classes it's crosscutting. AOP allows you to abstract and seamlessly componentize these concerns and apply them to your applications in a unique way that regular object-oriented programs cannot achieve very easily.*

Das folgende Beispiel soll dieses Vorgehen erläutern:

```
public int calc() {
    RemoteHome rh;
    RemoteObject ro;

    rh = new InitialContext().lookup("rh");
    ro = rh.create();

    int res = ro.doSomething();

    int ret = res * 4 +15 ;
    log.info("Resultat: " + ret);
    return ret;
}
```

Die eigentliche Geschäftslogik besteht darin, einen Wert zu holen und diesen mit einer bestimmten Formel zu verrechnen. Zusätzlich muss sich die *calc()*-Methode noch um das Auffinden eines anderen EJBs kümmern und das Ergebnis über einen Logger ausgeben. Damit wurden die Aspekte Logging und Remote-Aufruf mit der eigentlichen Geschäftslogik vermischt, so dass diese nicht mehr in anderen Projekten oder Umgebungen (einfach) wiederverwendet werden kann.

Diese Geschäftslogik könnte wie folgt formuliert werden:

```
public int calc () {
    ro = new RO();
    int res= ro.doSomething();
    int ret = res * 4 +15;
    return ret;
}
```

Der ursprüngliche Logger-Aufruf kann dann vom AOP-Framework hinzugefügt werden (Pseudocode):

```
<advice name="logReturn">
    log.info(returnValue);
</advice>
<aspect>
    @calc().return:
        use logReturn(return->value);
</aspect>
```

Die Trennung von eigentlicher Geschäftslogik und sonstigen Belangen wie Security, Logging etc. ist für JBoss nichts Neues, sondern wird für den EJB- und JMX-Teil bereits seit JBoss 2 durch die Interceptoren (siehe Abschnitt 4.6) realisiert. Neu ist nun, dass beliebige Java-Objekte (POJOs) bearbeitet werden können und dass die Code-Stücke granularer eingearbeitet werden können.

## 8.2 Begriffe

Bei der AO-Programmierung haben sich eine Reihe von neuen Begriffen etabliert – unter anderem (siehe auch [AOP]):

**Joinpoint** Ein beliebiger Punkt im Java-Code, an dem neue Funktionalität eingefügt werden kann. Dies können Methodenaufrufe oder Zuweisungen etc. sein.

**Pointcut** Ein oder mehrere Joinpoints, an denen ein Advice eingewebt wird.

**Advice** Code, der mit dem Originalcode verwoben werden soll.

**Aspect** Bezeichnung für Pointcut und Advice zusammen. In JBossAOP bezeichnet ein Aspect eine Java-Klasse, die beliebig viele Advices beinhaltet.

**Introduction** Eine Introduction führt ein neues Verhalten in eine Klasse ein. Dies kann die Implementierung eines bestimmten Interfaces sein oder auch das Hinzufügen einer Annotation<sup>1</sup>.

**Weaving** Das Verweben stellt das »Verheiraten« des Originalcodes mit den Aspekten dar. Diese Arbeit wird entweder vom Compiler beim Erstellen einer Applikation oder dem Laufzeitsystem beim Laden der Klassen erledigt.

**Interceptor** Dies ist ein Aspect, der nur den Advice namens Invoke kennt. Interceptoren sind im JBoss-Server schon seit langem enthalten.

## 8.3 AOP im JBossAS

AOP ist im Applikationsserver 4.0.0 in der *all*-Konfiguration enthalten. Wenn Sie AOP in anderen Konfigurationen nutzen wollen, reicht es aus, das Archiv *jboss-aop.deployer* aus dem *deploy/*-Verzeichnis der *all*-Konfiguration in das *deploy/*-Verzeichnis Ihrer Wunschkonfiguration zu kopieren. Ab JBoss 4.0.1 soll das Archiv immer auch in der *default*-Konfiguration zu finden sein.

4.0

JBoss kennt zwei AOP-Bearbeitungsweisen:

- ❑ Existierende Klassen werden beim Entwickeln mit den Aspekten verwebt. Dies geschieht über den AopC-Compiler, der nach dem Java-Compiler aufgerufen wird.

<sup>1</sup>Annotations sind ein neues Sprachmerkmal seit JDK 5.0 und wurden u.a. in JSR-175 beschrieben. Annotations ähneln Javadoc-Kommentaren im Aussehen.

- Das Verweben geschieht zur Laufzeit, wenn die Klassen vom Classloader geladen werden. Damit ist es möglich, innerhalb des Applikationsservers, Aspekte über Hot Deployment nachzuladen.

Die folgenden Abschnitte gehen vom Verweben der Aspekte zur Laufzeit aus. Der AopC-Compiler wird weiter unten vorgestellt.

### 8.3.1 jboss-aop.deployer

In diesem Archiv (siehe 4.5) liegen die für AOP benötigten Java-Archive, ein Archiv mit vordefinierten Aspekten und die Datei *jboss-service.xml*. In dieser wird insbesondere der AspectDeployer und der AspectManager definiert.

Der *AspectManager* hat die folgenden Attribute.

**EnableTransformer** Damit das AOP-Subsystem Klassen zur Laufzeit mit Aspekten versieht, muss dieses Attribut auf true gesetzt werden (siehe Listing 8.1).

**Optimized** Dieses Attribut steht immer auf true. Es dient dazu, die Optimierungen zur Fehlerverfolgung (der AOP-Bibliotheken) abschalten zu können. Dies ist allerdings für den Normalbetrieb nicht empfehlenswert, da die Ausführung des verwobenen Codes dann deutlich langsamer wird.

**SuppressTransformationErrors** Hiermit können Exceptions unterdrückt werden, die daher rühren, dass zu ladende Klassen nicht gefunden werden. Diese Exceptions können daher kommen, dass JBossAOP versucht, alle referenzierten Klassen zu laden, selbst wenn diese für das eigentliche Verweben nicht benötigt werden.

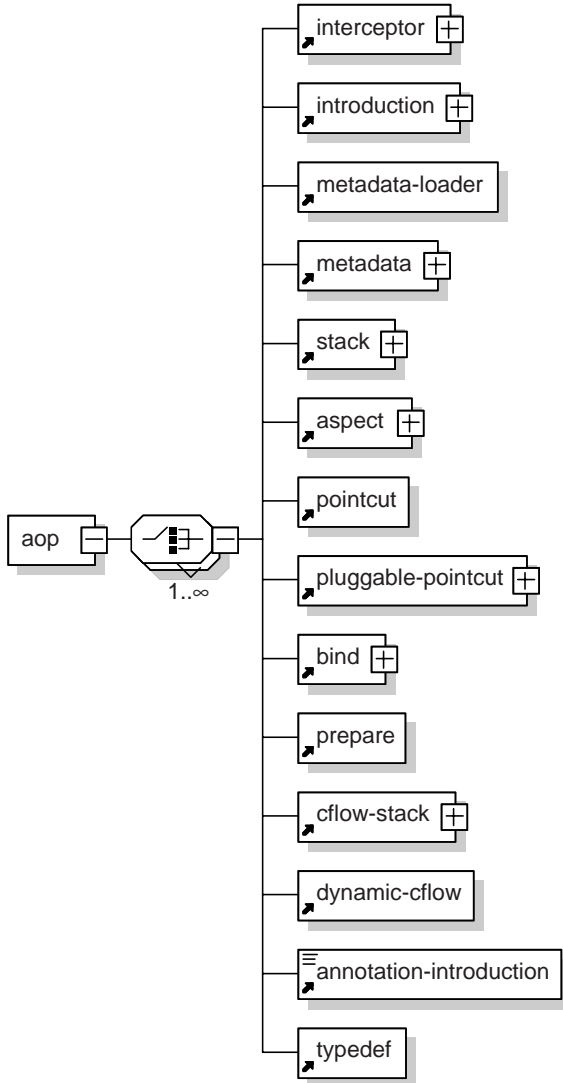
**Verbose** Steht dieses Attribut auf true, dann gibt das AOP-Subsystem alle Aktionen auf der Konsole aus. Dies ist hilfreich zur Fehlersuche, produziert aber sehr viel Text und verlangsamt die Operation deutlich.

Für Tests können die Attribute im MBean `jboss.aop:service=AspectManager` wie gewohnt via JMX gesetzt werden. Damit die Aspekte allerdings beim Systemstart mit deployt werden können, muss die Datei *jboss-service.xml* geändert werden.

**Listing 8.1** /jboss/bin\$ twiddle set "jboss.aop:service=AspectManager" \  
 Das Verweben EnableTransformer true  
 anschalten EnableTransformer=true

### 8.3.2 Der AOP-Deployment-Deskriptor

Der AOP-Deployment-Deskriptor folgt der DTD [http://www.jboss.org/aop/dtd/jboss-aop\\_1\\_0.dtd](http://www.jboss.org/aop/dtd/jboss-aop_1_0.dtd) und hat eine Grobstruktur wie in Abbildung 8.1 gezeigt.



**Abbildung 8.1**  
Der AOP-Deployment-Deskriptor,  
*jboss-aop.xml*

#### interceptor

Dieses Element beschreibt einen Interceptor und hat vier Attribute.

`class` Der voll qualifizierte Klassenname des Interceptors.

**factory** Alternativ zur Angabe einer Klasse eines Interceptors kann auch eine Factory-Klasse angegeben werden, die Interceptoren erzeugt. In diesem Fall werden in `<interceptor>` eingeschlossene Elemente an die Factory weitergereicht.

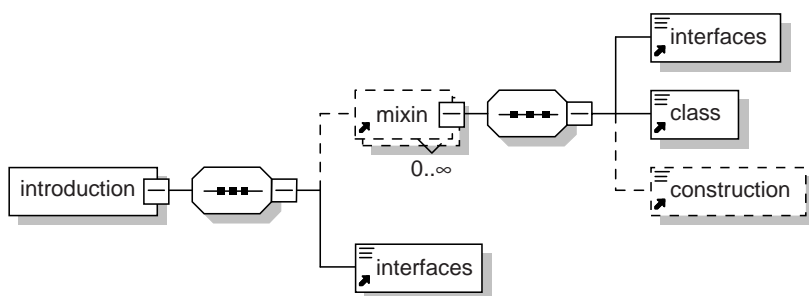
**name** Der optionale Name des Interceptors.

**scope** Der Einsatzbereich des Interceptors. Es gibt vier Werte (`PER_VM`, `PER_CLASS`, `PER_INSTANCE` und `PER_JOINTPOINT`), wobei `PER_VM` die Voreinstellung ist.

### introduction

Eine Introduction führt neues Verhalten in Klassen ein. Dies kann z.B. die Implementierung eines Interfaces sein. Man könnte also z.B. einer Klasse, die nicht als serialisierbar markiert wurde, das Interface *Serializable* hinzufügen.

**Abbildung 8.2**  
Struktur des  
introduction-Elements



Über `<interfaces>` werden die zu implementierenden Interfaces durch Kommata separiert angegeben.

Die Attribute *name* und *expr* des `<introduction>`-Elements können entweder die Namen von Klassen oder Ausdrücke, wie für Pointcuts, zur Ermittlung der zu bearbeitenden Klassen enthalten.

```
<introduction class="de.bsd.adb.beans.*">
  <interfaces>java.io.Serializable</interfaces>
</introduction>
```

Mittels des `<mixin>`-Tags kann neben dem Interface auch die dazugehörige Implementierung angegeben werden. Mit dem Element `<construction>` kann man dabei einen Ausdruck angeben, der in die Klasse mit dem Joinpoint direkt eingewoben wird und z.B. die Mixin-Klasse initialisieren kann.

## metadata

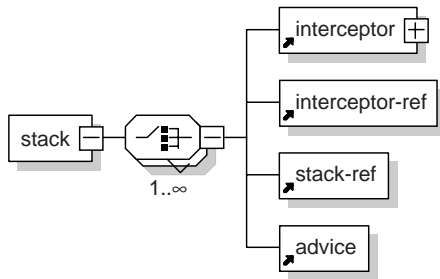
Über dieses Tag können Metadaten an die Klassen gegeben werden. Dabei gibt es eingebettete Elemente für Konstruktoren, Methoden, Felder und Voreinstellungen.

## metadata-loader

Wenn die Metadaten aus dem `<metadata>` nicht ausreichend sind, lassen sich diese auch über eine Java-Klasse laden.

## stack

Ein Stack ist eine vordefinierte Reihe von Interceptoren und Advices, die nacheinander aufgerufen werden und die aus dem `<bind>`-Element heraus referenziert werden können.



**Abbildung 8.3**  
Struktur des  
stack-Elements

Die eingebetteten Elemente von `<stack>` sind dieselben wie bei `<bind>` und werden dort erläutert. Das Tag `<stack>` hat das zwingende Attribut *name*, das den Stack identifiziert.

## aspect

Dieses Element dient zur Definition einer Klasse als Aspekt, damit diese dann innerhalb von `<bind>` referenziert werden kann. Ist das Attribut *name* vorhanden, gibt es den Namen an, über den der Aspekt angesprochen werden kann. Ist das Attribut nicht vorhanden, kann dies über den angegebenen Klassennamen geschehen.

```
<aspect class="some.class" name="superAspekt"/>
```

Innerhalb des Aspekts können auf JavaBean-Weise auch Attribute der Klasse gesetzt werden. Hierzu wird der entsprechende Setter des Attributs aufgerufen

```
<aspect class="some.class" name="superAspekt">
  <attribute name="aFloatVal">-3.2</attribute>
</aspect>
```

### pointcut

Über dieses Tag können Pointcuts definiert werden, die dann innerhalb anderer Pointcuts referenziert werden können.

```
<pointcut name="myClasses" expr="within(de.bsd.adb.*)"/>

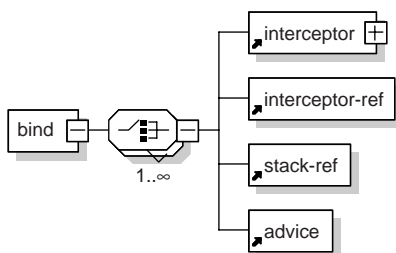
<bind pointcut="myClasses">
  <interceptor-ref name="..."/>
</bind>
```

### pluggable-pointcut

Über dieses Element können Pointcuts auch durch Java-Klassen definiert werden. Dabei müssen die beiden Attribute *name* und *class* vorhanden sein. Sie dienen dazu, einen Referenznamen und die zu verwendende Klasse anzugeben. Um als Pointcut in Frage zu kommen, muss die Klasse das Interface `org.jboss.util.xml.XmlLoadable` implementieren. Elemente, die als Subelement angegeben sind, werden an den Pointcut weitergeleitet.

### bind

**Abbildung 8.4**  
Das bind-Element



Bind selbst hat drei Attribute:

**name** Der (optionale) Name des Bindings.

**pointcut** Ein Pointcut, der die Bedingung angibt, wann die angegebenen Advices und Interceptoren ihre Arbeit verrichten sollen (siehe Listing 8.2).

**cflow** Der Name eines cflow-stacks, der als zusätzliche Bedingung dient. Dabei können cflows auch über »AND«, »OR« oder »!« verknüpft werden.

```
<bind pointcut="execution(de.bsd.adb.*->new())">
  <interceptor class="de.bsd.adb.aop.ADBAspect"/>
</bind>
```

**Listing 8.2**

*Interceptor, der bei Ausführung des Konstruktors aufgerufen wird*

Darüber hinaus kennt `<bind>` noch die eingebetteten Elemente

**interceptor** Die Klasse des Interceptors, der an den Pointcut gebunden wird.

**interceptor-ref** Eine Referenz auf einen bereits definierten Interceptor. Verwiesen wird über das *name*-Attribut entweder auf den Klassennamen des Interceptors oder auf den in dessen *name*-Attribut angegebenen Wert.

**stack-ref** Eine Referenz auf einen Stack.

**advice** Eine Referenz auf einen Advice.

```
<bind pointcut="foo">
  <advice name="trace" aspect="myAspect"/>
</bind>
```

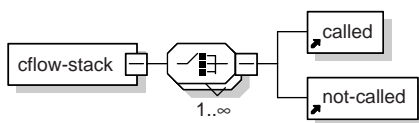
Bei Zutreffen des Pointcuts wird also die Methode *trace* der als *myAspect* benannten Klasse ausgeführt<sup>2</sup>.

**prepare**

Über `<prepare>` können Pointcut-Ausdrücke vorbereitet werden. Dies ist notwendig, wenn man JBossAOP über die Java-API<sup>3</sup> ansprechen möchte.

**cflow-stack**

Ein Cflow-Stack (Control-Flow-Stack) ist ein boolesches Element. Der Stack gibt *true* zurück, wenn der Control-Flow der analysierten Methode dem Flow des Stacks entspricht.

**Abbildung 8.5**

*Die Optionen von cflow-stack*

<sup>2</sup>Die Methoden können überschrieben sein. Es wird also die Methode ausgeführt, die als Parameter den passenden Typ von Invocation hat. Dies wird auch weiter unten im Beispiel genauer erläutert.

<sup>3</sup>Diese ist hier nicht erläutert. Es sei auf die JBoss-Doku verwiesen.

Hierbei gibt es Elemente, die zutreffen müssen (called) und die nicht zutreffen dürfen.

Dieses Beispiel illustriert die Verwendung.

```
<cfow-stack name="cstack">
  <called expr="* *->foo()"/>
  <not-called expr="* *->bar()"/>
  <called expr="* *->foo()"/>
</cfow-stack>
```

Bei diesem Stack würde für die Methode A() wahr geliefert und falsch für Methode B().

```
public void A() {                public void B() {
    foo();                        foo();
    foo();                        bar();
}                                  foo();
                                   }
```

### dynamic-cflow

Dieses Element definiert eine Funktionalität ähnlich der von `<cfow-stack>`. Allerdings wird der Flow hier nicht deklarativ angegeben, sondern durch die im Attribut `class` angegebene Klasse geprüft. Diese Klasse muss dafür das Interface `org.jboss.aop.pointcut.DynamicCFlow` implementieren.

### annotation-introduction

Über dieses Element können JDK-5.0-Annotations an passende Methoden, Konstruktoren oder Felder sowie an die Klasse selbst angebracht werden.

### typedef

Mittels `<typedef>` lassen sich Typen definieren, die dann in Pointcut-Ausdrücken referenziert werden können. Dabei sind die zu definierenden Ausdrücke selbst Pointcut-Ausdrücke, wobei nur die Actions `call`, `has` und `hasfield` zum Einsatz kommen können.

```
<typedef name="adb" expr="class(de.bsd.adb.*)"/>

<bind pointcut="within($typedef(adb))">
  ...
</bind>
```

### 8.3.3 Pointcut-Ausdrücke

Pointcuts definieren die Stelle im Code, an denen ein Advice eingewebt werden soll. Diese Stellen werden über reguläre Ausdrücke definiert. Ein solcher Ausdruck folgt dem Muster<sup>4</sup>:

```
pointcut ::= action ( kausdruck [logic kausdruck])
kausdruck ::= [attribut] [return] klassen [-> methode|feld ]
logic ::= AND|OR|!
```

#### methode|feld

Der Parameter methodelfeld gibt die Signatur einer Methode oder eines Konstruktors an bzw. den Namen eines Feldes. Konstruktoren werden über den Namen »new()« generisch angesprochen. Sowohl bei Methoden als auch bei den Konstruktoren können Wildcards angegeben werden. Wenn dieser Parameter weggelassen wird, darf auch der Pfeil nicht mit angegeben werden.

#### klassen

In diesem Teil werden die Klassen adressiert, auf die der Advice zutreffen soll. Hier gibt es noch das Konstrukt `$instanceof{Klasse}` das zutrifft, wenn die aktuelle Klasse eine Instanz der in geschweiften(!) Klammern angegebenen Klasse ist (letztlich genau wie in Java).

#### attribut

Das Attribut bezeichnet die Sichtbarkeit der Methode (public, private, static) und ist optional.

#### return

An dieser Stelle kann der Rückgabotyp der zutreffenden Methoden gesetzt werden. Auch hier ist der Stern als Wildcard möglich. Ausdrücke für Konstruktoren dürfen keinen Rückgabotyp haben.

#### action

Die Action bezeichnet die Bedingung, wann der nachfolgende Ausdruck im Fall der Übereinstimmung ausgeführt werden soll. Für jede Action ist ein kleines Beispiel angegeben, das sich auf diese Klasse bezieht.

---

<sup>4</sup>Dies ist keineswegs eine komplette Grammatik der Sprache. Diese ist als JJTree-Datei im Quellcode der JBossAOP-Distribution zu finden.

```
package de.bsd.aop;

public class Calc
{
    int result;
    String ops;

    public Calc() {
        result=0;
    }

    public Calc(int i) {
        result=i;
    }

    public add (int x, int y) {
        result = x + y;
    }

    public void print() {
        System.out.println(result);
    }
}
```

**all** Diese Aktion trifft auf alle Operationen der Klasse zu. Bei `all(de.bsd. aop.Calc)` wird der Advice bei jedem Zugriff auf die Klasse aufgerufen.

**call** Trifft für Methoden und Konstruktoren zu. Der Advice wird beim Aufruf des entsprechenden Ausdruck aufgerufen. Der Unterschied zu `execution` ist, dass bei `call` der Code des Aufrufers modifiziert wird und nicht der des Aufgerufenen. Damit lassen sich über `call` Aufrufe von Systemklassen modifizieren.

`call(*->add(int))` wird bei Aufruf von `add()` aufgerufen.

`call(*->add())` würde nicht aufgerufen, da es keine `add()`-Methode ohne Parameter gibt.

**class** Liefert `true`, wenn die aktuelle Klasse mit der angegebenen Klasse übereinstimmt `class(de.bsd.adb.*)`. Im Ausdruck kann auch `$instanceof` verwendet werden, um zu überprüfen, ob die aktuelle Klasse von der angegebenen Klasse erbt.

**execution** Der Advice wird ausgeführt, wenn der Ausdruck zutrifft und ausgeführt wird (also Aufruf der Methode oder des Konstruktors).

**field** Dies ist eine Zusammenfassung von `get` und `set`.

- get** Der Advice wird ausgeführt, wenn das Feld aus dem kausdruck gelesen wird. Dieser lesende Zugriff muss nicht über einen Getter geschehen. `get(* *->result)` würde also in `print()` aufgerufen werden.
- has** Diese Bedingung trifft zu, wenn die Klasse eine Methode oder einen Konstruktor hat, für den der kausdruck zutrifft. Sowohl `has(* *->foo())` als auch `has(* *->add())` ergibt also keinen Match, im Gegensatz zu `has(* *->add(..)`. Über *has* lässt sich auch überprüfen, ob eine bestimmte Annotation vorliegt.
- hasfield** Trifft für Klassen zu, die das angegebene Feld haben.
- set** Das Gegenstück zu `get`, welches schreibende Zugriffe auf das Feld abfängt.
- within** Trifft auf jeden Joinpoint im aufgerufenen Code zu. `within(de.bsd.aop.Calc)`
- withincode** Trifft auf jeden Joinpoint in der angegebenen Methode zu. `withincode(* *.Calc->print())`

## logic

Eine action kann sich auf einen kausdruck oder auf eine Verkettung von kausdrücken beziehen. Diese Ausdrücke können über »AND«, »OR« und »!« (für nicht) verknüpft werden. `has(@singleton) AND !class(com.acme.*)`

## Wildcards

Die Sprache kennt zwei Arten von Wildcards:

- ❑ »\*« Der Stern steht für null oder mehrere Zeichen und kann überall außer in Annotation-Ausdrücken stehen.
- ❑ ».« Die beiden Punkte stehen für eine beliebige Anzahl an Parametern in Methoden- oder Konstruktorsignaturen.

### 8.3.4 Was JBossAOP nicht kann

JBossAOP ist ein sehr mächtiges Subsystem, das aber auch seine Grenzen hat. Beispielsweise ist es nicht möglich, Joinpoints auf EntityBean-Finder oder andere Methoden in den Home Interfaces der EJBs zu legen. Das hat damit zu tun, dass es keine Klassen gibt, die die Home Interfaces direkt implementieren. Der EJB-Container erledigt dies dynamisch zur Laufzeit.

Wenn Methoden der Home Interfaces instrumentiert werden sollen, kann dies aber über die etablierten JBoss-Interceptoren geschehen, die in Abschnitt 4.6.2 beschrieben wurden.

## 8.4 Beispiel: Timing-Aspekt

Im Abschnitt über die klassischen JBoss-Interceptoren haben wir ein Beispiel gesehen, bei dem die Dauer der Aufrufe der Methoden im FacadeSessionBean aus Kapitel 3 über einen Interceptor ermittelt wurde. In diesem Beispiel wollen wir diese Funktionalität über AOP nachbauen.

### Implementierung als Interceptor

Die erste Implementierung erfolgt über einen Interceptor.

**Listing 8.3**  
*Timing-Aspekt als  
Interceptor,  
TraceAspect.java*

```
package de.bsd.adb.aop;

import java.lang.reflect.Method;
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jboss.logging.Logger;

public class TraceAspect implements Interceptor {

    Logger log = Logger.getLogger(TraceAspect.class);

    public String getName() {
        return "TraceAspect";
    }

    public Object invoke(Invocation inv)
        throws Throwable
    {
        long start, now;

        String meth;
        if (inv instanceof MethodInvocation) {
            MethodInvocation minv = (MethodInvocation) inv;
            meth= getClassDotMethod(minv.getMethod());
        }
        else {
            meth= "unknown ";
        }
    }
}
```

```
start = System.currentTimeMillis();
Object ret = inv.invokeNext();
now = System.currentTimeMillis();
log.info(meth + (now-start) + "ms");
return ret;
}
```

Der eigentliche Code ist dem aus Abschnitt 4.6.4 sehr ähnlich. Die hier verwendete Methode `getClassDotMethod()` ist dort auch bereits abgedruckt.

Die Konfiguration für diesen Interceptor kann wie folgt aussehen:

```
<aop>
  <interceptor name="adb-ins" class="de.bsd.adb.aop.TraceAspect"/>

  <bind pointcut="execution(* *->suche(..))">
    <interceptor-ref name="adb-ins"/>
  </bind>
</aop>
```

**Listing 8.4**

*jboss-aop.xml für den  
Timing-Aspekt als  
Interceptor*

Zunächst wird ein Interceptor mit dem Namen *adb-ins* definiert. Dieser wird dann über seine Referenz an den Joinpoint gebunden, der bei der Ausführung der Methode *suche()* eingewoben wird.

## Implementierung als Advice

Wie man im Deployment-Deskriptor *jboss-aop.xml* gesehen hat, kann ein Aspekt auch als Advice ausgeführt werden. Hier ist der Code sogar weniger restriktiv, und es gibt hier die Möglichkeit, den Advice zu überschreiben.

**Listing 8.5**

Timing-Aspekt als  
Advice, Advices.java

```
package de.bsd.adb.aop;

import org.jboss.aop.joinpoint.MethodInvocation;

public class Advices {

    public Object measure(MethodInvocation inv)
        throws Throwable
    {
        long start,now;

        start = System.currentTimeMillis();
        Object ret = inv.invokeNext();
        now = System.currentTimeMillis();
        System.out.println(inv.getMethod().getName() +
            ":" + (now-start) + "ms");

        return ret;
    }
}
```

Wie man sieht, ist dies eine ganz normale Java-Klasse, die von keinem spezifischen Interface mehr erbt. Methoden, die dem Muster `public Object mname(Invocation)` entsprechen, können als Advice benutzt werden. Der eigentliche Code des Advices ist dem des Interceptors gleich. Bei den Advices ist es, wie oben gesagt, auch möglich, die Methoden zu überschreiben. In unserer Klasse könnte es also auch einen Advice `measure(ConstructorInvocation)` geben, der bei einem Konstruktor-Joinpoint aufgerufen würde.

**Listing 8.6**

*jboss-aop.xml* für den  
Timing-Aspekt als  
Advice

```
<aop>
  <aspect name="myAdvices"
    class="de.bsd.adb.aop.Advices"/>

  <pointcut name="add2"
    expr="execution(* de.bsd.adb.ejb.*->add(..)"/>

  <bind pointcut="add2">
    <advice name="measure" aspect="myAdvices"/>
  </bind>
</aop>
```

Über das `<aspect>`-Element wird eine Klasse definiert, die Advices beinhaltet. Diese Advices können dann innerhalb von `<bind>` im `<advice>`-Element angesprochen werden. Dabei verweist das *name*-Attribut auf die entsprechende Methode in der Klasse, während das Attribut *aspect* die Referenz auf das *name*-Attribut des `<aspect>`-Tags hält. In diesem Beispiel wird auch gezeigt, wie ein Pointcut innerhalb eines `<pointcut>`-Elements definiert und von `<bind>` referenziert werden kann.

## 8.5 Der AopC-Compiler

Neben der Instrumentierung der Klassen zur Laufzeit bietet JBossAOP auch die Möglichkeit, die Klassen bereits zum Zeitpunkt des Kompilierens mit den Aspekten zu verweben. Hierzu gibt es den AOP-Compiler *AopC*. Dieser kann als Aufgabe in ant integriert werden.

```
<path id="aop.path.ref">
  <fileset dir="${jboss.aop.dir}">
    <include name="*.jar"/>
  </fileset>
</path>

<target name="aopc" depends="build-aop">
  <taskdef name="aopc"
    classname="org.jboss.aop.ant.AopC"
    classpathref="aop.path.ref"/>
```

**Listing 8.7**  
Ant-Integration von  
*AopC*

Wie immer, wenn eine neue Aufgabe in ant bekannt gemacht werden soll, muss der entsprechende Klassenpfad definiert sein, damit dieser in `<taskdef>` genutzt werden kann. Das Verzeichnis `${jboss.aop.dir}` zeigt dabei auf das Verzeichnis, in dem die ausgepackten Bibliotheken von JBossAOP liegen.

```
<aopc compilerclasspathref="aop.path.ref"
  verbose="false"
  report="true"
>
  <classpath>
    <pathelement
      location="${jboss.client.dir}/jbossall-client.jar"/>
    <pathelement
      location="${jboss.client.dir}/jboss-j2ee.jar"/>
    <pathelement path="${gen-class.dir}"/>
    <pathelement path="${interceptor-lib}"/>
```

```

        <pathelement path="${servlet.lib.path}"/>
        <pathelement location="server.path.ref"/>
        <pathelement path="${build.dir}/aop"/>
    </classpath>
    <src path="${gen-class.dir}"/>
    <aoppath
        path="srcs/de/bsd/adb/aop/jboss-aop.xml"/>

    <aopclasspath>
        <pathelement
            path="/d/jboss4/aop/jboss-aspect-library.jar"/>
    </aopclasspath>
    </aopc>
</target>

```

Der AopC-Compiler-Task hat einige eingeschlossene Elemente, welche die Orte der diversen Dateien benennen.

**classpath** Zusammen mit dem Attribut `compilerclasspathref` der Klassenpfad, in dem alle referenzierten Klassen vorhanden sein müssen.

**src** Pfad zu den kompilierten Original-Klassen, die instrumentiert werden sollen.

**aoppath** Der Fundort der Datei *jboss-aop.xml*. Zeigt der Pfad auf ein Verzeichnis, werden alle *\*aop.xml*-Dateien angezogen.

**aopclasspath** Der Pfad zu Bibliotheken mit »vorgefertigten« Aspekten.

Über das Flag *report* kann dem AopC-Compiler mitgeteilt werden, dass er zwar die Klassen nicht instrumentieren, dafür aber einen Report über unreferenzierte Pointcuts liefern soll. Damit dieser Report funktioniert, muss der Compiler doppelt aufgerufen werden: das erste Mal ohne die Report-Option, um die Klassen zu verweben, das zweite Mal mit der Report-Option, wie oben gezeigt, um die nicht verwendeten Pointcuts zu listen.

Wird der Compiler nur mit der Report-Option aufgerufen, ist zwar der Report der verwendeten Pointcuts nicht sehr genau, man erhält aber einen guten Syntax-Check für *jboss-aop.xml*, so dass man das *.aop*-Archiv nicht erst im Server deployen muss.

*Syntax-Checker für  
jboss-aop.xml*

## 8.6 JBossAOP standalone

Neben dem im JBoss-4-Applikationsserver vorhandenen AOP-Subsystem gibt es auch eine Stand-alone-Umgebung zum Download bei

---

SourceForge. Damit können beliebige Klassen in Java-Projekten instrumentiert werden.

Das Paket bietet neben der AOP-Funktionalität, die auch der Applikationsserver bietet, noch die Javadoc-API des Frameworks. Außerdem sind verschiedene Versionen der Bibliotheken vorhanden, um JDK 1.4 und JDK 5.0 zu unterstützen. Dies betrifft insbesondere die in JDK 5.0 eingeführten Annotations.