

Aspekt-Orientierte Programmierung (AOP)

von

Robert Switzer

1 Einleitung

Nehmen wir an, wir wollen die Metriken einer Anwendung messen und hätten keinen Profiler. Dann könnten wir jede unserer Klassen mit Code folgender Art ausstatten:

```
public class BankAccountDao
{
    public void withdraw(BigDecimal amount)
    {
        long startTime = System.currentTimeMillis();
        try
        {
            // Actual method body
        }
        finally
        {
            long elapsedTime = System.currentTimeMillis() - startTime;
            System.out.println("withdraw took: " + elapsedTime + " ms");
        }
    }
}
```

Dieser Code wird sicherlich funktionieren, aber es gibt hier einige Probleme, die sofort ins Auge springen:

1. es wäre sehr mühsam das Messen der Metriken an und abzuschalten, da wir einen solchen `try/finally`-Block in jede einzelne Methode unserer Anwendung einflücken müßten. (Hier empfindet man es als Manko, daß Java keine bedingte Compilation vorsieht.)
2. der Profilierungscode gehört eigentlich nicht in unsere Klassen, denn er hat nichts mit unserer Anwendungslogik zu tun.
3. wollten wir noch dazu Sicherheitskontrollen einführen, um zu verhindern daß Unbefugte unsere Methoden ausführen, müßten wir den ganzen Zirkus wiederholen.

Die Idee bei AOP ist, daß man zusätzliche Codeschichten um seine Methoden herumwickelt – und zwar transparent (also ohne den eigentlichen Code ändern zu müssen).

Es gibt hier zwei grundsätzlich verschiedene Vorgehensweisen:

- zur Compilierzeit wird der zusätzliche Code von einem speziellen Compiler nach der eigentlichen Compilation zu dem Bytecode der Klasse hinzugefügt
- während die Klasse zur Laufzeit von der JVM geladen wird, wird der zusätzliche Code in den Bytecode der Klasse eingefügt (*eingewoben*, wie die AOPler sagen).

In der Anfangszeit des AOP (die ja nicht allzu weit zurückliegt) beschrieb man die gewünschten Modifikationen in einem XML-Dokument – etwa `aop.xml` – das von einem speziellen Compiler gelesen und umgesetzt wurde. Heute nach der Einführung der Annotationen im JDK 1.5 kann man auf das XML-Dokument verzichten und die durchzuführenden Modifikationen durch Anbringung geeigneter Annotationen in seinem Klassentext spezifizieren; dadurch entfällt der nachgeordnete Compilationsschritt.

Bemerkungen. Die Lösung mit Annotationen ist nicht unbedingt vorzuziehen:

- unser Code wird dann mit Sprachelementen “verunziert”, die mit der Anwendungslogik nichts zu tun haben.
- es ist dann nicht so einfach, Aspekte an- und abzuschalten.

2 Einige Definitionen

Hier will ich einige Definitionen von grundlegenden Begriffen bringen, die dieses Gebiet prägen:

- die sekundären Anliegen (z. B. Profiling, Sicherheitskontrollen etc.), von denen ich eingangs sprach, werden in der AOP-Literatur oft *crosscutting concerns* genannt, da sie in der Regel orthogonal zum eigentlichen Zweck einer Klasse liegen.
- der Punkt im Code, wo zusätzlicher Code eingefügt werden soll, heißt *Point-cut*.
- der einzufügende Code nennt man *Advice*.
- die Kombination aus Point-cut plus Advice heißt dann *Aspekt*.

Diese Begriffe werden in unseren Beispielen immer wieder auftauchen.

3 AOP-Frameworks

Es gibt diverse AOP-Frameworks, meistens im OSS-Bereich:

AspectJ dies ist eines der ältesten und galt bis vor kurzem als am meisten ausgereift – wenn nicht ganz unproblematisch.

Wenn man allerdings die Web-Seite von AspectJ besucht, bekommt man den Eindruck, daß dieses Projekt 2004 eingeschlafen ist.

Aspectwerkz die meisten, die AspectJ aus dem einen oder anderen Grund abgelehnt haben, haben Aspectwerkz vorgezogen.

Besucht man die Web-Seite von Aspectwerkz, erfährt man den Grund, warum AspectJ scheinbar eingeschlafen ist: die beiden Teams haben beschlossen, ihre Kräfte zu vereinen und ein gemeinsames Framework zu entwickeln.

JBoss-AOP auch die JBoss-Mannschaft hat relativ früh die Bedeutung von AOP erkannt. JBoss-AOP ist heute ein zentraler Baustein von der EJB3-Implementation bei JBoss – genau wie Hibernate.

Nanning die Web-Seite von Nanning beschreibt dies als “kleines aber erweiterbares” Framework.

Man muß wohl nicht lange raten, um zu tippen, welches dieser Frameworks ich in diesem Vortrag einsetzen werde.

4 Ein erstes Beispiel

Wir greifen unser Beispiel mit den Metriken vom ersten Abschnitt nochmal auf und wollen sehen, wie wir dieses Problem mit AOP besser lösen können.

Hierzu setze ich (wie der Leser wohl vermutet haben wird) JBoss-AOP ein. Ich habe mich nicht sehr intensiv mit AspectJ/Aspectwerkz beschäftigt. Es kann sein das das gemeinsame Framework dieser beiden Teams leichter zu erlernen ist als JBoss-AOP. Bei JBoss-AOP ist die sogenannte Lernkurve relativ steil, aber ich habe die Erfahrung gemacht, daß sich die Mühe lohnt. JBoss-AOP ist erstaunlich flexibel; man kann z. B. Annotationen mit Einweben der Aspekte sowohl zur Kompilations- als auch zur Laufzeit kombinieren.

Bemerkung. Man braucht den Anwendungsserver von JBoss nicht, um JBoss-AOP einzusetzen; man kann eine selbständige Version (standalone) von JBoss-AOP herunterladen. Mit dieser kann man Anwendungen, die mit Aspekten angereichert sind, wie normale Javaanwendungen ausführen; man muß nur dafür sorgen, daß die JBoss-AOP-jardateien im Classpath sind.

Hier ist unsere `BankAccountDao`-Klasse nochmal, nunmehr von dem fachfremden Profilierungscode bereinigt:

```
package banking;

import java.math.*;

public class BankAccountDAO
{
    public void    withdraw(BigDecimal amount)
    {
        System.out.println("withdrawing amount " + amount);
    }
}
```

Und hier ist die Klasse, die unseren Profilerungsadvice implementiert:

```
package interceptor;

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

public class MetricAOPInterceptor implements Interceptor
{

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

    public Object invoke(Invocation inv) throws Throwable
    {
        MethodInvocation invocation = (MethodInvocation) inv;
        Object result = null;
        long startTime = System.currentTimeMillis();

        try
        {
            result = invocation.invokeNext();
        }
        finally
        {
            long elapsedTime = System.currentTimeMillis() - startTime;
            System.out.println(invocation.getMethod().getName() +
                               " took " + elapsedTime + "ms");
        }
        return result;
    }
}
```

Kommentar:

1. Ein Interceptor ist eine Klasse, die nur einen einzigen Advice mit Signatur

```
    public Object invoke(org.jboss.aop.joinpoint.Invocation inv)
        throws Throwable
```

implementiert. Dabei transportiert das Argument vom Typ `Invocation` Informationen über dasjenige Feature (Methode, Konstruktor, oder Attribut), das den Pointcut

darstellt. Das zurückzugebende Objekt muß das Ergebnis des tatsächlichen Aufrufs sein.

2. Es kann eine ganze Kette oder einen ganzen Stapel von Interzeptoren geben, deren `invoke`-Methoden der Reihen nach ausgeführt werden.

Die Klasse `Invocation` hat eine Methode

```
public Object invokeNext()
```

mit der die `invoke`-Methode des nächsten Interzeptors in der Kette aufgerufen wird; ist der betreffende Interzeptor der letzte in der Kette, bewirkt `invokeNext`, daß die eigentliche Methode des Pointcut aufgerufen wird. Vergißt der Programmierer eines Interzeptors `invokeNext` aufzurufen, stockt die Abarbeitung der Interzeptor-Kette und die eigentliche Methode wird nie ausgeführt.

3. Die Klassenhierarchie von `Invocation` sieht so aus:

```
Invocation
  ConstructorInvocation
  FieldInvocation
    FieldReadInvocation
    FieldWriteInvocation
  MethodInvocation
```

Die Klasse `MethodInvocation` hat eine Methode

```
public java.lang.reflect.Method getMethod()
```

mit der man alle Informationen über die Pointcut-Methode holen kann. Die restlichen Subklassen von `Invocation` haben entsprechende `get<...>`-Methoden.

In unserer `finally`-Klausel machen wir gebrauch von dieser Möglichkeit.

Wenn man nicht mit Annotationen arbeitet, verlangt JBoss-AOP einen Deskriptor

```
jboss-aop.xml
```

Man kann Annotationen sogar mit einem solchen Deskriptor kombinieren, um optimale Flexibilität zu erreichen.

Hier ist der Deskriptor für unser Beispiel:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
  <bind pointcut=
    "execution(public void banking.BankAccountDAO->withdraw( \
    java.math.BigDecimal))">
    <interceptor class="interceptor.MetricAOPInterceptor"/>
  </bind>
</aop>
```

Kommentar:

1. Das Wurzelement des Deskriptors heißt stets `aop`.
2. Das Wurzelement kann eine ganze Reihe von Subelementen besitzen – darunter `bind`.
3. Ein `bind`-Element wird benutzt, den Advice eines Aspect zu binden oder einen Interzeptor an einen spezifischen *joinpoint*, wie in unserem Beispiel.

Ein `bind`-Element muß ein Attribut `pointcut` besitzen, dessen Wert einen oder mehrere Pointcuts beschreibt. Hierzu bietet JBoss-AOP eine ausdrucksstarke *Pointcut and Type Expression Language*. In unserem Beispiel spezifizieren wir nur die eine Methode

```
BankAccountDAO.withdraw(BigDecimal amount)
```

5 Ein besseres Beispiel

In unserem ersten Beispiel hatte unsere Klasse `BankAccountDAO` nur eine einzige Methode `withdraw` und bei der Definition unseres Aspect in `jboss-aop.xml` haben wir exakt diese eine Methode als Pointcut spezifiziert.

Das ist unrealistisch. In der Regel hat man viele Methoden verteilt über mehrere Klassen, deren Metriken man messen möchte. In unserem zweiten Beispiel wollen wir sehen, wie man das machen könnte.

Diesmal geben wir unserer Klasse `BankAccountDAO` mehrere Features (Attribute und Methoden), deren Metriken wir evtl. messen möchten:

```
package banking;

import java.math.*;

public class BankAccountDAO
{
    private BigDecimal balance = null;

    public BankAccountDAO()
    {
        System.out.println("initializing account with 0.00");
        balance = new BigDecimal("0.00");
    }

    public void deposit(BigDecimal amount)
    {
        System.out.println("depositing amount " + amount);
        balance = balance.add(amount);
    }
}
```

```

    }
    public void withdraw(BigDecimal amount)
    {
        System.out.println("withdrawing amount " + amount);
        balance = balance.subtract(amount);
    }

    public BigDecimal getBalance()
    {
        return balance;
    }
}

```

Unsere Interzeptorklasse muß etwas komplexer werden, da wir nicht mehr davon ausgehen dürfen, das der Pointcut die eine Methode `withdraw` ist. Hier ist unsere neue Interzeptorklasse:

```

package interceptor;

import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.*;
import java.lang.reflect.*;

public class MetricAOPInterceptor implements Interceptor
{
    public String getName()
    {
        return "MetricAOPInterceptor";
    }
    public Object invoke(Invocation inv) throws Throwable
    {
        Object result = null;
        long startTime = System.currentTimeMillis();

        try
        {
            result = inv.invokeNext();
        }
        finally
        {
            long elapsedTime = System.currentTimeMillis() - startTime;
            System.out.println(describePointcut(inv) +
                " took " + elapsedTime + "ms");
        }
        return result;
    }
}

```

```

private String describePointcut(Invocation inv)
{
    String result = null;

    if (inv instanceof MethodInvocation)
    {
        MethodInvocation invocation = (MethodInvocation) inv;
        Method method = invocation.getMethod();
        result = describeMethod(method);
    }
    else if (inv instanceof ConstructorInvocation)
    {
        ConstructorInvocation invocation = (ConstructorInvocation) inv;
        Constructor constructor = invocation.getConstructor();
        result = describeConstructor(constructor);
    }
    else if (inv instanceof FieldInvocation)
    {
        FieldInvocation invocation = (FieldInvocation) inv;
        Field field = invocation.getField();
        if (invocation instanceof FieldReadInvocation)
        {
            result = describeFieldGetter(field);
        }
        else
        {
            result = describeFieldSetter(field);
        }
    }
    return result;
}

private String describeMethod(Method m)
{
    return m.toString();
}

private String describeConstructor(Constructor c)
{
    return c.toString();
}

private String describeFieldGetter(Field f)

```

```

    {
        /*
         * Use reflection on f to construct a string
         * describing the signature of get<F>.
         */
    }

    private String describeFieldSetter(Field f)
    {
        /*
         * Use reflection on f to construct a string
         * describing the signature of set<F>.
         */
    }
}

```

Und nun brauchen wir einen neuen Deskriptor, der uns erlaubt, alle Features unserer Klasse zu profilieren. Hier ist ein solcher Deskriptor:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
    <bind pointcut="all(banking.*)">
        <interceptor class="interceptor.MetricAOPInterceptor"/>
    </bind>
</aop>

```

Kommentar: Hier mußten wir lediglich den Pointcut-Ausdruck gegen einen austauschen, der sämtliche Features aller Klassen im Package **banking** spezifiziert.

6 Annotationen einsetzen

Nun wollen wir wie wir die Annotationen verwenden können, die Java 5.0 ermöglicht hat. Von unseren drei Klassen ist lediglich die Interzeptorklasse betroffen; sie sieht nun so aus:

```

package interceptor;

import org.jboss.aop.Bind;
import org.jboss.aop.InterceptorDef;
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.*;
import java.lang.reflect.*;

@InterceptorDef
@Bind(pointcut = "all(banking.*)")
public class MetricAOPInterceptor implements Interceptor

```

```
{
    // The rest is unchanged.
}
```

Kommentar:

1. Die Annotation

```
@InterceptorDef
```

signalisiert JBoss-AOP, daß die folgende Klasse einen Interzeptor im Sinne von JBoss-AOP definiert.

2. Die Annotation

```
@Bind(pointcut = "all(banking.*)")
```

bindet unseren Advice an eine Menge von Pointcuts, die wir mit dem Pointcut-ausdruck in runden Klammern spezifizieren. Wir können nun auf den Deskriptor verzichten, denn alle Informationen, die er enthalten hätte, befinden sich nun in unserem Klassentext.

3. Damit wir über diese zwei Annotationen verfügen können, müssen wir die zwei import-Klauseln

```
import org.jboss.aop.Bind;
import org.jboss.aop.InterceptorDef;
```

hinzufügen. Sie nennen die Klassen, in denen diese Annotationen definiert sind.

7 Schlußbemerkung

So schön die hier gezeigten Beispiele sind, ich habe einige Bedenken, die ich nicht verschweigen will:

- Das Beispiel mit den Metriken ist in gewisser Weise ein optimales Beispiel, denn Metriken sind wirklich ein Aspekt (Wortspiel beabsichtigt!), der orthogonal zur Anwendungslogik ist. Es schadet der eigentlichen Anwendung in keiner Weise, wenn dieser Aspekt hinzugefügt oder aber wieder weggenommen wird.
- AOP wird häufig als der Königsweg gepriesen, Methoden mit Sicherheitskontrollen oder Transaktionalität zu versehen. AOPler argumentieren, daß solche Aspekte nicht *wirklich* relevant zur eigentlichen Anwendungslogik sind. Aber eine Anwendung, die Sicherheitskontrolle oder Transaktionalität benötigt, ist in einem tieferen Sinn "kaputt", wenn man diese fortläßt.

Da finde ich den Weg, den EJB einschlägt, schon sinnvoller. Bei AOP werden diese wichtigen Aspekte gewissermaßen durch die kalte Küche eingeführt.

- Es gibt leider keinen AOP-Standard (z. B. kein entsprechendes JSR); das führt dazu, daß jedes Framework sein eigenes Süppchen kocht.

So schön das eine oder andere Framework sein mag, es dürfte sehr mühevoll sein eine Anwendung, die mit dem einen Framework implementiert wurde, auf ein anderes Framework zu portieren.

Der JBoss-Anwendungsserver ist J2EE 1.4-konform und wird voraussichtlich J2EE 1.5-konform sein, wenn J2EE 1.5 freigegeben wird. So müßte es relativ einfach sein, eine EJB-Anwendung, die man für JBoss entwickelt hat, auf, sagen wir, BEA's WebLogic zu portieren.

Aber ich möchte meine Beispiele aus diesem Vortrag wirklich nicht auf Aspectwerkz portieren müssen!

Literatur

- [aop] <http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html>. Hier findet man einen guten Artikel über AOP.
- [jbaop] <http://www.jboss.com/aop/1.3/aspect-framework/userguide/en/html/index.html>. Dies ist die Benutzeranleitung für die JBoss-Implementation von AOP.