

**Marco Schulz, 3INF2**

**Aspektororientierte Programmierung**  
**AOP**  
**Eine Einführung**

© 2006 banaaloMEDIA, Fach: Softwarearchitekturen, HS Merseburg

## Inhaltsangabe

1. Motivation	Seite 2
2. Begriffsdefinition	Seite 3
2.1 Aspekt	
2.2 Crosscutting Concerns	
2.3 Weaving	
2.4 Join-Point	
2.5 Pointcut	
2.6 Advice	
3. Technologie	Seite 5
3.1 Einsatzmöglichkeiten	Seite 5
3.2 Vorgehensweisen	Seite 6
3.3 Syntax	Seite 7
3.3.1 Aspekt	Seite 7
3.3.2 Pointcut	Seite 7
3.3.3 Advice	Seite 8
4. Beispiel in AspectJ	Seite 8
5. Resümee	Seite 12
6. Literaturverzeichnis	Seite 13
6.1 Internetressourcen	Seite 13

Der vorliegende Aufsatz gibt einen Einblick in die Aspektorientierte Programmierung und stellt die verwendete Technologie vor. Es versteht sich, dass auf den folgenden Seiten die Thematik nicht erschöpfend behandelt werden kann. Mir ist es wichtig einen in sich geschlossenen Überblick zu schaffen.

Dieser Text soll alle notwendigen Informationen, für einen Einstieg in die Aspektorientierte Programmierung, bereitstellen und zugleich soll die Lust auf eine tiefer gehende Auseinandersetzung mit dem Thema gegeben werden.

## **1. Motivation**

Mittlerweile hat die Informationstechnologie beachtliche Weiterentwicklungen hervorgebracht, die sich auf den verschiedensten Sektoren bewegen. Schnellere Hardware und höhere Dimensionen an Speicher waren eine Grundvoraussetzung für leistungsfähigere Computerprogramme. Die geschaffenen Ressourcen sorgen aber auch gleichzeitig für eine höhere Komplexität in ihren Anwendungen.

Um Software innerhalb kurzer Zeit in mittelgroßen bis großen Teams produzieren zu können, bedarf es leistungsfähiger Werkzeuge. Ein solches Werkzeug ist unter anderem die Art der Programmierung, auch das Programmierparadigma genannt. Denn dies entscheidet grundlegend über den Entwurf und die Vorgehensweise für das Projekt. Diese Analysen im Vorfeld führen zu sichereren Prognosen, besonders wenn während der Machbarkeitsstudie über ein geeignetes Denkmodell verfügt werden kann. Durch einen hohen Abstraktionsgrad ist man zu dem auch in der Lage immer wiederkehrende Aufgaben allgemein gültig darzustellen.

Alleine eine Modularisierung der Komponenten zeigte, dass dieser Schritt zwar notwendig ist aber dennoch nicht ausreicht. Das Modell komplizierte Vorgänge in kleine und somit besser beherrschbare Teilprobleme zu zerlegen findet ihren Ursprung bei dem französischen Philosophen und Mathematiker René Descartes. Der durch die Modularisierung erzielte Abstraktionsgrad ist verglichen mit heutigen Techniken eher schwach. Dieser bildet jedoch die Basis für alles weitere.

Um dieses Modell zu erweitern, wurde schon sehr früh die funktionale Programmierung entwickelt. Dadurch war ein Entwickler in der Lage wiederkehrende Problemstellungen, die sehr stark miteinander verwandt sind, zu einer Problemdomäne zusammen zu fassen. Dieses Konstrukt wird als Funktion bezeichnet. Somit konnte nun erstmalig Code in einer Weise formuliert werden, der auch mehrmals zu verwenden ist. Dabei handelte es sich um einfache Primitiven. Diese Primitiven sind Funktionssammlungen, die thematisch sortiert in Dateien gespeichert werden. Die daraus resultierende Zeitersparnis ist nicht unerheblich und stellt einen bedeutenden wirtschaftlichen Faktor dar.

Im Jahr 1995 stellte das Unternehmen SUN Microsystems erstmalig die Sprache JAVA vor. Diese Sprache unterstützt die Objektorientierte Programmierung [OOP] konsequent und ist mittlerweile zum Standard geworden.

Die Vorstellung von Objekten mit ihren Eigenschaften und Methoden findet unter anderem ihren Ursprung in der Evolutionstheorie von Charles Darwin.

Die Methoden eines Objektes setzen dessen Eigenschaften. Dieser Formalismus beschreibt das Verhalten des Objektes ausreichend, aber nicht vollständig. Der Grad der Abstraktion ist enorm hoch und ermöglicht somit die Erstellung allgemein gültiger Baupläne für Objekte, die in unterschiedlichem Kontext wiederverwendet werden können.

Daher kann man sagen, daß jede Art [Objekt] durch ihre typischen Eigenschaften beschrieben wird und somit auch zu klassifizieren ist. Erst das Benennen der Eigenschaften macht das beschriebene Objekt einzigartig.

Wie auch in der Realität besteht zwischen den unterschiedlichen Objekten der virtuellen Welt eine Beziehung. Dadurch treten die beteiligten Objekte in Wechselwirkung zueinander und beeinflussen sich gegenseitig in ihrem Verhalten.

Ein klassisches Beispiel für diese Problematik ist das "Logging". Der Logger protokolliert jede Aktion des aufgerufenen Objektes. Betrachten wir die Klasse Kunde, die die beiden Methoden Registrierung und Bestellung besitzt. Für die Gestaltung eines Tests, ist es hilfreich zu wissen, unter welchen Bedingungen die beiden Methoden aufgerufen werden.

Ohne die Aspektorientierte Programmierung [AOP] wird die gesamte Logik des Loggers in der Klasse Kunde verstreut. Die horizontale Dekompression [WAM03, S.6] der Logik wird durch die AOP behandelt. So besteht nun die Möglichkeit die komplette Funktionalität des Loggers in eine eigene Klasse zu kapseln. Die Erweiterung der OOP durch Aspekte führt somit zur AOP.

Ein erheblicher Vorteil ist, dass bei konsequenter Verwendung bekannter Techniken, aus der OOP für den Softwareentwurf, generalisierte Klassen entstehen und somit der Code effizienter wieder verwendet werden kann.

Für die Umstellung neuer Projekte ist es sinnvoll nicht gleich die gesamte mögliche Funktionalität im entstehenden Code zu berücksichtigen. Dies führt zur Unübersichtlichkeit und erzeugt hohe Kosten. Die Funktionalität ist nach Bedarf mit den hinzukommenden Anforderungen zu erweitern. Auf diese Weise entsteht eine umfangreiche und mächtige Bibliothek.

## 2. Begriffe

Im Folgenden werden Begrifflichkeiten geklärt, die für die weitere Betrachtung erforderlich sind.

### *Aspekt*

Ein Aspekt ist ein Modul, das die Logik für eine mehrdimensionale Problemdomäne in eigenständige Teilprobleme trennt. Der Aspekt beschreibt von einem oder mehreren Objekten deren Interaktionen und löst die Problematik der Crosscutting Concerns.

### *Crosscutting Concerns*

Objekte bzw. Anwendungen die so stark miteinander verwoben sind, dass eine strikte Trennung ohne die Technologie der AOP nur schwer oder gar nicht möglich ist, werden als Crosscutting Concerns (sich schneidende Interessen) bezeichnet.

### *Weaving*

Als Weaving (einwerfen) wird der Vorgang bezeichnet, der die Aspekte mit Ihren Objekten verknüpft. Das Weaving übernimmt ein Compiler der für die gewählte Sprache die AOP unterstützt.

### *Join-Point*

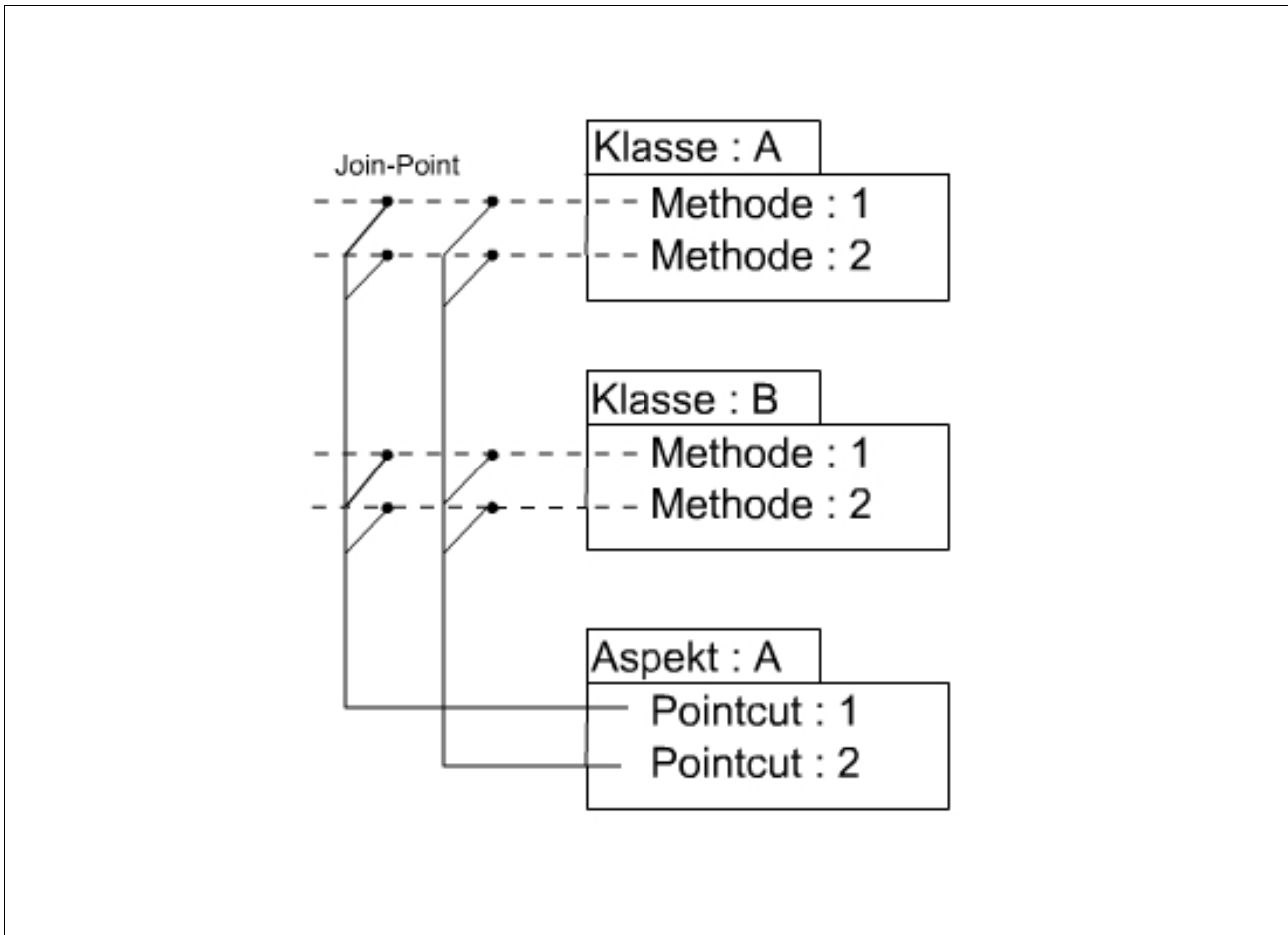
Ein Join-Point ist ein identifizierbarer Punkt, bei der Ausführung eines Programmes. Der Join-Point kann z. B. ein Methodenaufruf sein.

### *Pointcut*

Ein Pointcut ist ein Programmkonstrukt, welches verschiedene Join-Points auswählt und sie nach Ihrem Kontext in einen Zusammenhang bringt.

### Advice

Im Zusammenhang mit der Sprache AspectJ, ist ein Advice (übersetzt: Rat) der Programmcode der an einem Pointcut ausgeführt wird.



**Bild1:** Stellt den Zusammenhang von Klassen und Aspekten dar. Besonders der Unterschied zwischen Join-Point und Pointcut ist hier gut zu erkennen.

Um den Sachverhalt der Definitionen auf ein konkretes Beispiel zu projizieren, betrachten wir die eingangs vorgestellte Problematik eines Loggers.

Alle Methoden der Klasse A und der Klasse B [Bild 1] sollen beim Betreten und beim Verlassen des Loggers protokolliert werden. Der im Bild 1 dargestellte Aspekt ist nun äquivalent zum Logger.

Man kann sagen, dass die Joint Points, die Methoden 1 und 2 der Klasse A und die Methoden 1 und 2 der Klasse B darstellen.

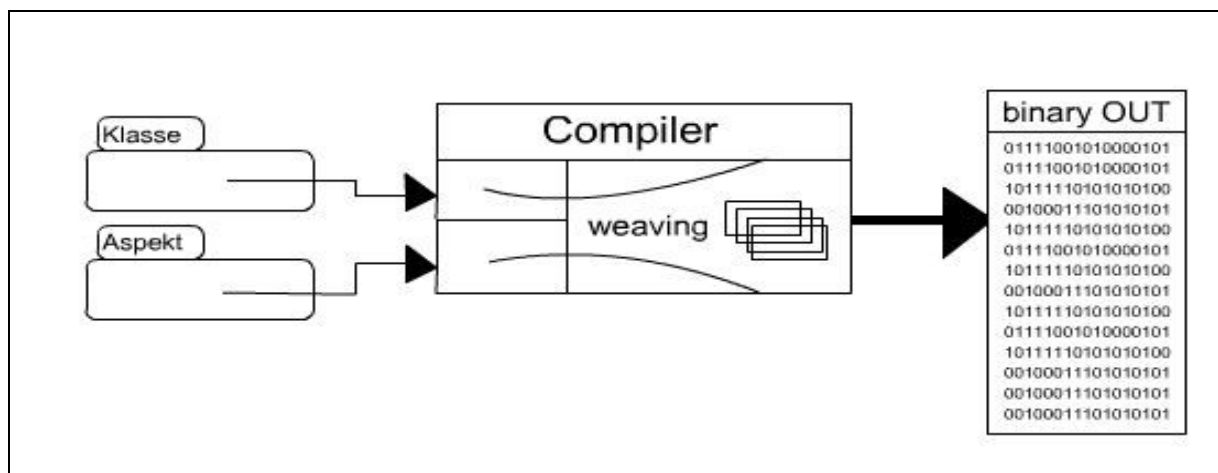
Das resultiert daher, dass der Logger jede Methode beider Klassen schneidet. Die Funktionalität für alle Methoden bei deren Betreten zu protokollieren, wird im Pointcut 1 dargestellt. Man kann sagen, dass die Funktionalität des Protokollierens für das Verlassen der Methoden im Pointcut 2 abgebildet wird. Es kann auch vorkommen das nur eine beziehungsweise ausgewählte Methoden einer Klasse benötigt werden.

Die direkte Implementierung für die Logiken der Pointcut's betreten und verlassen passiert an einer anderen stelle im Aspekt. Der Pointcut zeigt auf die abzuarbeitende Routine, den Advice. Wie dies dann im Detail aussieht wird im Kapitel 3.3 vorgestellt.

### 3. Voraussetzungen / Technologie

Die Aspektorientierte Programmierung ist ein Programmierparadigma, welches die Objektorientierte Programmierung um die Aspekte erweitert. Auf diese Art wird das Problem der Crosscutting Concerns gelöst. Um eine Applikation mit AOP zu erstellen ist ein spezieller Compiler nötig. Dieser Compiler muss für eine AOP Sprache geeignet sein um die formulierten Aspekte in die vorhandenen Objekte zu weave.

Für eine Vielzahl von Programmiersprachen gibt es mittlerweile eine aspektorientierte Variante. C++, C#, JAVA und PHP sind eine Auswahl dieser Sprachen.



**Bild 2:** Die Arbeitsweise eines AOP Compilers und der Vorgang des Weaving.

Ein sehr populäres Projekt ist das frei verfügbare AspectJ, das einerseits die Sprache bereitstellt, aber auch den nötigen Compiler liefert. Und für viele IDE's, wie z. B. Eclipse, Forte, JBuilder verfügbar ist. Dabei handelt es sich um Plugins, welche die jeweilige IDE erweitern und alle notwendigen Werkzeuge für die Aspektorientierte Entwicklung integrieren.

Ein anderes Projekt für die AOP in JAVA ist zum Beispiel Java Aspect Component [JAC].

#### 3.1 Einsatzmöglichkeiten

AOP ist für viele Bereiche gut geeignet und kann generell für alle Probleme die mit der OOP gelöst werden können, zum Einsatz kommen.

Die volle Stärke der aspektorientierten Lösung entfaltet sich bei Crosscutting Concerns wie z. B. bei dem zu Beginn vorgestellten Logger.

Weitere Einsatzgebiete sind:

- White Box Tests
- Security
- Error Handling
- Transaktionen
- Authentifizierung

Diese Aufzählung ist natürlich nicht abschließend und stellt nur eine Auswahl dar.

### 3.2 Vorgehensweisen

Die typische Vorgehensweise für die Entwicklung von Applikationen in AOP ist analog zu OOP, Identifizieren und Implementieren der Anforderungen und diese zu einem geschlossenen System wieder zusammen zufügen. [LAD 2003, Kap.: 1.5.2]

#### a) Dekoposition

Dieser Schritt zerlegt die Anforderungen eines Systems und bestimmt somit die Kernfunktion und deren schneidenden Interessen. Das Vorgehen trennt also Core-Level Concerns, Crosscutting Concerns und System-Level Concerns.

Im Beispiel des Loggers konnten wir zeigen, dass die Logik, die für das loggen zuständig ist von der Geschäftslogik getrennt werden kann. Dabei konnten die Fälle (Cases) betreten und verlassen heraus gearbeitet werden.

Eine Besonderheit in diesem Beispiel ist, dass der Vorgang für das Protokollieren meist eine systemweite Bedeutung besitzt und nicht nur auf eine Geschäftslogik angewendet wird.

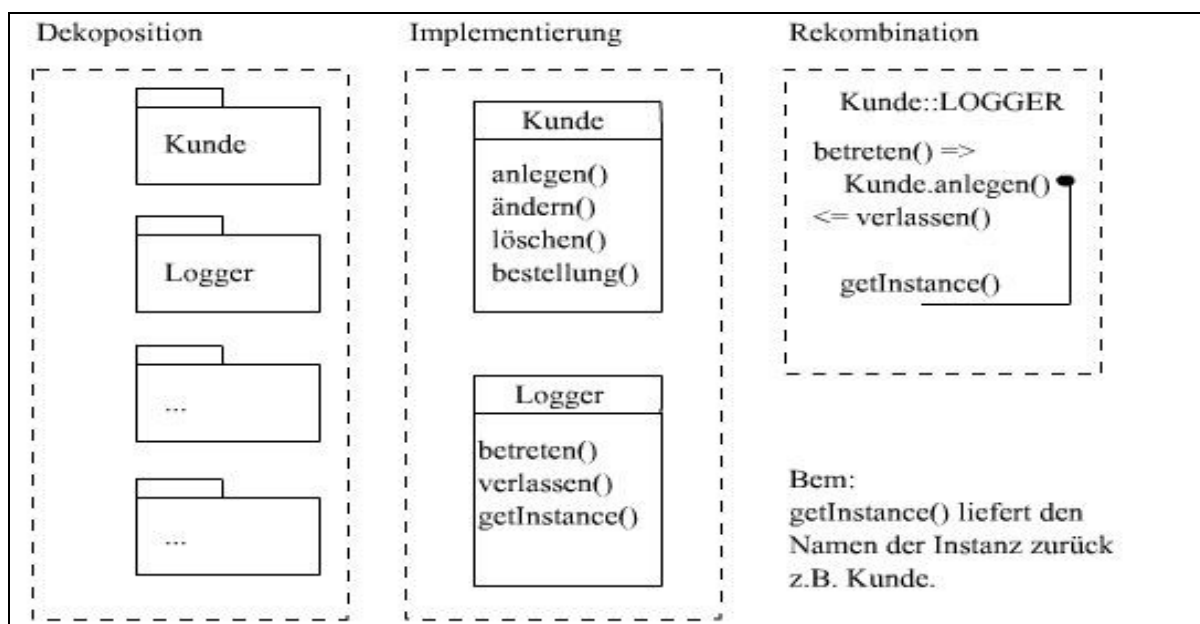
#### b) Implementierung

Im zweiten Schritt wird jede Anforderung für sich einzeln implementiert. Für die Kernanforderung eines Moduls können alle nutzbaren Techniken wie prozedurale oder auch objektorientierte Programmierung zum Einsatz kommen.

Betrachten wir wieder unser Beispiel. In dieser Phase wird das Logging als einzelnes Modul behandelt, das parallel zur Geschäftslogik arbeitet aber noch nicht darin integriert ist.

#### c) Rekombination

Nun werden die Rekompositionsregeln für die Zusammenführung in Aspekten spezifiziert. Dieser Prozess, auch als Weaving oder Integration bekannt, verwendet diese Informationen um das fertige System zu erstellen.



**Bild 3:** Vorgehensweise zur Modularisierung und Trennung der Logiken.

### 3.3 Syntax

Die von mir vorgestellte Syntax, ebenso alle Beispiele, beziehen sich auf die Sprache AspectJ. Meine Referenz für den folgenden Abschnitt ist das Buch „AspectJ in Action“ [LAD 2003, Kap. 3]. Ich behandle lediglich die notwendigen Grundlagen, die für den ersten Kontakt mit AspectJ notwendig sind. Für eine detailliertere Ausführung und weiterführenden Techniken sei auf die gerade erwähnte Referenz verwiesen.

#### 3.3.1 Aspekt

Die Definition eines Aspektes ist vergleichbar mit der einer Klasse. Das Schlüsselwort **aspect** leitet den Aspekt ein. Anschließend folgt der Name des Aspektes. Optional kann vor dem Aspekt noch die Sichtbarkeit angegeben werden.

```
Sichtbarkeit Schlüsselwort Name
public      aspect      example {
                /* some code */
}
```

#### 3.3.2 Pointcut

Der Pointcut wird im Aspekt notiert und mit dem Schlüsselwort **pointcut** eingeleitet. Wie bei dem Aspekt selbst kann auch für einen Pointcut die Sichtbarkeit festgelegt werden.

Die Struktur eines Pointcut ist wie folgt aufgebaut:

```
Sichtbarkeit Schlüsselwort Name() : Typ ( Signatur );
```

Wie hier zu erkennen ist, gibt es unterschiedliche Pointcut –Typen. Die wichtigsten davon sind in Tabelle 01 aufgeführt.

Der Vollständigkeit halber sei noch erwähnt, dass es noch weitere Pointcut –Typen gibt z.B. um den Kontrollfluss steuern.

Typ	Syntax
Methoden -Ausführung	execution(MethodSignature)
Methoden –Aufruf	call(MethodSignature)
Konstruktor -Ausführung	execution(ConstructorSignature)
Konstruktor –Aufruf	call(ConstructorSignature)
Klasseninitialisierung	staticinitialization(TypeSignature)
Feld Lesezugriff	get(FieldSignature)
Feld Schreibzugriff	set(FieldSignature)
Exception Handler -Ausführung	handler(TypeSignature)
Objektinitialisierung	initialization(ConstructorSignature)
Objekt pre Initialisierung	preinitialization(ConstructorSignature)
Advice –Ausführung	adviceexecution()

Tabelle 01

Jeder Pointcut –Typ besitzt eine Signatur. Diese Signatur selbst ist sehr komplex und ein wesentlicher Teil der AOP. Mit ihr wird festgelegt welche Methoden, Konstruktoren etc. in einem Pointcut zusammen gefasst werden.

Da es sehr aufwendig ist jede einzelne Methode in einem Pointcut zu notieren, können Wildcards verwendet werden.

In AspectJ sind drei Wildcards zulässig (\*, +, ..).

Eine Signatur leitet sich immer an den Klassen und deren Methoden, welche im Pointcut zusammengefasst werden her.

Ihre Verwendung lässt sich mit dem nachfolgendem Beispiel gut verstehen. Eine Ausführliche Beschreibung für mögliche Signaturen sind in [LAD 2003, Kap. 3.1.2] zu finden.

```
public aspect example {  
  
    public pointcut cut1() : call( public Account*+(..) );  
  
}
```

In dem Beispiel wird für den Pointcut cut1() jede als public deklarierte Methode die mit Account (\*) beginnt und deren Subklassen (+) selektiert. „(..)“ ist eine Wildcard, dass alle Parameter der Methoden mit einbezieht.

### 3.3.3 Advice

Der Advice gibt für einen Pointcut an was zu tun ist. Wie in der Definition schön erwähnt, enthält er den auszuführenden Programmcode.

Es gibt drei verschiedene Arten eines Advice:

- before, führt den Code vor betreten des Join-Point aus
- after, nach verlassen des Join-Point wird der Code des Pointcut abgearbeitet
- arround, umschließt den Join-Point und ermöglicht einen Bypass, um einen alternativen Code für den Join-Point auszuführen

```
public aspect example {  
  
    public pointcut cut1() : call(* Account.*(..)) ;  
  
    //Advice  
    before() : cut1() {  
        /* some code */  
    }  
  
}
```

Für den after –Advice gibt es zudem noch eine Besonderheit. Ein Join-Point kann auf drei verschiedene Arten verlassen werden. Die Rückkehr vom Join-Point kann normal, durch eine Exception oder auf undefinierte Weise erfolgen.

## 4. Ein Logging -Beispiel in AspectJ

An dieser Stelle folgt nun ein kleines Beispiel, das die Arbeitsweise von AspectJ vorstellt.

Die Listings sind unter [LAD 2003, Kapitel 2] im Original zu finden.

Getestet habe ich das Programm unter Eclipse 3.1 mit dem AspectJ Plugin 1.3.1 für Windows.

In den folgenden fünf Listings wird ein einfacher Logger realisiert. Dabei sind die Klassen Account, InsufficientBalanceException und SavingAccount (Listing 1-3) die eigentliche Anwendung. Im Aspekt (Listig 04) ist der Logger für die Anwendung eingebunden. Die Klasse Test (Listing 05) ist die ausführbare Datei.

Das Programm stellt ein einfaches Konto mit einem Kreditrahmen dar. Account (Listing 01) ist dabei das Konto, welches nur um ein bestimmtes Kontingent überzogen werden darf. Ist der Kreditrahmen erschöpft wird eine Ausnahme (Listing 02) geworfen. Die Kontonummer wird durch eine Vererbung von Account zu SavingsAccount umgesetzt. Damit erhält SavingsAccount die komplette Funktionalität von Account (Listing 03). Es wird nun auch klar weshalb in der Klasse Test die Klasse SavingAccount instanziiert wird, da die einzelnen Konten durch eine Kontonummer unterschieden werden müssen.

Listing: 01 :: Account.class

---

```
public abstract class Account {
    private float _balance;
    private int  _accountNumber;

    //Konstruktor
    public Account(int accountNumber) {
        _accountNumber = accountNumber;
    }

    //Kreditrahmen
    public void credit(float amount) { setBalance( getBalance() + amount); }

    //Prüft vor Auszahlung ob Kreditrahmen noch nicht erschöpft ist
    public void debit(float amount) throws InsufficientBalanceException {
        float balance = getBalance();
        if (balance < amount) {
            throw new InsufficientBalanceException("Total balance not sufficient");
        } else setBalance( balance - amount);
    }

    public float getBalance() { return _balance; }

    public void setBalance(float balance) { _balance = balance; }
}
```

Listing: 02 :: InsufficientBalanceException.class

---

```
public class InsufficientBalanceException extends Exception {

    public InsufficientBalanceException(String message) { super(message); }
}
```

Listing: 03 :: SavingAccount.class

---

```
public class SavingsAccount extends Account {

    public SavingsAccount(int accountNumber) { super(accountNumber); }
}
```

Bis zu dieser Stelle ist die Vorgehensweise nicht unbekannt gewesen. Die wirkliche Neuerung passiert nun im Listing 04, in dem der Aspekt definiert wird.

Der Aspekt loggt das Betreten und das Verlassen für alle Methoden. Dazu wird ein Pointcut mit dem Namen `tracePoints` definiert. Dieser Pointcut besitzt als Join-Point's alle Methoden der Klassen `Account`, `SavingsAccount`, `InsufficientBalanceException` und `test`. Der Pointcut –Typ `!within` sorgt dafür dass die Methoden des Aspektes `JoinPointTraceAspect` ignoriert werden.

Die Variable `_callDepth` zählt die Tiefe und ermöglicht so die Einrückung für die formatierte Ausgabe. Die Methode `print` wird von dem `before` sowie dem `after` Advice verwendet.

Listing: 04 :: `JoinPointTraceAspect.aj`

---

```
public aspect JoinPointTraceAspect {

    private int _callDepth = -1;

    pointcut tracePoints() : !within(JoinPointTraceAspect);

    before() : tracePoints() {
        _callDepth++;
        print("Before", thisJoinPoint);
    }

    after() : tracePoints() {
        print("After", thisJoinPoint);
        _callDepth--;
    }

    private void print( String prefix, Object message) {
        for(int i = 0, spaces = _callDepth * 2; i < spaces; i++) {
            System.out.print(" ");
        }
        System.out.println(prefix + ": " + message);
    }
}
```

Listing: 05 :: `test.class`

---

```
public class test {
    public static void main(String[] args) throws InsufficientBalanceException {
        SavingsAccount account = new SavingsAccount(12456);
        account.credit(100);
        account.debit(50);
    }
}
```

Wird nun das Programm kompiliert und ausgeführt erhält man folgende Ausgabe.

## OUTPUT:

---

```
Before: staticinitialization(test.<clinit>)
After: staticinitialization(test.<clinit>)
Before: execution(void test.main(String[]))
  Before: call(SavingsAccount(int))
    Before: staticinitialization(Account.<clinit>)
    After: staticinitialization(Account.<clinit>)
    Before: staticinitialization(SavingsAccount.<clinit>)
    After: staticinitialization(SavingsAccount.<clinit>)
    Before: preinitialization(SavingsAccount(int))
    After: preinitialization(SavingsAccount(int))
    Before: preinitialization(Account(int))
    After: preinitialization(Account(int))
    Before: initialization(Account(int))
      Before: execution(Account(int))
        Before: set(int Account._accountNumber)
        After: set(int Account._accountNumber)
        Before: set(int Account._accountNumber)
        After: set(int Account._accountNumber)
        After: execution(Account(int))
      After: initialization(Account(int))
    Before: initialization(SavingsAccount(int))
      Before: execution(SavingsAccount(int))
      After: execution(SavingsAccount(int))
    After: initialization(SavingsAccount(int))
  After: call(SavingsAccount(int))
Before: call(void SavingsAccount.credit(float))
  Before: execution(void Account.credit(float))
    Before: call(float Account.getBalance())
      Before: execution(float Account.getBalance())
      Before: get(float Account._balance)
      After: get(float Account._balance)
    After: execution(float Account.getBalance())
  After: call(float Account.getBalance())
Before: call(void Account.setBalance(float))
  Before: execution(void Account.setBalance(float))
    Before: set(float Account._balance)
    After: set(float Account._balance)
  After: execution(void Account.setBalance(float))
After: call(void Account.setBalance(float))
After: execution(void Account.credit(float))
After: call(void SavingsAccount.credit(float))
Before: call(void SavingsAccount.debit(float))
  Before: execution(void Account.debit(float))
    Before: call(float Account.getBalance())
      Before: execution(float Account.getBalance())
      Before: get(float Account._balance)
      After: get(float Account._balance)
    After: execution(float Account.getBalance())
  After: call(float Account.getBalance())
Before: call(void Account.setBalance(float))
  Before: execution(void Account.setBalance(float))
    Before: set(float Account._balance)
    After: set(float Account._balance)
  After: execution(void Account.setBalance(float))
After: call(void Account.setBalance(float))
After: execution(void Account.debit(float))
After: call(void SavingsAccount.debit(float))
After: execution(void test.main(String[]))
```

## 5. Resümee

Oft resultiert die an der AOP geübte Kritik aus dem unterschiedlichen Verständnis. Der Entwurf von Aspektorientierter Software erfordert ein Umdenken. Es werden dadurch neue Wege für das Design und die Implementierung eröffnet.

Da es sich um eine Erweiterung der OOP handelt, können bisher gesammelte Erfahrungen gut mit eingebracht werden. AOP wird die OOP nicht ersetzen. Es offeriert lediglich neue Lösungswege für bekannte Probleme.

Die Vorteile dieser Technik überwiegen klar. Als wichtige Eigenschaften sind folgende zu erwähnen:

- gute Modularisierung durch Trennung der Logiken
- einfache Systemerweiterung dank klar getrennter Module
- die Module sind übersichtlich, weil sie nichtmehr von fremden Logiken geschnitten werden
- durch die daraus resultierende späte Bindung des Designs sind Änderungen einfacher durchzuführen
- er erzeugte Code lässt sich noch besser wiederverwenden

Dank dieser Eigenschaften kann Software in kürzeren Entwicklungszyklen produziert werden. So werden automatisch die anfallenden Produktionskosten gesenkt. Wiederkehrende Aufgaben müssen nur noch einmal gelöst werden und enthalten somit wenige Fehlerquellen.

Es ist abzusehen dass durch die starken Vorteile die Popularität der AOP weiter anwachsen wird und in naher Zukunft dies der Standard in der Programmierung sein wird.

Natürlich darf man nicht vergessen daß die Entwicklung der AOP noch nicht abgeschlossen ist.

Der Programmablauf eines Systems wird nun größtenteils von Aspekten geregelt, was die Übersichtlichkeit des Kontrollflusses erschwert. Es kann leicht vorkommen, daß die durchlaufenden Sequenzen nun mehrdimensional werden.

Gut geeignete Darstellungsformen die das gesamte System übersichtlich kartografieren müssen noch gefunden werden. Denn nur mit einer geeigneten Dokumentation für diese Mehrdimensionalität lässt sich die Komplexität großer Systeme beherrschen.

Wir dürfen also gespannt sein was die AOP in der Zukunft noch bereit hält. Die Diskussion über die Notwendigkeit dieser Technologie kann als abgeschlossen betrachtet werden. Nun ist zu klären wie eine optimale Umsetzung und ein wirkungsvoller Umgang erreicht wird.

## 6. Literaturverzeichnis

[WAM 2003] Dean Wampler “The Role of AOP in OMG’s MDA”  
(2003 Aspect Programming Inc. Artikel)

[LAD 2003] RAMNIVAS LADDAD „AspectJ in Action“ (2003 Manning)

Lars Wunderlich: AOP: Aspektorientierte Programmierung in der Praxis.  
Entwickler.press, Frankfurt am Main 2005, ISBN 3-935042-74-4

Oliver Böhm: Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP.  
Dpunkt Verlag, Heidelberg 2006, ISBN 3-89864-330-1

Renaud Pawlak, Lionel Seinturier, Jean-Philippe Retaille: Foundations of AOP for J2EE  
Development (Foundation). Apress, Berkeley, Kalifornien , ISBN 1-590-59507-6

### 6.1 Internetressourcen

Quelle: Wikipedia

[http://de.wikipedia.org/wiki/Aspektorientierte\\_Programmierung](http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung) WIKIPEDIA

<http://common-lisp.net/project/closer/aspectl.html> AOP für Lisp

<http://www.eclipse.org/aspectj> AOP für Java

<http://aspectsharp.sourceforge.net> AOP für C#

<http://www.aspectc.org/> AOP für C++

<http://phpaspect.org/wiki/doku.php> AOP für PHP

<http://www.aosd.net> AOSD – Aspect-Oriented Software Development conference

<http://www.topprax.de> TOPPrax Forschungsprojekt. AOP für die Praxis.

TU Darmstadt

<http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp>

UNI Bonn

<http://www.cs.uni-bonn.de/~gk/>

Die Zahl der verfügbaren Ressourcen wächst kontinuierlich. Eine Vielzahl an Internetseiten diskutiert die Technologie der AOP. Mittlerweile sind auch aktuelle Buchtitel in deutscher Sprache erhältlich.