

Bitte Platzhalter löschen  
und durch eigenes Bild  
ersetzen



# Aspektorientierte Programmierung (AOP) mit AspectJ

Bericht 1 - Infoseminar 2015

Studiengang: Informatik  
Autoren: Emanuel Knecht, David Aeschlimann  
Betreuer: Prof. Dr. Jürgen Eckerle  
Datum: 26. Oktober 2015

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>2</b>
1.1. Auftrag . . . . .	2
1.2. Vorgehen . . . . .	2
<b>2. Aspektorientierte Programmierung</b>	<b>3</b>
2.1. Geschichte . . . . .	3
2.2. Motivation . . . . .	3
2.3. Aspektorientierte Sprache . . . . .	6
2.4. Konzepte . . . . .	7
2.5. Programmiersprachen . . . . .	8
<b>3. AspectJ</b>	<b>9</b>
3.1. Bestandteile von AspectJ . . . . .	9
3.2. Syntaxvarianten . . . . .	10
3.3. Weaving . . . . .	11
3.4. Entwicklungstools . . . . .	11
<b>4. Schlussfolgerungen</b>	<b>12</b>
4.1. Nutzen von AOP . . . . .	12
4.2. Nachteile von AOP . . . . .	12
4.3. Alternativen . . . . .	13
4.4. Fazit . . . . .	13
<b>Selbständigkeitserklärung</b>	<b>14</b>
<b>Literaturverzeichnis</b>	<b>15</b>
<b>Abbildungsverzeichnis</b>	<b>15</b>
<b>A. Demoprogramm</b>	<b>17</b>
A.1. Programmbeschreibung . . . . .	17
A.2. Source Code . . . . .	17

# 1. Einleitung

Dieses Dokument ist der schriftliche Teil des Modules Informatikseminar an der Berner Fachhochschule. In den kommenden Kapiteln wird die Aspektorientierte Programmierung mit AspectJ vorgestellt und erklärt.

## 1.1. Auftrag

Der Auftrag ist es die folgenden Fragen mit diesem Bericht zu beantworten.

*"Was versteht man unter dem Konzept der Aspektorientierten Programmierung?*

*Worin besteht der Vorteil gegenüber der OOP?*

*Erläutern Sie die wichtigsten Methoden und Ideen von AspectJ und stellen Sie heraus, in welcher Form OOP erweitert wird."* Dozenten (2015)

Unsere Erkenntnisse werden in diesem Dokument festgehalten. Anschliessend an die Abgabe dieses Berichtes erfolgt eine Präsentation im Plenum mit Fragerunde und Diskussion.

In der ersten Besprechung mit dem betreuenden Dozenten wurde uns nahegelegt auf eine zu technische und detailreiche Ausarbeitung des Themas zu verzichten und stattdessen den Fokus auf die unterliegenden Konzepte und Vorteile der Aspektorientierten Programmierung zu legen insbesondere in der Präsentation.

## 1.2. Vorgehen

In einem ersten Schritt musste das notwendige Wissen aufgebaut und gefestigt werden. Dazu wurden verschiedenste Informationsquellen konsultiert. Als eine wichtige Basis dieses Berichtes dient jedoch das Buch „AspectJ in Action“ Laddad (2009). Nach gemeinsamer Ausarbeitung der Struktur unseres Berichtes teilten wir die Kapitel auf und arbeiteten individuell weiter.

Durch Gegenlesen der vom Partner verfassten Abschnitten gelang es uns Fehler zu erkennen und einige Themen verständlicher zu formulieren. Die Präsentation basiert auf dem Bericht, der Fokus liegt jedoch auf dem Kapitel Aspektorientierte Programmierung.

## 2. Aspektorientierte Programmierung

### 2.1. Geschichte

Das Konzept der Aspektorientierten Programmierung wurde im Xerox PARC (Palo Alto Research Center Incorporated) entwickelt und gewann ab 1995 an Wichtigkeit. Wie bei allen neuen Spezifikationen war der Umfang und die Bestandteile der Aspektorientierten Programmierung zunächst nicht klar abgegrenzt. Gregor Kiczales und ein Team von Forschern waren massgeblich an der Entwicklung von AOP beteiligt.

Nach Entwicklung der theoretischen Grundlage folgte im Jahre 1998 eine erste Version von AspectJ, eine Implementation von AOP in Java. Die Version 1.0 von AspectJ wurde jedoch erst im Jahre 2002 nach weiterer Forschung veröffentlicht.<sup>1</sup>

Die Aspektorientierte Programmierung wurde durch die Publikation von AspectJ bekannt gemacht und es wurden seither Erweiterungen für die meisten populären Programmiersprachen entwickelt.

Die Entwicklung und der Betrieb von AspectJ wurde von XeroX Parc an Eclipse weitergereicht. Dort läuft die Entwicklung von AspectJ bis heute als Open-Source Projekt weiter. Mit der Version 1.8.7 wurde am 9. September 2015 die bis heute aktuellste Version veröffentlicht.

### 2.2. Motivation

Einer der Hauptgründe warum AOP entwickelt wurde ist die erweiterte Modularität die damit erreicht werden kann. Beim Design eines Systems werden in der Regel verschiedene Kategorien von Funktionalitäten aufgestellt und so die Software in verschiedene sogenannte Anliegen aufgeteilt. Diese Anliegen werden isoliert betrachtet und deren Funktionalität spezifiziert. Dabei unterscheidet man zwischen den folgenden zwei Gruppen von Anliegen (concern) einer Applikation.

- Core concerns (Kernanliegen)  
Hierbei handelt sich um die Kernfunktionalität der Applikation, die sogenannte Business Logic. Diese Gruppe beinhaltet zum Beispiel den Datenbankzugriff, die Interagierung mit dem Benutzer etc.
- Cross-cutting concerns (System-übergreifende Anliegen)  
Andere Funktionalitäten wie das Logging, die Sicherheit, Concurrency sowie Transaktionen betreffen das gesamte System und werden an vielen verschiedenen Orten verwendet.

Diese Gruppen dienen als Grundlage zur Veranschaulichung, warum gerade bei der Modularisierung die OOP Schwachstellen aufweist.

---

<sup>1</sup>Lopes (2002)

### 2.2.1. Objektoriente Programmierung

Mit der Objektorientierten Programmierung wurden viele Probleme und Unschönheiten von Prozeduralen Sprachen gelöst. Die OOP besitzt riesige Vorteile, welche die Softwareentwicklung vereinfachen. Einige der Kernbestandteile von OOP sind:

- Encapsulation: Daten und Methoden zur Veränderung derjenigen werden in Objekten gekapselt
- Inheritance: Das Verhalten oder die Daten können von einer Klasse geerbt werden.
- Polymorphism: Verschiede Objekte mit gleichem Supertyp reagieren unterschiedlich bei einem Methodenauf-ruf.

Die OOP erlaubt es Code modular zu strukturieren und Daten zu kapseln. Mit steigender Komplexität jedoch wird es schwierig den Code klar zu trennen und Abhängigkeiten so klein wie möglich zu halten.

Die Kernanliegen der Applikation werden in Klassen der Business Logic abgebildet. Diese Klassen werden jedoch sehr schnell durch den Code der System übergreifenden Anliegen "verschmutzt", so dass eine Klasse nicht nur für ein Anliegen verantwortlich ist. Dadurch wird das Single Responsibility Principle verletzt. Die folgende Graphik zeigt eine solche Beispielklasse, welche das beschriebene Phänomen veranschaulichen soll. Nur ein kleiner Teil der Methode (rot markiert) beschäftigt sich mit der Ausführung des Kernanliegens dieser Klasse, der Rest mit den System übergreifenden Anliegen.

```
public class SomeClass {  
  
    // Core data members  
    // Logging object reference  
    // Concurrency lock  
  
    public void someOperation(Object param) {  
        // check input parameter  
        // Ensure authorization  
        // Lock object (thread safety)  
        // Start transaction  
        // Perform operation  
        // Log successful performance of operation  
        // Commit transaction  
        // Unlock object  
    }  
}
```

Abbildung 2.1.: Beispielsklasse - Motivation für AOP

**Code Tangling** Von Code Tangling (to tangle - sich verwickeln) spricht man wenn ein Modul verschiedene Anliegen bearbeitet. Als Beispiel kann man die Abbildung 2.1 nehmen. Während der Designphase werden alle Anliegen separat entworfen und in der Implementation wird dennoch alles wieder miteinander verwickelt und verwoben. Die gewünschte Trennung der Anliegen und Modularisierung der Applikation wird nicht erreicht.

**Code Scattering** Bei Code Scattering (to scatter - streuen) ist die Perspektive eine andere. Die Funktionen eines Moduls werden in verschiedene anderen Modulen verwendet und so im ganzen System gestreut. Es gibt oftmals verschiedene Arten um auf Funktionen eines Modules zuzugreifen. Durch diese Aufrufe wird ein Teil der Logik des aufzurufenden Moduls ins aufrufende Modul verschoben. Folgende Abbildung soll dies anhand eines Sicherheitsmoduls veranschaulichen.

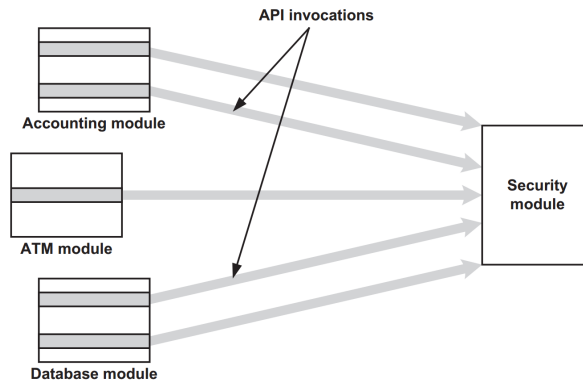


Abbildung 2.2.: Code Scattering ((Laddad, 2009, p 54))

Bei Verwendung von OOP ist das Code Tangling und Code Scattering unumgänglich. Selbst bei einem perfekt designten System werden diese Phänomene vorhanden sein. Dies beeinflusst Software Design und Entwicklung auf vielerlei Arten: Schlechte Nachvollziehbarkeit, weniger Produktivität, weniger Wiederverwendbarkeit von Code, viele repetitive Arbeiten, schlechtere Qualität und Wartbarkeit von Applikationen. Aus diesen Gründen macht es Sinn nach Alternativen zu suchen, ohne jedoch auf die Vorzüge von OOP zu verzichten.

### 2.2.2. Modularisierung mit AOP

AOP ist diese Alternative. Bleiben wir beim Beispiel eines Security Moduls; Das Modul wird mit Klassen implementiert und mittels Interfaces gegen aussen sichtbar gemacht. In der OOP werden nun alle Codeteile welche Securityfunktionen verwenden möchten einen Aufruf dieses Moduls beinhalten. Bei einer Änderung in der API müssen unter Umständen tausende Aufrufe ebenfalls geändert werden. Mit AOP jedoch beinhalten die Client Module keine Aufrufe mehr. Aufrufe des Sicherheitsmoduls werden bei genau definierten Punkten im Code automatisch ausgelöst. Es kann beispielsweise definiert werden, dass bei allen öffentlichen Methoden einer Klasse vor Ausführung des Bodys die Berechtigung des Benutzers geprüft wird. Bei dieser Deklaration der Einstiegspunkte im Code und des in diesen Fällen auszuführenden Codes spricht man von einem Aspect. Um diese automatische Ausführung möglich zu machen muss der Code der Kernanliegen (Business Logic) mit dem des Aspects verwebt werden. Man spricht hierbei von Weaving.

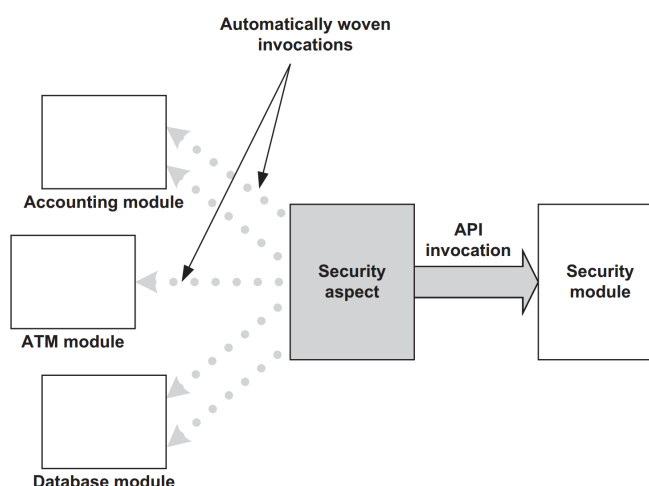


Abbildung 2.3.: Systemdesign mit AOP ((Laddad, 2009, p 55))

## 2.3. Aspektorientierte Sprache

Die Aspektorientierte Programmierung ist eine Methode in der Softwareentwicklung. Damit eine Programmiersprache um den Funktionsumfang von AOP erweitert werden kann, wird zuerst eine Spezifizierung erstellt und diese Spezifikation anschliessend umgesetzt.

### 2.3.1. Spezifizierung

**Programmiersprache** Als Grundlage für die Entwicklung eines Aspektorientierten Programms dient eine Programmiersprache. Es werden in der Regel bestehende Programmiersprachen wie C, C++, CSharp und Java verwendet, da diese von vielen Entwicklern bereits verstanden und verwendet werden. Im Falle der Entwicklung einer neuen Applikation werden alle Module in jener Programmiersprache isoliert voneinander entwickelt.

**Weaving Rules Spezifizierung** Die Sprache muss nun eine Möglichkeit bieten um diese entwickelten Module miteinander verknüpfen zu können. Dazu müssen sogenannte Weaving Rules deklariert werden. Die Weaving Rules bestimmen wie der Code verknüpft wird. Hierfür können Standardelemente einer Sprache verwendet (Annotations) oder auch die bestehende Sprache um neue Bestandteile erweitert werden (Keywords).

### 2.3.2. Implementation

Eine Implementation einer AOP Sprache kann in zwei Schritte gegliedert werden. Zuerst muss die Verknüpfung der verschiedenen Anliegen mittels Weaving rules sichergestellt und anschliessend daraus ausführbarer Code generiert werden. Bei AOP führt der sogenannte **Weaver** diese Aufgaben aus. Es gibt drei verschiedene Typen von Weavern:

- Source-to-Source weaver  
Der Sourcecode der Core und Crosscutting concerns wird zuerst verwoben und dieser neu entstandene Source Code von einem regulären Compiler kompiliert.
- Binary weaver  
Der Sourcecode der Core und Crosscutting concerns wird zuerst kompiliert und der daraus entstandene Byte Code wird vom Weaver verknüpft.
- Load time weaver  
Vergleichbar mit dem Byte Code weaving, ausser dass der Verknüpfungsvorgang erst beim Aufrufen des Programms statt findet.

Ein Weaver ist nicht mit einem Compiler gleichzustellen. Je nach Typus braucht es jedoch eine enge Zusammenarbeit zwischen Weaver und Compiler. Deswegen stellen einige Anbieter von AOP Erweiterungen den Entwicklern ihre eigenen Compiler inklusive Weaver zur Verfügung.

## 2.4. Konzepte

Die folgenden Konzepte sind die Grundlage von Aspektorientierter Programmierung. Dies ist ein generisches Modell und nicht jede Implementation einer Aspektorientierten Programmiersprache muss zwingend alle Konzepte implementieren.<sup>2</sup>

- *Identifizierbare Punkte in der Ausführung des Programms*  
Während der Ausführung eines Programms gibt es Points of Interest. Solche Punkte sind beispielsweise der Aufruf einer Methode oder das Werfen von Exceptions. Im Umfeld von AOP werden diese Punkte **Join Points** genannt.
- *Selektion von Punkten während des Programmablaufs*  
Diese Join Points müssen irgendwie angesteuert werden können. Dies geschieht mit einem **Pointcut**. Ein Pointcut beinhaltet ein Statement, welches eine gewisse Anzahl von Join Points selektiert. So könnten beispielsweise alle öffentlichen Methoden aller Klassen eines Systems selektiert werden. Ein Pointcut kann auch auf den Kontext eines Join Points zugreifen (Parameter einer Methode, Klasse etc.).
- *Veränderung des regulären Programmablaufs (dynamic crosscutting)*  
Wenn ein Join Point von einem Pointcut selektiert wurde, soll der reguläre Programmablauf durch Ausführung von zusätzlichem Code verändert werden. Dieser Code ist vom Entwickler frei wählbar und der Programmablauf wird dadurch dynamisch. Der auszuführende Code wird in einem **Advice** gesammelt.
- *Veränderung der statischen Struktur des Systems (static crosscutting)*  
Um gewisse crosscutting concerns umsetzen zu können müssen in einer Klasse zusätzliche Felder deklariert werden (**inter-type declaration**). Ausserdem kann es nötig sein bereits beim weaving zu wissen, ob gewisse Join Points im System vorhanden sein werden, um angemessen darauf reagieren zu können. Dies wird durch **weave-time declarations** ermöglicht.
- *Deklaration aller Konstrukte*  
All diese Elemente (pointcuts, dynamic & static crosscutting) werden logisch an einem Ort, dem **Aspect**, deklariert. Der Aspekt kapselt ein systemübergreifendes Anliegen.

Alle diese Konzepte werden in der Abbildung 2.4 zusammengefasst und deren Beziehung zueinander graphisch abgebildet.

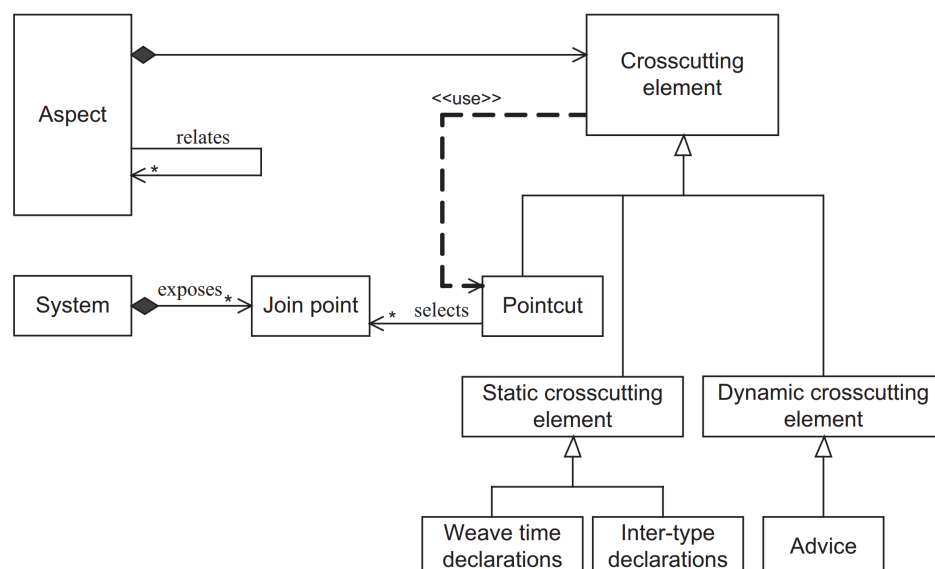


Abbildung 2.4.: AOP Konzepte ((Laddad, 2009, p 60))

<sup>2</sup>(Laddad, 2009, p 58)



## 2.5. Programmiersprachen

Beinahe für jede bekanntere Programmiersprache gibt es eine Erweiterung oder ein Framework um AOP als Gesamtes oder Teile davon zu ermöglichen. Nachfolgend eine Übersicht über die bekanntesten Sprachen mit den verbreitesten Frameworks.

### 2.5.1. Java

AspectJ gilt als erstes komplettes AOP Framework überhaupt und ist deshalb auch am Verbreitesten.

- AspectJ - Wird detailliert im Kapitel AspectJ beleuchtet
- Spring AOP - Spring ist ein Framework zur Entwicklung von Java Applications. Auch dort gibt es die Möglichkeit AOP zu verwenden. Jedoch ist man limitiert auf Methodenaufrufe als Join Points.

### 2.5.2. C, C++

- AspectC++ - Eine Adaptierung von AspectJ in C/C++ <sup>3</sup>

### 2.5.3. .NET Framework

CSharp unterstützt nativ sogenannte Extension methods. Hiermit können nachträglich zu Klassen oder Interfaces zusätzliche Methoden hinzugefügt werden (static crosscutting). Dies ist allerdings nur ein Teil der AOP, für alles andere benötigt man ein Framework.

Nemerle -> Emaaaaa

- PostSharp - Kommerzielle Software welche eine grosse Verbreitung genießt (Siemens, Roche, Lufthansa und viele mehr) <sup>4</sup>
- AspectSharp - AOP Framework für .NET<sup>5</sup>
- Afterthought - Afterthought ist eine Open Source Alternative zu Postsharp.<sup>6</sup>

---

<sup>3</sup>?

<sup>4</sup>Unknown (2015d)

<sup>5</sup>Unknown (2015c)

<sup>6</sup>Unknown (2015b)

## 3. AspectJ

### 3.1. Bestandteile von AspectJ

In diesem Kapitel betrachten wir, wie die verschiedenen Komponenten des im Abschnitt Konzepte vorgestellten Modells einer Aspektorientierten Programmiersprache in AspectJ umgesetzt sind. Alle hier verwendeten Codeteile sind Bestandteil unseres Demoprogrammes, welches sich im Anhang befindet (Demoprogramm).

#### 3.1.1. Allgemein

##### Aspect

Der Aspect ist die zentrale Einheit in AspectJ. Im Aspect werden alle Bestandteile eines Anliegens einer Applikation zusammengefasst. Ein Aspect kann gleich wie eine normale Javaklasse Attribute und Methoden enthalten und dient zur Kapselung des Anliegens. So werden alle Funktionen welche das Logging einer Applikation betreffen im Logging aspect zusammengefasst. Aspects werden in Dateien mit der Endung .aj gespeichert.

```
public aspect LogAspectShort {
    private int logCount;
    private void increaseLogCount() {
        logCount++;
    }
    //pointcut, advices
}
```

##### Join-Point Modell

Die Join Points sind die Punkte im Programmablauf wo die Systemübergreifenden Anliegen im Code der Business Logic anknüpfen sollen. AspectJ bietet eine Menge solcher Punkte an und der Entwickler muss sich genau überlegen welchen Punkt er als Einstiegspunkt verwenden will. Die nachfolgende Tabelle zeigt eine Übersicht der vorhandenen Join Points in AspectJ.

Kategorie	Join Point
Methode	Ausführung der Methode (Method Body)
Methode	Aufruf der Methode (aufrufender Kontext)
Konstruktor	Ausführung des Konstruktors
Konstruktor	Aufruf des Konstruktors
Feldzugriff	Lesender Zugriff auf Feld
Feldzugriff	Schreibender Zugriff auf Feld
Exception Handling	Catch Block
Initialisierung	Laden, Initialisierung und Pre-Init einer Klasse
Advice	Ausführung eines Advice

Tabelle 3.1.: Übersicht über alle Join Points in AspectJ

Die Pointcuts wählen einen oder mehrere solcher Punkte aus und können intern benannt werden. Die Syntax der Pointcuts beruht auf Signaturen von Methoden oder Feldern. Die Selektion der Join Points kann aufgrund des Access Modifiers, der Rückgabetypen, der Klasse und des Namens des Members eingeschränkt werden. Der Stern kann als Wildcard Charakter verwendet werden. So selektiert der nachfolgende pointcut alle Methoden, aller Klassen die öffentlich sind.

```
pointcut publicMethods() : execution(public * *.*(..));
```

### 3.1.2. Dynamic crosscutting

Beim Dynamic crosscutting wird der Programmfluss verändert und um zusätzlichen Code erweitert. Dieser zusätzliche Code wird in AspectJ in einem Advice gekapselt. Der Advice kann vor, nach oder um den Join Point herum ausgeführt werden (before, after, around). Folgender Advice schreibt den Start und das Ende der Methode auf die Konsole. Über das Objekt thisJoinPoint kann auf den Kontext des Join Points zugegriffen werden. In diesem konkreten Fall verwenden wir den Kontext um auf die Signatur der Methode zuzugreifen.

```
Object around() : publicMethods() {
    System.out.println("Executing:_" + thisJoinPoint.getSignature());
    Object ret = proceed();
    System.out.println("Finished:_" + thisJoinPoint.getSignature());
    return ret;
}
```

### 3.1.3. Static crosscutting

**Inner-type declaration**

**Weave-time declaration**

## 3.2. Syntaxvarianten

Für die Verwendung von AspectJ können zwei Syntaxvarianten eingesetzt werden.

- Traditionelle Variante  
Diese Variante haben wir bisher in allen unseren Beispielen verwendet. Man verwendet die Schlüsselwörter von AspectJ und der gesamte Umfang von AspectJ steht zur Verfügung. Man nennt diese Variante traditionell, da dies zu Beginn die erste und einzige Möglichkeit war um AspectJ zu verwenden.
- Annotation-based Variante  
In der annotation-based Variante verwendet man normale Javaobjekte um die Konstrukte von AspectJ abzubilden. Diese Klassen und Methoden werden mit Annotations versehen, damit der Weaver versteht wie der Code zu verknüpfen ist. Diese Variante wurde erst ab Version 5 von AspectJ eingeführt und es sind nicht ganz alle Konstrukte abbildbar.

@Aspect

```
public class LogAspectAlt {
    @Pointcut("execution(public *_*.*(..))")
    public void publicMethods() {}
    @Around("publicMethods()")
    public Object logItAll(ProceedingJoinPoint point) throws Throwable {
        System.out.println("Executing_(Alt):_" + point.getSignature());
        Object ret = point.proceed();
        System.out.println("Finished_(Alt):_" + point.getSignature());
        return ret;
    }
}
```

```
}  
}
```

### **3.3. Weaving**

#### **3.3.1. Source weaving**

#### **3.3.2. Binary weaving**

#### **3.3.3. Load-time weaving**

### **3.4. Entwicklungstools**

#### **3.4.1. Eclipse**

AspectJ wird als Open Source Projekt von Eclipse weiterentwickelt. Deshalb werden für die IDE Eclipse AspectJ Development Tools (AJDT) zur Verfügung gestellt. Emaaaa Unknown (2015a)

#### **3.4.2. IntelliJ**

#### **3.4.3. Weitere**

## 4. Schlussfolgerungen

### 4.1. Nutzen von AOP

**Einfacheres Programmdesign** Die verschiedenen Module einer Applikation können einzeln entworfen werden, ohne den Fokus auf das Zusammenspiel derjenigen richten zu müssen. Design Entscheidungen können auch erst zu einem späteren Zeitpunkt im Projektverlauf getroffen werden ohne Änderungen im gesamten System zu verursachen. Features werden erst entworfen wenn sie wirklich gebraucht werden und nicht „auf Vorrat“.

**Saubere Implementation** Mit AOP wird das Single Responsibility Principle voll und ganz eingehalten. Eine Klasse oder ein Modul behandelt nur genau seine Aufgabe. Desweiteren werden in der Implementation Code Tangling und Code Scattering vermieden. Die Logik eines Modules ist genau an einem Ort zu finden, nämlich im Modul selber.

Dies macht es möglich bereits früh im Projekt einen voll funktionstüchtigen Protoypen der Kernfunktionalität (Business Logic) zur Hand zu haben. So können Fehler in den Anforderungen oder der Business Logic frühzeitig erkannt und die Crosscutting Concerns wie Logging oder Security erst nachträglich hinzugefügt werden ohne im ganzen System Änderungen machen zu müssen.

Dies macht die Entwicklung mit AOP schnell und effizient, daher auch günstiger für Unternehmen. Die einzelnen Module können ausserdem besser getestet und durch die Verantwortlichen geprüft werden (review).

**Bessere Wiederverwendbarkeit** Der Schlüssel zur Wiederverwendung von Code ist so wenig Coupling wie nur möglich. Bei AspectJ ist es möglich auf Codeebene gar kein Coupling zu haben. Ein Aspect kann beliebig oft wiederverwendet werden. Im Gegensatz zu einem klassischen Modul (API) muss die Business Logik der Applikation nicht durch Aufrufe verschmutzt werden. Wenn genau die gleiche Funktionalität gesucht wird, kann auch der Aspect ohne Veränderung übernommen werden. Dies vereinfacht insbesondere bei Unternehmen mit Applikationen ähnlicher Architektur der Betrieb und die Entwicklung dieser Applikationen.

**Einfachere Erweiterbarkeit und Anpassung** Neue systemübergreifende Anliegen können problemlos und ohne Anpassung des Systems hinzugefügt werden. Bei bestehenden Modulen reicht es die einzelnen Module anzupassen und es ist nicht überall Änderungen vorzunehmen.<sup>1</sup>

### 4.2. Nachteile von AOP

**Zusätzliches Wissen** Die Verwendung von AOP setzt ein fundiertes Wissen voraus. Die Konzepte müssen verstanden und umgesetzt werden. Entweder werden bestehende Entwickler ausgebildet oder es werden erfahrene Entwickler angestellt. Beides kostet Geld und ist eine Investition in die Zukunft. Um die Architektur einer komplexen Applikation mit AOP zu designen und umzusetzen braucht es definitiv einen Experten.

**Veränderung des Build- und Entwicklungsprozesses** Je nach Unternehmen gibt es genaue Richtlinien und Prozesse wie Software entwickelt werden muss. Aufgrund der grossen Flexibilität von AOP können sich dort Konflikte ergeben. Ausserdem muss ein anderer Compiler und ganz sicher ein Weaver für das Builden eingesetzt werden. Das kann zu Veränderung in der Konfiguration der Infrastruktur und des Deployments führen.

---

<sup>1</sup>Pratap (2012)

**Komplexer Programmfluss** Durch blosses Betrachten des Codes ist der genaue Programmfluss für Entwickler ohne AOP Kenntnisse nur schwer zu erkennen. Hierbei können die Entwicklertools (bspw. Eclipse) Klarheit schaffen indem sie den Ablauf visualisieren und die von einem Pointcut angesprochenen Join Points markieren. Doch eigentlich ist genau diese Unabhängigkeit des Codes bei Verwendung von AOP gewünscht. Abstraktion versteckt Details und AOP ist nach OOP eine zusätzliche Ebene an Abstraktion.

### 4.3. Alternativen

Es gibt keine echte Alternative die genau den gleichen Funktionsumfang wie AOP hat.<sup>2</sup> Es gibt jedoch Methoden oder Patterns, welche die Modularität und die Trennung von Crosscutting und Core Concerns auch ohne die Verwendung von AOP begünstigen. Im Detail auf diese Alternativen einzugehen sprengt den Rahmen dieses Berichtes.

- Design Patterns
  - Observer, Chain of Responsibility, Proxy and Decorator Pattern<sup>3</sup>
  - Component-Based Programming<sup>4</sup>
- Dependency injection
- Functional Programming
- Code Generation<sup>5</sup>

### 4.4. Fazit

Aspect-Oriented Programming ist eine ansprechende und hilfreiche Erweiterung zu OOP. AOP löst einige uns allen wohlbekannte Probleme der klassischen Softwareentwicklung mit OOP. Uns erstaunt dass sich die Verbreitung von AOP in Grenzen hält und sie sich nicht als neuer Standard in der Softwareentwicklung etabliert hat. Gründe hierfür könnte die zusätzliche Komplexität und die nötige Weiterbildung sein, die Unternehmen scheuen. Vielleicht liegt es auch einfach an der mangelhaften Bekanntheit der Technologie oder daran dass bestehende Applikationen oder Bibliotheken nur mit viel Aufwand auf AOP umzuschreiben sind.

Das Kennenlernen der Konzepte und Ideen von AOP war für uns beide eine Bereicherung. Wir glauben das AOP die Softwareentwicklung vereinfachen und revolutionieren könnte. Die Lernkurve ist aber relativ steil. Wir freuen uns darauf AOP in einem realen Projekt anwenden zu können und so einem Praxistest zu unterziehen.

---

<sup>2</sup>Bruce (2001)

<sup>3</sup>(Laddad, 2009, p 66)

<sup>4</sup>Various (2015)

<sup>5</sup>Various (2010)

# Selbständigkeitserklärung

Wir bestätigen, dass wir die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe/n. Sämtliche Textstellen, die nicht von uns stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum:                      Biel, 26. Oktober 2015

Namen Vornamen:            Emanuel Knecht                      Aeschlimann David

Unterschriften:              .....                      .....

# Literaturverzeichnis

- D. Bruce. Alternatives to aspect-oriented programming, 2001. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.6399&rep=rep1&type=pdf>.
- Dozenten. Auftragsbeschreibungen informatikseminar, 2015. URL <https://moodle.bfh.ch/mod/data/view.php?d=462>.
- R. Laddad. *AspectJ in Action 2nd Edition*, volume 2. Manning Publications, 2009. ISBN 978-1933988054.
- C. V. Lopes. Aspect-oriented programming:an historical perspective, 2002. URL [http://isr.uci.edu/tech\\_reports/UCI-ISR-02-5.pdf](http://isr.uci.edu/tech_reports/UCI-ISR-02-5.pdf).
- B. Pratap. Benefits of aop, 2012. URL <http://www.tekhnologia.com/2012/01/benefits-of-aspect-oriented-programming.html>.
- Unknown. Aspectj development tools (ajdt), 2015a. URL <http://www.eclipse.org/ajdt/>.
- Unknown. Afterthought open source framework, 2015b. URL <https://github.com/vc3/Afterthought>.
- Unknown. Aspectsharp framework, 2015c. URL <http://sourceforge.net/projects/aspectsharp/>.
- Unknown. Postsharp, 2015d. URL <https://www.postsharp.net/customers>.
- Various. Alternatives to aspect-oriented programming, 2010. URL <http://programmers.stackexchange.com/questions/1060/what-alternatives-are-there-for-cross-cutting-concerns-other-than-aspect-orient>.
- Various. Component based programming, 2015. URL [https://en.wikipedia.org/wiki/Component-based\\_software\\_engineering](https://en.wikipedia.org/wiki/Component-based_software_engineering).



# Abbildungsverzeichnis

2.1. Beispielsklasse - Motivation für AOP . . . . .	4
2.2. Code Scattering ((Laddad, 2009, p 54)) . . . . .	5
2.3. Systemdesign mit AOP ((Laddad, 2009, p 55)) . . . . .	5
2.4. AOP Konzepte ((Laddad, 2009, p 60)) . . . . .	7

# **A. Demoprogramm**

Dieses Demoprogramm wurde mit Eclipse Mars und den AspectJ Developer Tools für Eclipse entwickelt und getestet. Es soll die grundlegenden Möglichkeiten und die Funktionsweise von AspectJ veranschaulichen.

## **A.1. Programmbeschreibung**

## **A.2. Source Code**