Aspektorientierte Programmierung (AOP) mit AspectJ

Bericht 1 - Infoseminar 2015

Studiengang:	Informatik
Autoren:	Emanuel Knecht, David Aeschlimann
Betreuer:	Prof. Dr. Jürgen Eckerle
Datum:	1. November 2015

Inhaltsverzeichnis

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 des 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.[1]
Das Konzept der Aspektorientierte Programmierung wurde durch die Publikation von AspectJ bekannt und es wurden seither Erweiterungen für die meisten populären Programmiersprachen entwickelt und veröffentlicht.

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

2.2. Motivation

Einer der Hauptgründe warum AOP entwickelt wurde ist die erweiterte Modularität, welche 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 (concerns) aufgeteilt. Diese Anliegen werden isoliert betrachtet und der Funktionsumfang einzeln spezifiziert. Dabei unterscheidet man zwischen den folgenden zwei Gruppen von Anliegen einer Applikation:

- Core concerns (Kernanliegen)
  Hierbei handelt sich um die Kernfunktionalität der Applikation, die sogenannte Business Logic. Diese Gruppe beinhaltet zum Beispiel den Datenbankzugriff sowie die Interaktion mit dem Benutzer
- 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 in der Business Logic verwendet.

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

[1] Lopes (2002)

2.2.1. Objektoriente Programmierung

Mit der Objektorientierten Programmierung wurden viele Probleme und Unschönheiten von Prozeduralen Sprachen gelöst. Die OOP vereinfacht und abstrahiert die Softwareentwicklung. Einige der wichtigsten Konzepte der 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: Verschiedene Objekte mit gleichem Supertyp reagieren unterschiedlich bei einem Methodenaufruf. Der genaue Typ des aufgerufenen Objektes soll der aufrufenden Instanz nicht bekannt sein

Die OOP erlaubt es Code modular zu strukturieren und Daten zu kapseln. Mit steigender Komplexität wird es jedoch schwierig den Code klar zu trennen und die Abhängigkeiten zwischen Modulen 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". Dadurch wird das Single Responsibility Principle verletzt; eine Klasse ist nicht nur für ein bestimmtes Anliegen verantwortlich. 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 seperat entworfen und in der Implementation wird dennoch alles wieder miteinander verwickelt und verwoben. Die gewünschte Trennung der Anliegen und Modularisierung der Applikation wird mit OOP nicht vollständig 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.

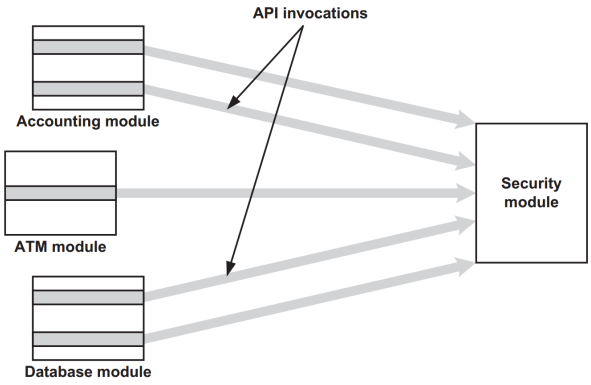

Abbildung 2.2.: Code Scattering ((Laddad, 2009, p 54))

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 OOP zu suchen, ohne jedoch auf deren Vorzüge 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.

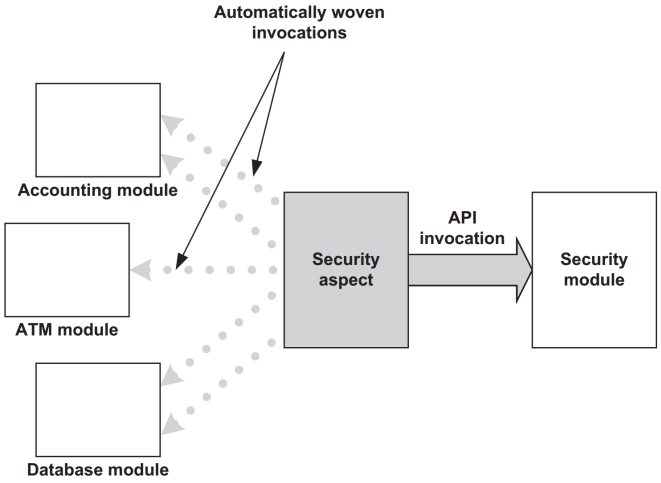

Abbildung 2.3.: Systemdesign mit AOP ((Laddad, 2009, p 55))

2.3. Aspektorientierte Sprache

Die Aspektorientierte Programmierung ist eine Methode der Softwareentwicklung. Damit eine Programmiersprache um den Funktionsumfang von AOP erweitert werden kann, muss zuerst ein genaues Konzept der verschiedenen Komponenten einer Aspektorientierten Programmiersprache spezifiziert und diese Spezifikation anschliessend umgesetzt werden.

2.3.1. Spezifikation

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 angewendet 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, Xml config) 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 Aspekorientierter Programmierung. Dies ist ein generisches Modell und nicht jede Implementation einer Aspekorientierten Programmiersprache muss zwingend alle Konzepte implementieren. AspectJ jedoch implementiert alle hier vorgestellten Komponenten.[2]

- *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 Moduls selektiert werden. Ein Pointcut kann auch auf den Kontext eines Join Points zugreifen (Parameter einer Methode, Typ, 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 Komponenten (pointcuts, dynamic & static crosscutting) werden logisch an einem Ort, dem **Aspect**, deklariert. Der Aspekt kapselt ein systemübergreifendes Anliegen, respektive enthält die Verknüpfung zwischen der Business Logic und einem systemübergreifenden Modul.

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

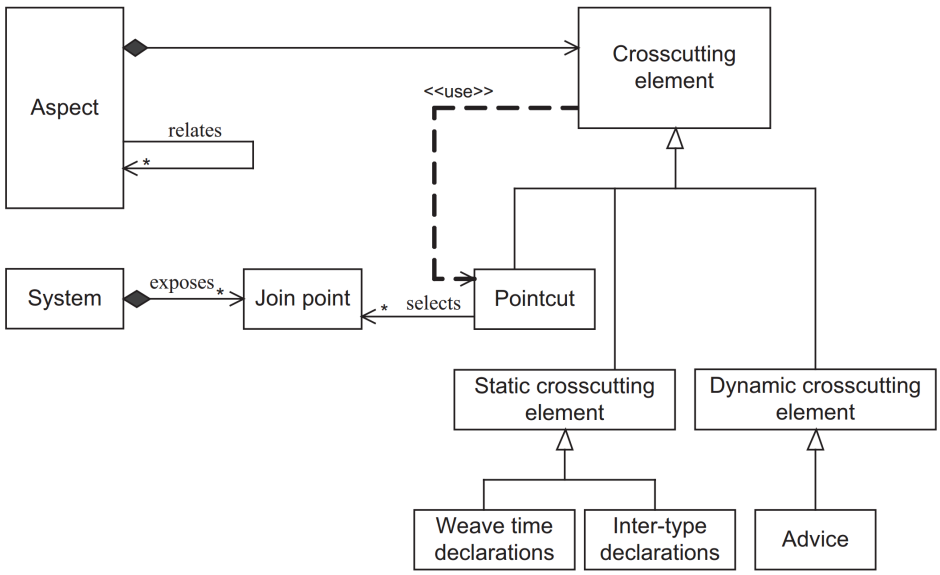

Abbildung 2.4.: AOP Konzepte ((Laddad, 2009, p 60))

[2](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++ [3]

2.5.3. .NET Framework

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

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

2.5.4. Nemerle

Nemerle ist eine relativ junge, quelloffene Programmiersprache (2003 erstmals öffentlich erschienen, erster stabiler Release 13. Mai 2011). Sie benutzt die Common Language Infrastucture (CLI) und kann somit Plattformunabhängig (.net/Mono) genutzt werden. Nemerle unterstützt Funktionale-, Objektorientierte- und Imperative Programmierung und verfügt über ein mächtiges Meta-Programming System (Makros) welches dadurch auch Aspektorientierte Programmierung ermöglicht. Beispiele unter: (RSDN)

[3] AspectC (2015)
[4] PostSharp (2015)
[5] AspectSharp (2015)
[6] Afterthought (2015)

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 können. AspectJ bietet eine Menge solcher Punkte an und der Entwickler muss sich genau überlegen welchen Punkt er als Einstiegspunkt verwenden will, da der Kontext des Join Points sich teils sehr stark unterscheidet (Bsp. Aufruf oder Ausführung von Methode). 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

Mit Static crosscutting wird die statische Struktur des Programms verändert. Es können Members wie Variablen und Methoden zu Klassen hinzugefügt, Annotations an Typen, Feldern, Methoden und Konstruktoren angebracht, Warnings und Errors generiert und Exceptions abgefangen werden.

Inner-type declaration Mit Inter-type declarations können Variablen und Methoden an bestehende Klassen angefügt werden, auch wenn man keinen Zugrif auf deren Code hat (in APIs).
Beispiel:

```
public interface Visitor{
        void visitPoint(Point p);
}


public aspect PointVisitorAspect {
        private java.util.ArrayList<Visitor> Point.visitors =
            new java.util.ArrayList<Visitor>();

        public <T extends Visitor> void Point.visit(T v){
                this.visitors.add(v);
                v.visitPoint(this);
        }
}

//====== Call ======
public class SomeCallerClass{
        public void someMethod(Visitor v){
                Point p = new Point(0,0);
                p.visit(v);
        }
}
```

Hier wird der Klasse java.awt.Point eine Methode visit(Visitor) hinzugefügt, welche sich alle Besucher (visitors) dieses Punktes in einer ArrayList merkt. Mit dem Code aus dem Beispiel könnte diese Liste zwar nicht ausgelesen werden, was aber für diese Veranschaulichung keine Rolle spielt.
Inner-type declarations können auch verwendet werden um Standardimplementationen für Interfaces bereitzustellen.

Type-hierarchy modification Mit Aspekten kann auch die Klassenhierarchie verändert werden. Sowohl das implementieren von Interfaces als auch das Erweitern und Erben von Klassen ist möglich. Multi-inheritance wird jedoch nach wie vor nicht unterstützt.
Hier einige Beispiele:

```
//====== 1 ======

public class VisitorImpl {

}

public aspect VisitorImplementationAspect{
        declare parents: VisitorImpl implements Visitor;

        public void VisitorImpl.visitPoint(Point p){
                System.out.println("visited:"+p.toString());
        }
}

//====== 1 ======

public @interface PointVisitor {

}

@PointVisitor
public class PointVisitorImpl {

}

public aspect PointVisitorImplementationAspect {
        declare parents: @PointVisitor * extends VisitorImpl;
}

//====== Call ======
public class AnotherCallerClass{
        public void anotherMethod(){
                java.awt.Point p = new jawa.awt.Point(0,0);
                Visitor v1 = new VisitorImpl();
                Visitor v2 = new PointVisitorImpl();
                p.visit(v1);
                p.visit(v2);
        }
}
```

Im ersten Teil wird die Klasse VisitorImpl als Implementation von Visitor deklariert. Bekanntlich müssen bei Interfaces alle Methoden implementiert werden. Auch dies kann wie wir zuvor gesehen haben vom Aspekt übernommen werden. Anstelle des Klassennamen kann auch ein Pattern angegeben werden, sodass mehrere Klassen aufs Mal verändert werden können. Dies ist beim zweiten Beispiel der Fall, wo alle Klassen mit der @PointVisitor Annotation zu Subklassen von VisitorImpl werden.

Weave-time declaration Weave-time declarations geben dem Entwickler die möglichkeit Errors und Warnings zu generieren. Eine typische Anwendung dafür ist es zu verhindern, dass nicht unterstützte Methoden verwendet werden.

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.
- Xml Variante
  Wird vom Load-time Weaving-Agent verwendet um den Bytecode geladener Klassen mit jenem der Aspekte zu verweben. Da diese Variante eher selten eingesetzt wird, verzichten wir hier auf Beispielcode.

```
@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

Weavingprozesse in AspectJ können entweder nach Input oder nach Zeitpunkt des Weavings unterschieden werden.

3.3.1. Build-time weaving

Der AspectJ compiler ajc produziert aus .java, .aj, .class und .jar Dateien Bytecode, der auch auf Standard Java Virtual Machines ausgeführt werden kann. Es muss jedoch äspectjrt.jar"der Umgebungsvariable Classpath hinzugefügt werden, wenn eine Standard JVM genutzt werden soll. Diese Klassenbibliothek beinhaltet AspectJ Typendefinitionen sowie Definitionen für JointPoints oder Signaturen, Annotationen und interne Klassen von AspectJ.

Build-time Source weaving

Die am Oftesten verwendete Form des Weavings akzeptiert sowohl reguläre Klassen als auch Aspekte als Source code. Aspekte können in klassischer Form in .aj Dateien oder in Annotationsschreibweise (@Aspect) vorliegen.
Beispiel build Befehl:
> ajc path/to/src/*.java path/to/aspects.aj
Es gilt zu beachten, dass ,im Gegensatz zum Standard Java Compiler javac, mit ajc alle Source Dateien zusammen kompiliert werden müssen. Die folgenden Befehle ergeben nicht das selbe Resultat wie das oben erwähnte Beispiel.

> ajc path/to/src/*.java

> ajc path/to/aspects.aj

Da solche Aufrufe schnell unübersichtlich werden können wird bei komplexeren Projekten Ant oder Maven verwendet.
Im Inneren des Compilers werden selbst die Source Dateien zuerst in Bytecode umgewandelt und erst dann verwoben. Folglich ist Build-time weaving immer Binary weaving. Build-time source weaving und Build-time binary weaving können deshalb auch kombiniert werden.
> ajc path/to/src/*.java path/to/aspects.aj -inpath application.jar -aspectpath precompiledAspects.jar

Build-time Binary weaving

Build-time weaving bietet sich besonders dann an, wenn man keinen Zugang zum Source code der Applikation (oder der Aspekte) hat.
Es spielt keine Rolle ob die Klassen mit javac oder ajc kompiliert wurden, sogar Aspekte in Annotationsschreibweise können problemlos mit javac kompiliert und anschliessend mit ajc verwoben werden. Aspekte in klassischer Form müssen mit ajc kompiliert werden, da javac die traditionelle AspectJ Syntax nicht versteht.

3.3.2. Load-time weaving

Beim Load-time weaving werden die Aspekte verwoben, während die JVM die Klassen lädt. In neueren Java-Versionen (>5) wird hierfür das Java Virtual Machine Tools Interface (JVMTI) gebraucht. Bei früheren Java Versionen musste ein eigener Classloader verwendet werden, der Interaktionen mit dem Bytecode ermöglicht, bevor die Klasse geladen wird.
Um das ganze dennoch performant zu machen müssen alle Aspekte in einer aop.xml Datei im META-INF Verzeichnis des Classpaths aufgeführt werden.

Beispielaufruf:
> java -javaagent:path/to/aspectjweaver.jar [Optionen] <Main-Klasse>
Mit diesem Aufruf initialisiert die JVM den Weaver-Agent. Dieser kombiniert alle aop.xml Dateien aller angegebenen Classpaths und lädt die darin aufgelisteten Aspekte. Der Rest passiert Event basiert. Der Agent registriert sein Interesse am Class-Loading Event und lässt die JVM den Einstiegspunkt der Applikation laden. Die JVM lädt Klassen sobald sie gebraucht werden und benachrichtigt dabei den Agent der den geladenen Bytecode bei Bedarf verändern und verweben kann.

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. Eclipse (2015)

3.4.2. IntelliJ

Die IntelliJ IDEA der tschechischen Entwicklerfirma Jetbrains wird standardmässig mit einem AspectJ Plugin ausgeliefert. Dieses beinhaltet alles von der Integration des ajc Compilers bis hin zur Codevervollständigung.

3.4.3. Weitere

Auch für Netbeans, Emacs und JBuilder gibt es AspectJ Plugins.

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 erkennt 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 zu haben. 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. Auch der Aspect kann in der Regel ohne Anpassung in mehreren Applikationen wiederverwendet werden. Dies vereinfacht insbesondere bei Unternehmen mit Applikationen ähnlicher Architektur den Betrieb und die Weiterentwicklung derjenigen.

Einfachere Erweiterbarkeit und Anpassung Neue systemübergreifende Anliegen können problemlos und ohne Anpassung des Systems hinzugefügt werden. Bei Anpassungen an bestehenden Modulen reicht es die einzelnen Aspekte anzupassen.[1]

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 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.

[1] 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 Klarheit schaffen indem sie den Ablauf visualisieren. 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. [2] 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 [3]
  - Component-Based Programming[4]
- Dependency injection
- Functional Programming
- Code Generation[5]

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, welche Unternehmen bisher 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 kann. Die Lernkurve ist aber relativ steil. Wir freuen uns darauf AOP in einem realen Projekt anwenden zu können und so einem Praxistest zu unterziehen.

[2] Bruce (2001)
[3] (Laddad, 2009, p 66)
[4] Various (2015)
[5] 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, 1. November 2015	
Namen Vornamen:	Emanuel Knecht	Aeschlimann David
Unterschriften:	.....	.....

Literaturverzeichnis

Afterthought. Afterthought open source framework, 2015. URL https://github.com/vc3/Afterthought.

AspectC. Aspectc++, 2015. URL http://www.aspectc.org/.

AspectSharp. Aspectsharp framework, 2015. URL http://sourceforge.net/projects/aspectsharp/.

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.

Eclipse. Aspectj development tools (ajdt), 2015. URL http://www.eclipse.org/ajdt/.

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.

PostSharp. Postsharp, 2015. URL https://www.postsharp.net/customers.

B. Pratap. Benefits of aop, 2012. URL http://www.tekhnologia.com/2012/01/benefits-of-aspect-oriented-programming.html.

R. S. D. N. (RSDN). Nemerle aop example, 2015. URL https://github.com/rsdn/nemerle/tree/master/snippets/aop/src/ftests.

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-oriente.

Various. Component based programming, 2015. URL https://en.wikipedia.org/wiki/Component-based_software_engineering.

Abbildungsverzeichnis

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. Source Code

A.1.1. Main.java

```
public class Main {
        public static void main(String[] args) {
                System.out.println("Hello ");
        }
}
```

A.1.2. HelloWorld.aj

```
public aspect HelloWorld {

        pointcut MainMethod() : execution(public static void Main.main*(..));

        after() : MainMethod() {
                System.out.println("World!");
        }
}
```

A.1.3. LogAspect.aj

```
public aspect LogAspect {

        private int logCount;
        private void increaseLogCount() {
                logCount++;
        }
        // pointcut <nameOfPointcut>() : <type>(
        //      <accessModifier> <returnType> <Class>.<Method>(<Parameters>)
        pointcut publicMethods() : execution(public * *.*(..));

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

A.1.4. LogAspectAlt.java

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;

@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;
        }
}
```

A.1.5. Visitor.java

```
public interface Visitor{
        void visitPoint(Point p);
}
```

A.1.6. PointVisitorAspect.java

```
public aspect PointVisitorAspect {
        private java.util.ArrayList<Visitor> Point.visitors =
            new java.util.ArrayList<Visitor>();

        public <T extends Visitor> void Point.visit(T v){
                this.visitors.add(v);
                v.visitPoint(this);
        }
}
```

A.1.7. VisitorImpl.java

```
public class VisitorImpl {

}
```

A.1.8. VisitorImplementationAspect.java

```
public aspect VisitorImplementationAspect{
        declare parents: VisitorImpl implements Visitor;

        public void VisitorImpl.visitPoint(Point p){
                System.out.println("visited:"+p.toString());
        }
}
```

A.1.9. PointVisitor.java

```
public @interface PointVisitor {

}
```

A.1.10. PointVisitorImpl.java

```
@PointVisitor
public class PointVisitorImpl {

}
```

A.1.11. PointVisitorImplementationAspect.java

```
public aspect PointVisitorImplementationAspect {
        declare parents: @PointVisitor * extends VisitorImpl;
}
```