

Von Objektorientierung zu Aspektorientierung

AspectJ

Michael Kanis
Technische Universität München

24.06.2008

Zusammenfassung

Dieser Artikel gibt einen Überblick über die Konzepte aspektorientierter Programmierung, sowie über die Features von AspectJ, einer speziellen Umsetzung dieser Programmiermethodik, und beleuchtet praktische Anwendungsbeispiele.

1 Einleitung

Die Erfahrung, dass mit herkömmlichen Programmierparadigmen und -modellen bestimmte Probleme nicht hinreichend gelöst werden können, ist wahrscheinlich die wichtigste Motivation für aspektorientierte Programmierung. So durchsetzen bspw. sicherheitsrelevanter Code und Logging-Ausgaben die Geschäftslogik jeder größeren Anwendung. In der objektorientierten Programmierung, wo Klassen die Einheit für Modularisierung sind, durchziehen solche Querschnittsbelange, oder *Cross-Cutting Concerns*, oft die ganze Anwendung. Eine sinnvolle Modularisierung von solchem Code ist nur schwer oder gar nicht möglich. Aspektorientierung schafft hier Abhilfe, indem sie eine weitere Moduleinheit einführt - die *Aspekte*.

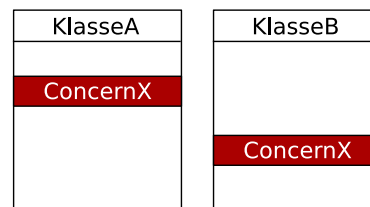
2 Was ist Aspektorientierung?

Benötigt man, z.B. aus Revisionsgründen, in seiner Business-Anwendung umfangreiche Trace-Ausgaben, steht man mit objektorientierten Methoden schnell vor dem Problem, dass die Geschäftslogik von Code

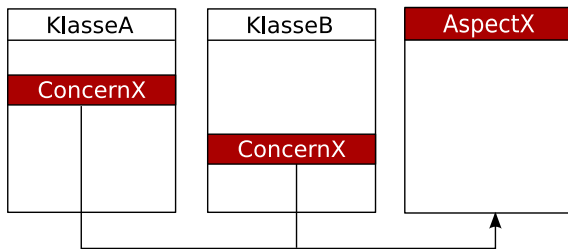
durchgesetzt ist, der mit den jeweiligen Geschäftsfällen nur wenig zu tun hat. Soll bspw. jede Ausführung einer Methode geloggt werden, ist es nötig, am Anfang (und ggf. am Ende) jeder Methode ein Statement zu haben, das den Methodennamen ausgibt.

Sollen bestimmte GUI-Elemente einer Anwendung nur angezeigt werden, wenn der angemeldete Benutzer die richtige Berechtigung dafür hat, muss auch hier bei jedem betroffenen Element eine entsprechende Abfrage stehen.

Mit Objektorientierung lassen sich diese Belange nur unzureichend auslagern. Es lassen sich zwar Module schreiben, die die jeweilige Logik (z.B. zum Prüfen der Legitimation) kapseln, aber der Aufruf eines solchen Moduls muss immer noch aus dem Business-Code heraus erfolgen.



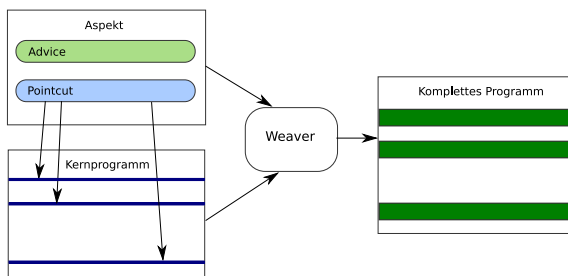
Ein Aspekt dagegen enthält nicht nur den auszuführenden Code, sondern auch eine Liste wohldefinierter Punkte an denen dieser ausgeführt werden soll. Somit lassen sich Cross-Cutting Concerns im Source Code sauber von einander und von Business Concerns trennen.



Um die nun auf Quellcode-Ebene getrennten Belange wieder zu einem kompletten Programm zusammenzufügen, gibt es mehrere Strategien. In einigen Umsetzungen von Aspektorientierung übernimmt diese Arbeit ein Precompiler. Bei AspectC++ [2] wandelt dieser bspw. zunächst den Aspekt-Code in standardkonformen C++-Code um, der dann mit einem herkömmlichen C++-Compiler übersetzt werden kann.

Bei AspectJ kommt jedoch eine andere Technik zum Tragen, das *Bytecode-Weaving*. Dabei wird nach dem eigentlichen Kompilervorgang der Bytecode zu einem "Gesamtkunstwerk" verwoben (engl. to weave = weben). Dieser Vorgang kann direkt nach dem Kompilieren stattfinden, man spricht dann vom *Compile-Time Weaving*. Dann wird die fertige Anwendung auf den oder die Zielrechner ausgerollt, so wie das ohne Aspektorientierung auch der Fall wäre. Man spricht hier auch vom Weben auf Entwicklungsebene.

Noch flexibler ist man mit dem sog. *Load-Time Weaving*, bei dem die Aspekte erst unmittelbar vor dem Laden der Klasse durch den Classloader in das Kernprogramm verwoben werden. Analog spricht man hier vom Weben in der Produktionsebene.



3 Überblick über AspectJ

AspectJ [8] ist eine der ältesten Umsetzungen einer aspektorientierten Sprache für die Java-Plattform. Sie wurde bei Xerox PARC entwickelt, von wo auch die erste objektorientierte Programmiersprache Smalltalk stammt, und später an die Eclipse Foundation übergeben. Dort wird AspectJ jetzt als eigenständiges Projekt weiterentwickelt. Mit den AspectJ Development Tools (AJDT [7]) existiert auch eine gute Unterstützung der Sprache für die Eclipse-IDE.

Das Ergebnis des AspectJ-Weavers ist reiner und vollständig kompatibler Java-Bytecode. Er kann auf jeder Java Virtual Machine ausgeführt werden. Dabei setzt AspectJ seit der Version 1.5 auf Java 5. Vorhergehende Versionen waren auch auf Java 1.4 lauffähig. Auch seit Version 1.5 bietet AspectJ eine umfassende Unterstützung der neueren Java-Features wie Annotations und Generics.

3.1 Inter-Type-Declarations

Statische Joinpoints, oder *Inter-Type-Declarations*, erlauben es dem Aspektprogrammierer die statische Struktur der Anwendung zu verändern. Sie werden daher grundsätzlich zur Compile-Zeit ausgewertet. Mit ihnen ist es möglich, die Funktionalität von Klassen zu erweitern, sie also mit neuen Eigenschaften und Methoden zu "impfen", sowie deren Beziehungen untereinander zu verändern, also in die Klassenhierarchie einzugreifen.

Der folgende Code erweitert die Klasse MyClass um das Feld message und die Methode sayHello().

```
public aspect HelloAspect {
    private String MyClass.message
        = "Hello_world!";

    public void MyClass.sayHello() {
        System.out.println(message);
    }
}
```

Wie man sieht, ist die Syntax sehr nah an normaler Java-Syntax. Der einzige Unterschied besteht darin, dass vor Variablen- und Methodennamen der Name des Typs anzugeben ist, der erweitert (nicht im OO-Sinne) wird.

Möchte man, dass die Klasse `MyClass` das Interface `MyInterface` implementiert, so kann man das mit dem `declare parents`-Statement erreichen. Auf diese Weise kann auch die Superklasse verändert werden.

```
public aspect ParentExtender {
    declare parents :
        MyClass implements MyInterface;
    declare parents :
        MyClass extends java.util.ArrayList;
}
```

Das dritte Feature in diesem Bereich ist das Erstellen eigener Compiler-Meldungen. Dieses fällt eher in den Bereich der Entwicklerunterstützung, kann dort aber sehr nützlich sein, z.B. bei der Umsetzung von Programmierrichtlinien.

```
public aspect NoDirectSql {
    declare error:
        withincode(* com.example.*(..)) &&
        call(* java.sql.*(..)) :
            "SQL not allowed";
}
```

Genauso kann mit `declare warning` eine Compiler-Warnung erzeugt werden.

3.2 Dynamische Joinpoints

AspectJ verfügt über ein sehr durchdachtes dynamisches Joinpoint-Modell. Dazu erweitert es Java um das (gedankliche) Konzept der *Joinpoints*. Dies sind Punkte im Programmablauf, an denen von AspectJ neuer Code eingefügt werden kann. AspectJ kennt viele verschiedene Joinpoints, so z.B. Methodenaufrufe, Feldzuweisungen und -abfragen und Ausnahmebehandlungen. Der Begriff Joinpoint ist also nur ein neuer Name für etwas, das Java-Programmierer ohnehin schon kennen.

Zusätzlich führt AspectJ aber auch einige neue Sprachkonstrukte für dieses Modell ein. Diese sind in erster Linie *Pointcuts* und *Advices*.

Pointcuts

Pointcuts dienen dazu, eine Menge von Joinpoints zu selektieren. Hier kann man mit regulären Ausdrücken bspw. bestimmte Methoden auswählen.

Die generelle Syntax von Pointcuts ist wie folgt:

```
pointcut Name ( Parameter ) : Expression;
```

Dabei kann die Pointcut-Expression aus einem oder mehreren mit `&&` bzw. `||` verknüpften Ausdrücken bestehen. Die Teilausdrücke können mit `!` negiert werden. Die wichtigsten sind:

`call (Signature)` wählt eine Methode nach Ihrer Signatur aus. Die Signatur kann mit regulären Ausdrücken versehen sein. Der call-Joinpoint befindet sich beim Aufruf der entsprechenden Methode, also noch im Quelltext des Aufrufers.

`execution (Signature)` funktioniert ähnlich wie ein call-Pointcut, aber der Joinpoint ist unmittelbar vor der Ausführung der Methode und damit schon in deren Kontext.

`get (Signature)` und `set (Signature)` wählen Joinpoints, an denen auf Variablen zugegriffen wird. Dabei steht `get` für einen lesenden Zugriff und `set` für eine Wertzuweisung.

`handler (TypePattern)` wählt Exception-Handler, also `catch{}`-Blöcke, die Ausnahmen von mit `TypePattern` beschriebenen Typen behandeln.

`this (Type or Id)` trifft dann zu, wenn das gerade ausführende Objekt (`this`) vom Typ `Type` oder dem Typ des mit `Id` bestimmten Parameters ist.

`target (Type or Id)` ist ähnlich wie `this`, nur das hier nicht auf das `this`-Objekt selektiert wird, sondern auf das Objekt, auf dem eine Methode aufgerufen wird, bzw. eine Wertzuweisung erfolgt.

`args (Type or Id, ...)` funktioniert analog zu `this` und `target`, selektiert aber Methodenparameter, bzw. zuzuweisende Werte.

Eine vollständige Auflistung findet sich im AspectJ Programming Guide [14].

Advices

Advices bestimmen das Verhalten eines dynamischen Aspekts. D.h. in ihnen wird definiert, welcher Code

ausgeführt werden soll, wenn ein bestimmter Joinpoint im Programm erreicht wird.

Ihre Syntax lautet:

```
AdviceSpec [ throws TypeList ] : Pointcut {
    Body
}
```

Dabei referenziert `Pointcut` einen definierten `Pointcut`, `Body` enthält normalen Java-Code. `AdviceSpec` kann einer der Folgenden sein.

`before (Formals)` wird direkt vor dem Joinpoint ausgeführt.

`after (Formals) [returning | throwing] [(Formal)]` wird nach einem Joinpoint ausgeführt. Die Schlüsselwörter `returning` und `throwing` bestimmen dabei, ob der Advice nur ausgeführt werden soll, wenn die Methode ordnungsgemäß beendet, oder eine Ausnahme wirft. Lässt man beide weg, wird er in jedem Fall ausgeführt.

`around (Formals)` wird um einen Joinpoint herum ausgeführt. Das bedeutet, dass er die Kontrolle vor dem Joinpoint übernimmt und dann mit `proceed(...)` über dessen Ausführung bestimmen kann. Mit diesem Advice kann auch auf den Rückgabewert des Joipoints noch Einfluss genommen werden.

Die vollständige Erklärung findet sich wiederum im AspectJ Programming Guide.

4 AspectJ in der Praxis

4.1 Programmierrichtlinien

Programmierrichtlinien spielen eine wichtige Rolle bei der Einhaltung von Qualitätskriterien in Software-Projekten, gerade in mittleren bis großen Entwicklerteams. Natürlich können solche Richtlinien überhaupt nur dann nützlich sein, wenn diese auch umgesetzt werden. AspectJ kann mit Hilfe von Inter-Type-Declarations helfen, sie durchzusetzen. Dies kann u.a. dazu dienen, den Zugriff auf bestimmte APIs aus einzelnen Code-Teilen

heraus zu unterbinden. Mit diesen Features ist eine viel feinere Steuerung von Zugriffsbeschränkungen möglich, als mit den Modifiern `public`, `protected` und `private`. So kann man damit auch Methoden- und Variablenzugriffe aus verschiedenen Teilen einer Software-Architektur, wie dem Model-View-Controller-Muster, steuern.

Als Beispiel soll hier eine Zugriffsbeschränkung für das JDBC-API dienen. Denkbar ist ein Szenario, indem es den Entwicklern nicht erlaubt ist, SQL-Anfragen direkt auszuführen, sondern nur über einen Object-Relational-Mapper wie Hibernate.

```
public aspect NoDirectSql {
    declare error:
        withincode(* conventions.*(..)) &&
        call(* java.sql.*(..)) :
            "Direct use of SQL not allowed.";
}
```

4.2 Design by Contract

Design by Contract ist eine Methodik, bei der schon während der Phase des Anwendungsdesigns bestimmte Vereinbarungen über das Laufzeitverhalten von Methoden gemacht werden. Dazu beschreiben Vor- und Nachbedingungen, welche Werte für die Parameter einer Methode zulässig sind und welche für deren Rückgabewert. So kann bspw. eingegrenzt werden, dass eine Methode `Integer add(Integer, Integer)` nicht null zurückgeben darf.

Durch die Kombination von AspectJ mit den in Java 5 eingeführten Annotations kann eine sehr elegante Umsetzung dieses Paradigmas erfolgen.

In folgendem Beispiel wird für alle Methoden die mit der Annotation `@NotNull` markiert sind, überprüft, ob ihr Rückgabewert ungleich `null` ist.

```
public aspect NotNullPostCondition {
    pointcut notNull():
        call(@NotNull !void *(..));

    after() returning(Object o) : notNull() {
        String MSG = ...
        if (o == null) {
            throw new IllegalStateException(MSG);
        }
    }
}
```

Dazu schreibt man einfach die leere Annotation `NotNull`, die nur zur Markierung dient.

```
public @interface NotNull {}
```

Wird nun eine Methode `calc` definiert, die die Geschäftslogik beinhaltet, kann man diese einfach mit `@NotNull` markieren und der Aspekt sorgt zur Laufzeit für die Einhaltung des Kontrakts.

```
@NotNull public Integer calc() {  
    // Berechnungen  
    ...  
    return null; // IllegalStateException  
}
```

Auf diese Weise können auch weit komplexere Nachbedingungen realisiert werden. Vorbedingungen sind natürlich ebenso möglich, wenn auch etwas komplizierter, da Pointcuts in AspectJ es derzeit nicht ermöglichen auf Methoden mit Annotations versehenen Parametern zu selektieren. Die Technik ist sehr ausführlich in dem Artikel *“Design by Contract”* [9] aus dem *eclipse* magazin 12/07 beschrieben. Leider scheint das darin erwähnte *ContractJ*-Projekt nicht mehr verfügbar zu sein. Eine andere Implementierung, die eine ähnliche Technik nutzt ist aber *Contract4J* [3].

4.3 Sicherheitsaspekte

Sicherheitsrelevanter Code ist ein typisches Beispiel für Cross-Cutting Concerns. Solchen Code der Form *“wenn Benutzer a die Berechtigung b hat, führe Code c aus, sonst wirf eine Exception”* hat sicherlich jeder schon gesehen. Mit Aspektorientierung lässt sich dieser relativ einfach auslagern.

Möchte man zum Beispiel die Buttons einer Swing-Anwendung je nach Berechtigung des Nutzers aktivieren oder deaktivieren, kann man z.B. einen Pointcut auf die Methode `add` nutzen, um schon beim Hinzufügen des Buttons zu seiner Elternkomponente diesen mittels `setEnabled(false)` zu deaktivieren.

Etwas eleganter wäre sicher ein Pointcut auf die `isEnabled()`-Methode, da sich auf diese Weise noch zur Laufzeit die Legitimation ändern kann. Andererseits müssen so aber auch ggf. nötige repaints

berücksichtigt werden.

Die gleiche Methode gilt natürlich ebenso für Menüs und andere GUI-Elemente. Denkbar ist somit auch, ganze Dialoge für den Benutzer nur dann aufrufbar zu machen, wenn er die nötigen Rechte hat. Da alle Widgets in Swing von der Klasse `java.awt.Component` erben, die die Methode `isEnabled()` vorgibt, kann diese Art der Sicherheitsüberprüfung sehr generisch erfolgen.

Mit anderen GUI-Toolkits kann das natürlich ähnlich umgesetzt werden, wie dies der Artikel *“Security Does Matter”* [12] ausführlich erläutert. Hier wird sogar auf der Eclipse RCP-Plattform aufgebaut. Dazu muss allerdings, aufgrund des komplexen Classloadings in Eclipse, einiges beachtet werden. Das Problem liegt dabei darin, dass der Classloading-Mechanismus der Eclipse-Plattform nicht von Haus aus die Abbildung von Abhängigkeiten zwischen Aspekten und Klassen in verschiedenen Bundles erlaubt. Aber auch hier existieren bereits Ansätze, dieses Problem zu lösen. Dank der flexiblen Erweiterbarkeit von Eclipse, ist es dem *AJEER*-Projekt [1] möglich dessen Classloading um AspectJ Loadtime-Weaving-Fähigkeiten zu erweitern.

4.4 Caching

Ein weiteres Beispiel für einen typischen Cross-Cutting Concern, der von Anfang an in die Planung einer Anwendung einfließen kann, ist das Caching von Daten. Die Umsetzung mittels Aspekten erhöht gerade hier die Möglichkeit der Wiederverwendung enorm.

Ein Cache ist ein schneller Zwischenspeicher für Daten, deren Bereitstellung normal wesentlich länger dauern würde. Dabei kann es sich um aufwendige Berechnungsergebnisse handeln, oder auch um Dateien die von Festplatte geladen oder über das Netzwerk übertragen werden.

Im Allgemeinen hat man dazu ein Modul, das das Laden (oder die Berechnung) übernimmt. Will man nun mit reinem Java ein Caching einbauen, muss man

dieses Modul mit den Belangen des Caches, also Laden aus und Speichern in dem Cache, durchsetzen. Eine elegantere Methode lässt sich mit AspectJ umsetzen. Hierzu wird ein Pointcut auf die Methode definiert, die die Daten bereitstellt. Ein dazu entwickelter around-Advice kann nun vor dem Ausführen des Joinpoints im Cache nachschauen, ob das zu ladende Objekt bereits im Cache liegt. Ist es zwischengespeichert, wird es direkt ausgegeben und der Joinpoint nie ausgeführt. Ist es nicht vorhanden, wird mit `proceed(...)` der Joinpoint aufgerufen. Danach muss nur noch das geladene Objekt in den Cache gespeichert werden, damit es beim nächsten zugriff vorhanden ist.

Eine einfache Implementierung eines solchen Caches für eine Methode, die Image-Objekte aus einer URL lädt, könnte wie folgt aussehen.

```
public aspect ObjectCache {

    private HashMap cache = new HashMap();

    protected Object get(Object key) {
        SoftReference reference = cache.get(key);
        return reference != null ?
            reference.get() : null;
    }

    protected void put(Object key,
        Object value) {
        cache.put(key, new SoftReference(value));
    }

    pointcut loadImage(URL url) :
        execution(Image Map.loadImage(URL)) &&
        args(url);

    Image around(URL url) : loadImage(url) {

        Image image = get(url);

        if (image != null) {
            return image;
        }

        proceed(tile);

        if (tile.getImage() != null) {
            put(tile.getUrl(), tile.getImage());
        }
    }
}
```

4.5 Design Patterns

Ein etwas fortgeschritteneres Beispiel für die Anwendung von aspektorientiertem Design ist die Umsetzung von Design Patterns mittels Aspekten.

Dabei wird eine möglichst wenig intrusive Umsetzung der Design Patterns der GoF [6] angestrebt. Als Ziel kann es angesehen werden, die beteiligten Objekte absolut ohne Änderungen an ihrer eigentlichen Logik in das jeweilige Muster einzubinden. Einige dieser Umsetzungen beschreiben mehrere Artikel der Serie AOP@Work auf IBM's DeveloperWorks-Site [4].

Das Adapter-Pattern soll als einfaches Beispiel dienen, um diese Methodik zu erklären.

Im Adapter-Pattern werden unterschiedliche (oft schon bestehende) Klassen, mit ähnlicher Funktionsweise in Adapter-Klassen gewrappt, um ihnen ein einheitliches Interface nach außen zu geben.

Man stelle sich eine Software für die Instrumente eines Autos vor. Der Sensor an der Tachowelle, der die Geschwindigkeit liefert und der Sensor am Schwimmer für den Ölstand sind bereits gegeben.

```
public class Tacho {

    public Double getSpeed() {
        ...
    }
}

public class Oelmesser {

    public Double getOelstand() {
        ...
    }
}
```

Ein Anzeigeelement soll nun beide Werte anzeigen. Aufgrund der unterschiedlichen Signaturen für die Sensoren geht dies aber nicht generisch, sondern muss für jeden Sensor extra ausprogrammiert werden.

Mit Intertype-Declarations von AspectJ lässt sich aber in einem Aspekt ein Interface deklarieren, dass eine einheitlich Methode bietet. Dann müssen Tacho und Oelmesser dieses Interface implementieren.

```
public interface ISensor {
```

```

    public Object getValue() {
        ...
    }
}

public aspect Sensor {

    declare parents :
        Tacho implements Sensor;
    declare parents :
        Oelmesser implements Sensor;

    public Double Tacho.getValue() {
        return getSpeed();
    }

    public Double Oelmesser.getValue() {
        return getOelstand();
    }
}

```

Nun kann das Instrument die Werte mit einem einheitlichen Aufruf auslesen.

```

...
public void showValues() {

    for (ISensor s : sensors) {
        System.out.println(s.getValue());
    }

}
...

```

Diese Lösung ist deshalb so elegant, weil sie den bestehenden Code (die beiden Sensoren) nicht verändert (non-intrusiv) und einfach erweiterbar ist. Der Concern ist zentral an einer Stelle gehalten und nicht über Module verteilt.

5 Fazit

Mit Aspektorientierung und speziell für Java-Entwickler mit AspectJ lässt sich die Modularisierung von Software auf das nächste Level heben. Die Vorteile, kürzere Entwicklungszeiten, bessere Strukturierung, einfachere Wartbarkeit und damit kosteneffektivere Software, liegen auf der Hand. Dennoch steht dem für den geeigneten Entwickler, wie bei jeder neuen Technik oder Methodik, ein gewisser Aufwand

entgegen, um sich in die neue Denkweise einzuarbeiten.

Dabei erscheint eine schrittweise Adoption, wie von Ron Bodkin [5] vorgeschlagen, äußerst sinnvoll.

Leider ist die zum momentanen Zeitpunkt aktuelle Version 1.5 der AspectJ Development Tools für Eclipse noch etwas fehleranfällig. Da es sich dabei aber um ein bei der Eclipse Foundation selbst gehostetes Projekt handelt ist ein zügiges Fortschreiten durchaus wahrscheinlich. Deshalb ist auch zu erwarten, dass die IDE-Integration immer die Features der gerade aktuellen Version von AspectJ unterstützen wird.

Literatur

- [1] AJEER. <http://sourceforge.net/projects/ajeer>.
- [2] AspectC++. <http://www.aspectc.org/>.
- [3] Contract4J. <http://www.contract4j.org/>.
- [4] IBM DeveloperWorks. <http://www.ibm.com/developerworks/>.
- [5] Ron Bodkin. AOP@Work: Next steps with aspects. *IBM DeveloperWorks*, 2006. <http://www.ibm.com/developerworks/java/library/j-aopwork16/>.
- [6] Erich Gamma, Richard Helm, Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1995.
- [7] The Eclipse Foundation. AspectJ Development Tools. <http://www.eclipse.org/ajdt/>.
- [8] The Eclipse Foundation. The AspectJ Project. <http://www.eclipse.org/aspectj/>.
- [9] Heiko Seeberger, Andreas Wagner. Design by Contract. *eclipse magazin*, 12 2007.
- [10] Nicholas Lesiecki. AOP@Work: Enhance design patterns with AspectJ. *IBM DeveloperWorks*, 2005. <http://www.ibm.com/developerworks/java/library/j-aopwork5/>.

- [11] Martin Lippert, Markus Völter. Die 5 Leben des AspectJ. *JavaSpektrum*, 3 2004.
- [12] Peter Friese, Martin Lippert, Heiko Seeberger. Security Does Matter. *eclipse magazin*, 12 2007.
- [13] The AspectJ Team. AspectJ Developer's Notebook.
<http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html>.
- [14] The AspectJ Team. AspectJ Programming Guide.
<http://www.eclipse.org/aspectj/doc/released/progguide/index.html>.