Aspektorientierte Programmierung – Einführung

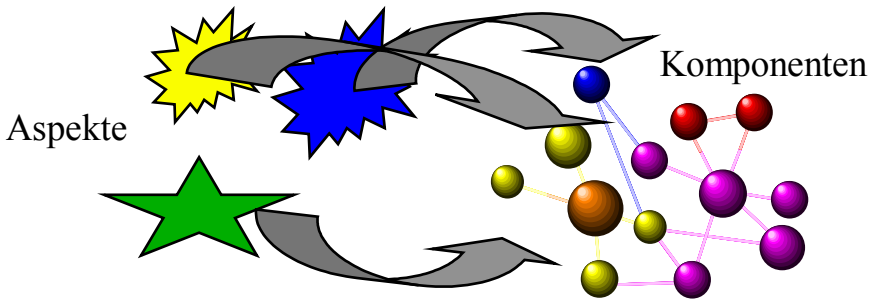
Thomas Grzenkowski
sithgze@stud.informatik.uni-erlangen.de

Motivation

Die Wiederverwendung von Klassen und (Teil-) Systemen blieb selbst durch Verwendung von OOP (object oriented programming) in der Praxis hinter den Erwartungen zurück.[i] Grund dafür ist, dass sich gewisse Anforderungen nicht in eigene Klassen kapseln lassen. Das sind z.B. Anforderungen wie Persistenz, Fehlerbehandlung, Kommunikation, Prozeßsynchronisation. Dies wiederum bewirkt, dass die Klassen an den aktuellen Einsatzkontext gebunden sind, welches bei Wiederverwendung für ein weiteres System aber ein anderer sein kann. Damit liegt es nahe sich um diesen Einsatzkontext zu kümmern: Durch Trennung von Belangen auf Implementierungsebene. D.h. man versucht diese Belange / Aspekte zu erkennen und auszulagern. Hilfe bietet hier AOP (aspect-oriented programming): 1984 trat Gregor Kiczales in PARC (Palo Alto Research Center[ii] von Xerox) ein und begann an der Entwicklung von AOP. Interessenten können ein Blick in seine Artikel [iii] und [iv] werfen, die erste Beispiele mit Hilfe von Lisp aufzeigen.

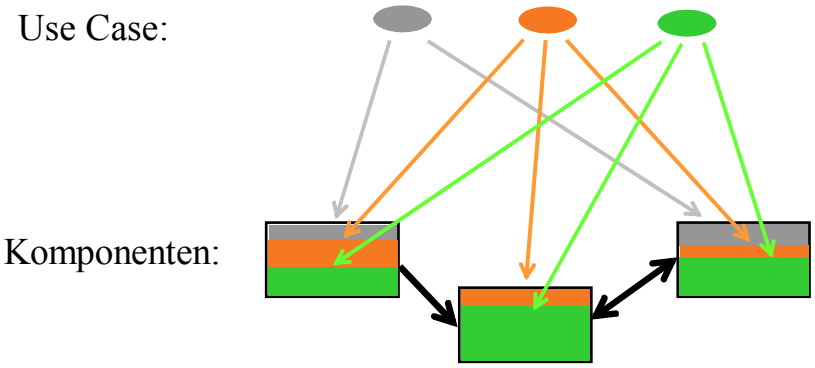
AOP

AOP gehört zu den POP (post-object programming) Technologien.[v] Zu POP gehört neben AOP z.B. auch domänenspezifische Sprachen, generative und generische Programmierung, Reflektion und Metaprogrammierung.[vi] AOP liefert uns den benötigten Ansatz zum Erkennen und Abtrennen von diesen Aspekten. Es erlaubt aber auch verschiedene Aspekte in existierende Anwendungen zu weben. Diese Aspekte stellen für gewöhnlich einen Querschnitt (crosscut) zur objekt-orientierten Struktur dar.[vii] D.h. der Code wird nicht vertikal herausgezogen, wie z.B. wenn man eine Unterklasse erstellt, sondern stellt einen horizontalen Schnitt dar:[viii]



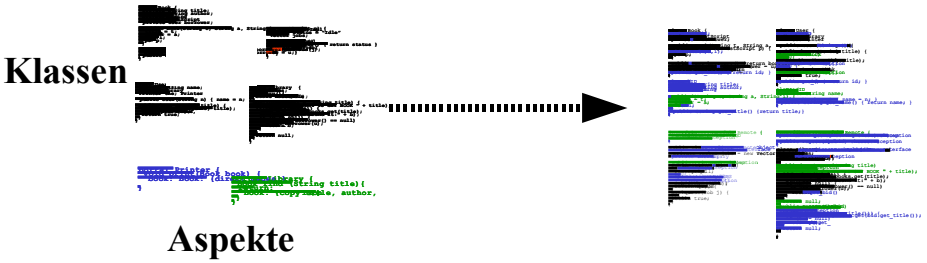
Zeichnung 1: Aspekte und Komponenten

Betrachten wir dies mal mit Hilfe von drei Anwendungsfällen (use cases):[ix]



Zeichnung 2: Anwendungsfälle

Einerseits haben wir für alle drei Anwendungsfälle gemeinsamen Code für alle Komponenten (nicht extra visualisiert). Aber z.B. für den „grauen" Anwendungsfall brauchen wir in der ersten und dritten Komponente zusätzlichen Code (visualisiert durch die grauen Balken in den Komponenten). Dieser Code wird wiederum von dem „orangen" und „grünen" Anwendungensfall nicht benötigt. Damit stellen sie einen Aspekt dar. Durch die Separierung dieser Aspekte mit Hilfe von AOP können diese Komponenten, je nach Anforderungsfall angepasst, gebaut werden ohne jedes mal neue Komponenten bauen zu müssen.
Schauen wir uns das mal auf Codeebene an:



Zeichnung 3: Aspekte und der Code

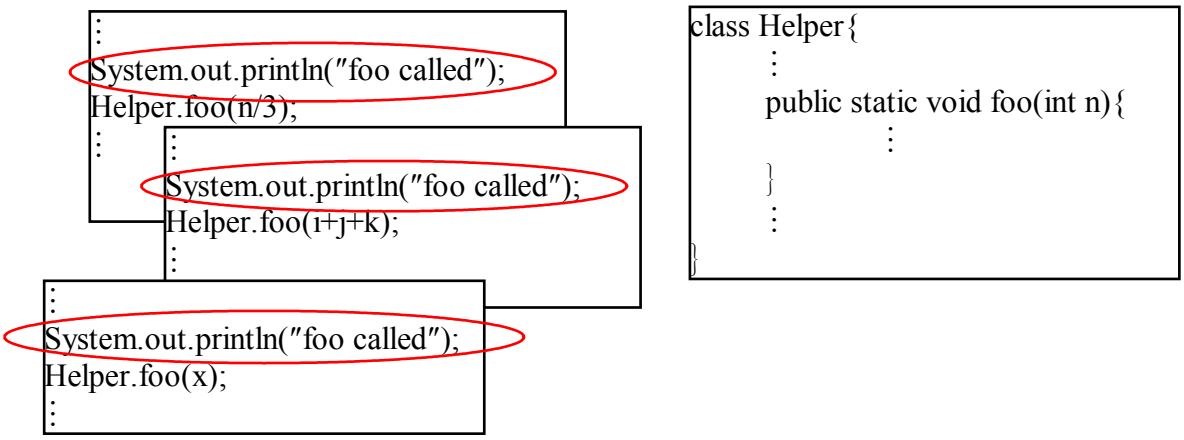
Links ist der Code wie wir mit Hilfe von AOP entwickeln: Der gemeinsame Code von allen Anwendungsfällen in den Klassen selber und der Code von den jeweiligen Aspekten. Rechts ist der Code wie er bisher aussah, bzw. wie er aussehen könnte nachdem er durch einen Weber zusammengesetzt wurde, dazu aber erst später. Wie man sieht wird der Code kürzer und übersichtlicher.
Um ein besseres Verständnis und Gefühl zu bekommen ein paar Beispiele:

Codebeispiele[x]

1. Beispiel:

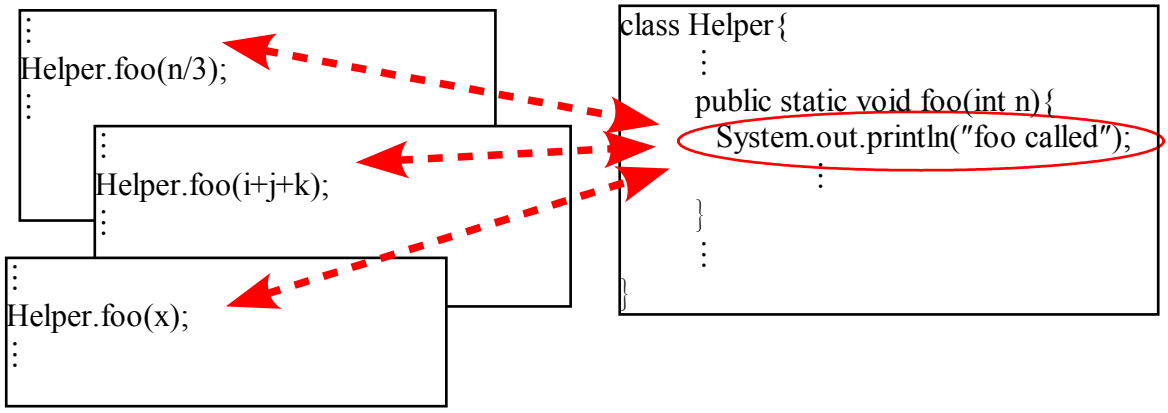
Wir haben folgendes Szenario: Vor jedem Aufruf der Methode „*foo*" erfolgt im Code immer ein und derselbe lokale Aufruf: „*System.out.println("foo called")*". Für die Wartbarkeit ist dies allerdings sehr schlecht, denn wenn dieser Aufruf geändert werden soll, so muss er an mehreren Stellen im

Code händisch gesucht und ersetzt werden. Können wir diesen Fall mit AOP lösen oder gibt es andere dafür geeignete Lösungen?



Zeichnung 4: Codebeispiel 1 - Problemskizze

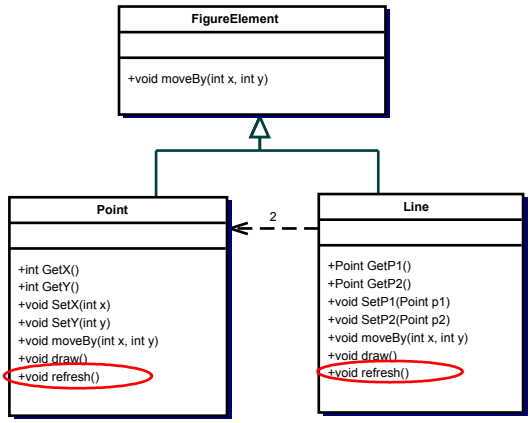
Eine sehr einfach Lösung wäre einfach den lokalen Aufruf in die Methode „*foo*“ mit hinein zu ziehen, so dass sich die Methode selber um den Aufruf kümmert. AOP ist hier also an und für sich nicht nötig. Aber sein Einsatz ist durchaus auch in diesem Beispiel berechtigt: Was wäre wenn wir die Methode nicht verändern könnten? Oder die Klasse Helper in verschiedenen Anwendungsfällen zum Einsatz käme, wo aber nicht in allen Fällen der Aufruf von „*System.out.println*“ erwünscht wäre? Mit Hilfe von AOP könnten wir auch den Code an einer Stelle ablegen, aber abhängig vom jeweiligen Einsatz entscheiden, ob vor dem Aufruf der Methode „*foo*“ unser Aspekt ausgeführt werden soll. Man könnte z.B. diese Codezeile als Debug – Aspekt betrachten.



Zeichnung 5: Codebeispiel 1 - Lösung

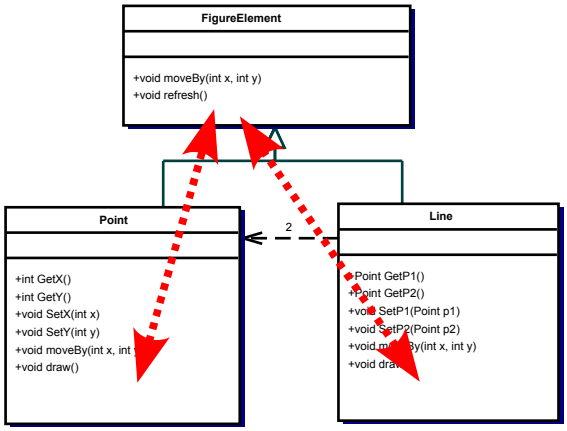
2. Beispiel:

Wir haben die Oberklasse „*FigureElement*“ und ihre beiden Unterklassen „*Point*“ und „*Line*“. Beide Unterklassen implementieren die Methode „*refresh*“ auf die gleiche Weise. Wieder wollen wir den Code möglichst nur einmal vorliegen haben.



Zeichnung 6: Codebeispiel 2 - Problemskizze

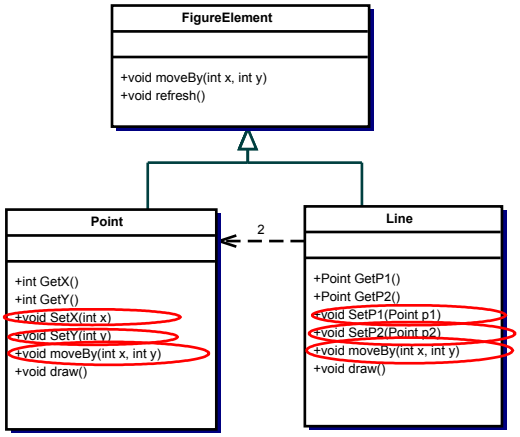
Dafür bietet die Vererbung die beste Lösung: Man zieht die Methode in die Oberklasse.



Zeichnung 7: Codebeispiel 2 - Lösung

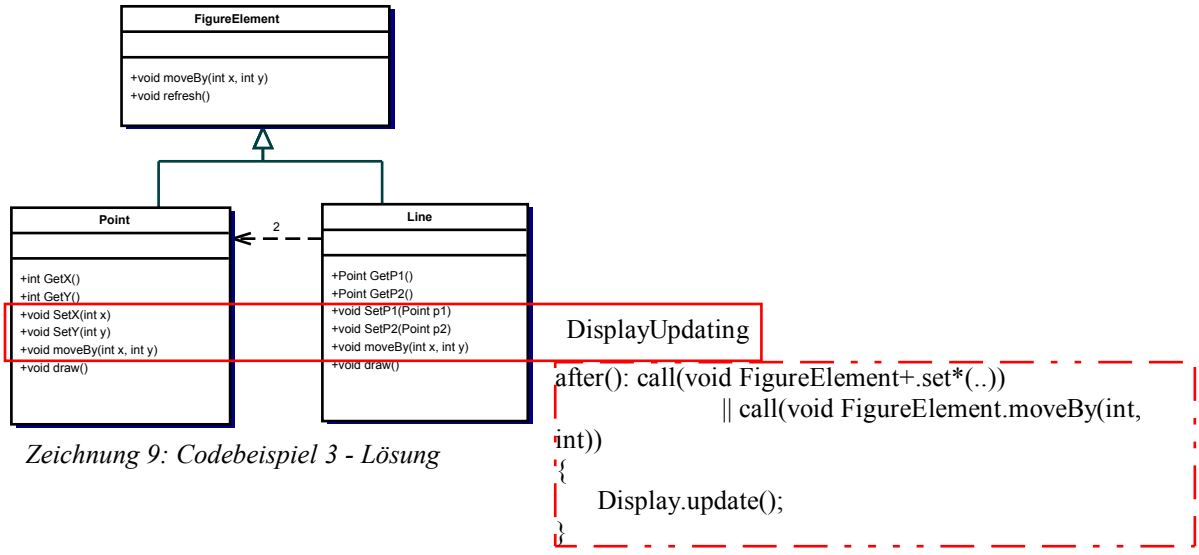
3. Beispiel:

Alle markierten Methoden führen am Ende den gleichen Code aus: „*Display.update();*“
Die Funktionen habe weder Namen noch den restlichen Code gemeinsam.



Zeichnung 8: Codebeispiel 3 - Problemskizze

Müßte man dies nur mit Hilfe vom objekt-orientierten Anzatz lösen, so könnte man dies nur, in dem man den Aufruf in eine extra Methode auslagert, die man in der Oberklasse ablegt. Lößt man jedoch dieses Problem mit Hilfe von AOP, bringt dies z.B. den Vorteil, dass man diesen Aspekt auch nach Bedarf weglassen kann: D.h. man kann z.B. die Objekte nur zum Verwalten verwenden, sprich sie müßten sich in diesem Fall um keine Aktuallisierung der Darstellung am Bildschirm kümmern, da sie nicht dargestellt würden.



Zeichnung 9: Codebeispiel 3 - Lösung

Crosscutting[xi] [xii]

Im letzten Beispiel sahen wir sehr gut wie sich ein Aspekt quer über die Codehierarchie erstrecken kann. Jedoch können wir zwischen verschiedenen Formen von Crosscutting unterscheiden:

- **Statisch:** Es werden zu bereits existierenden Typen neue Operationen hinzugefügt.
- **Dynamisch:** An definierten Stellen werden Aspekte aufgerufen.

Dynamisches Crosscutting:

Um zu zeigen wie das Definieren von Stellen, an denen der Aspektcode ausgeführt werden soll,

aussehen kann, hier ein kurzer grober Einblick in die Syntax von AspektJ.

Zuerst einmal die Definition der wichtigsten Begriffe:

Join points: z.B. Methodenaufruf, Feldzugriff, Klassen- / Objektinitialisierung

Pointcut = join points + [Rückgabewerte] + [Methoden] + [Parameter]

z.B.: call(void Point.setX(int))

Advice: wann(Parameter) : pointcuts { Code }

Schauen wir uns mal das Ganze in einem ersten Bsp. an:

```
Pointcut moves():
        call(void FigureElement.incrXY(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int));

after(): moves() {
        flag = true;
}
```

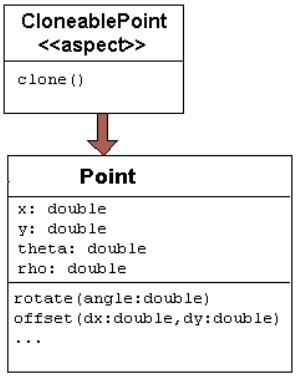
Zuerst werden der Übersicht wegen unter dem Namen „*moves*“ mehrere pointcuts zusammen gefaßt. Danach wird mit Hilfe des Advices festgelegt, dass nach all diesen pointcuts der Code „*flag = true;*“ ausgeführt wird. Dieser Code hat uns einen ersten Überblick verschafft. Allerdings ist er noch nicht vollständig, sondern wird in AspectJ in einem „aspect“-Objekt abgelegt. Ein Bsp. zu diesem Objekt werden wir bei statischem Crosscutting sehen. Nur soviel noch dazu: In diesem Objekt lassen sich Variablen deklarieren, auf welche unser Code aus dem Bsp. zu greifen kann: „*flag*“.

Da bei Methoden auch Parameter übergeben werden, möchte man diese evtl. auch in den Aspekten nutzen können. Ein kleines Bsp. dazu:

```
before(Point p, int nval):
        call(void p.setX(nval)) {
                System.out.println("x value of " + p +
                                                " will be set to " + nval + ".");
}
```

Dies bewirkt dass vor jedem Aufruf der Methode setX die Parameter ausgegeben werden.

Statisches Crosscutting:[xiii]



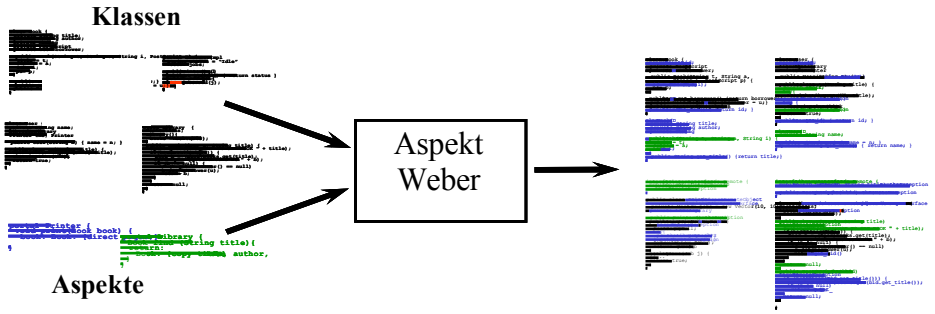

```
public aspect CloneablePoint {
        declare parents: Point implements Cloneable;

        public Object Point.clone() throws
                CloneNotSupportedException {
                // we choose to bring all fields up to
                //date before cloning.
                makeRectangular();
                makePolar();
                return super.clone();
    }
}
```

Zeichnung 10: Hinzufügen von clone()

In diesem Beispiel möchten wir Point um die Methode „*clone*“ mit Hilfe von statischem Crosscutting erweitern. Dafür legen wir ein „aspect“-Objekt an. Nun können wir dem Point sowohl das Interface „*Cloneable*“, als auch die Methode „*clone*“ hinzufügen.

Aspekt Weber:[xiv]



Zeichnung 11: Aspekt Weber

Ein Weber ist für das Zusammenfügen von Aspekten und Code zuständig. Dabei kann man zwei Arten von Aspektweben unterscheiden: Statisch und dynamisch.[xv]

Statisches Weben:

Der Code wird durch einen Präprozessor, Compiler oder einer Laufzeitumgebung einmalig mit einander verwebt. Dass heißt die **Verknüpfungen** sind danach **nicht mehr änderbar**. Dies hat den Vorteil, dass der **Code** für seinen Einsatz **optimiert werden kann**: z.B. dass der Compiler Inlinecode daraus macht. Sollte es von einem Objekt aber meherer Ausprägungen geben, so muss der Compiler allerdings auch meherer Versionen davon bauen, was sich in einem größeren Speicherverbrauch bemerkbar macht, als beim dynamischen.

Dynamisches Weben:

Dynamisches Weben geschieht zur Laufzeit: Eine Laufzeitumgebung verknüpft dynamisch vor jeder

Ausführung die Aspekte mit dem Code. Dadurch ist zu jeder Zeit **eine andere Verknüpfung möglich**: Man kann das selbe Objekt jederzeit einer anderen Aspektkompination ausführen. Z.B. bei Bedarf den Debuging-Aspekt de-/aktivieren. Allerdings setzt dies eine Laufzeitumgebung vorraus, welche zusätzlich Zeit und Ressourcen benötigt um diese Verknüpfungen ständig neu zu tätigen. Auch eine Codeoptimierung ist nicht in dem Mase möglich wie bei statischem Weben.

i Zeitschrift iX 5/1999 S. 74 - 78
ii Homepage von Parc http://www.parc.xerox.com/
iii Aspect-Oriented Programming von Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin.
z.B. zu finden unter: http://citeseer.nj.nec.com/cache/papers/cs/16616/http:zSzzSzwww.parc.xerox.comzSzcslzSzgroupszSzsdazSzpublicationszSzpaperszSzKiczales-ECOOP97zSzfor-web.pdf/kiczales97aspectoriented.pdf
iv A Case-Study for Aspect-Oriented Programming von Anurag Mendhekar, Gregor Kiczales, John Lamping.
z.B. zu finden unter: http://www2.parc.com/csl/groups/sda/publications/papers/PARC-AOP-RG97/for-web.pdf
v COMMUNICATIONS OF THE ACM – October 2001/Vol.44, No. 10
vi Aspect-Oriented Programming von Tzilla Elrad, Robert E. Filman, Atef Bader.
Artikel in COMMUNICATIONS OF THE ACM – Ausgabe October 2001/Vol. 44, No. 10
vii Zeitschrift IX - Ausgabe 5/1999 S. 74 – 78
viii Vortrag: „An Aspect-Oriented Design Approach“ von Mehmet Aksit
ix Vortrag: „Use Cases and Aspects Working Seamlessly Together“ von Ivar Jacobson (Firma Rational)
x Vortrag: „The fun has just begun“ von Gregor Kiczales
xi Paper: ECOOP 2001 – An Overview of ApectJ (0.8)
z.B. zu finden unter: http://www.parc.com/groups/csl/projects/aspectj/index.html
xii AspectJ Quick Reference
z.B. zu finden unter: http://www.eclipse.org/aspectj/
xiii Internetseite: Basic Techniques – Examples
http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/ch03s03.html#sec:RolesAndViews
xiv Vortrag von Gregor Kiczales über AOP
xv Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. von K. Czarnecki – Kapitel 7: Aspect-Oriented Decomposition and Composition