Was bringen AOP und GP für die Industrie

Lutz Dominck

Siemens AG, CT SE 2
Corporate Technology
Software and Engineering, Architecture
D-81730 München
Lutz.Dominick@mchp.siemens.de

Zusammenfassung. Neue Paradigmen bergen oft die langersehnte Lösung für bekannte Probleme. Aspekt-Orientierte Programmierung (AOP) und Generative Programmierung gehören zu einer Reihe neuer Paradigmen für Design und Programmierung von Software-Systemen, die neue Arten von Modularität zur Verfügung stellen. AOP erweitert Objekt-Orientierung und Komponenten-Orientierung nahtlos. AOP arbeitet bei Design und Implementierung mit Aspekten. Aspekte kapseln Eigenschaften und Anforderungen eines Software-Systems, die dieses an mehreren Stellen beeinflussen und somit durchdringen (cross-cutting concerns). Diese neue Art Kapselung war bislang nicht möglich. Durch Objekt-Orientierung und Komponenten ist es nur in wenigen Fällen möglich, die Aspekte eines Systems unabhängig voneinander und vom Gesamtsystem getrennt zu modellieren. Neue Werkzeuge, wie z. B. der Weaver AspectJ™, verbinden auf Anforderung die Aspekte mit dem Gesamtsystem. Fehlt ein Werkzeug, wie bei C++, können Techniken wie GP eingesetzt werden. Praxis-Beispiele für Aspekte sind End-To-End Quality of Service, Thread-Safety, Monitoring, History und Transaktionsbehandlung. Die Werkzeuge und Programmiertechniken für AOP erlauben Design und Implementierung von Aspekten in Projekten für die Programmiersprachen Java und C++.

1 Motivation

Software-Systeme werden immer größer und komplexer. Die Erfahrung zeigt, dass die Mittel der objekt-orientierten Programmierung (OOP) und des Komponenten-Ansatzes nicht ausreichen, auf alle Probleme des Designs eine optimale Antwort zu geben. Objekte und Komponenten, Klassifizierung und Spezialisierug geben noch keine Garantie dafür, dass alles, was zusammen gehört, auch gekapselt werden kann. Anforderungen wie End-To-End Quality of Service durchziehen viele Teile eines Systems. Sowohl bei der erstmaligen Erstellung des Systems als auch später, bei Änderung der Anforderung oder des Systems, ergeben sich Probleme wie Verständlichkeit, Redundanz und Konsistenz des Codes, die die Aufwände für Erstellung und Änderungen in die Höhe treiben. Eine Kapselung dieses Typs von Anforderungen in sogenannten *Aspekten* stellt eine Lösung dar.

Was sind *Aspekte*? Hat Aspect-Oriented Programming (AOP) etwas mit Object-Oriented Programming (OOP) zu tun? Diese Fragen sollen anhand zweier kurzer und anschaulicher Beispiele untersucht werden.
TechReview zählt AOP zu den zehn Technologien, die die Welt in Zukunft verändern werden [29].

2 Der Aspekt "Monitoring"

Im ersten Beispiel wird eine Java-Applikation an ein Test- und Monitoring-Werkzeug angeschlossen. Mit diesem Werkzeug kann man z. B. Interaktionsdiagramme dynamisch aufzeichnen und den Zeitverbrauch von lokalen und entfernten Aufrufen messen.
Die Applikation hat für die Werkzeuganbindung normalerweise keine Vorkehrungen getroffen. Außerdem soll dieser Anschluss nicht permanent bestehen bleiben, z. B. wird er nur in der Entwicklungs- und der Testphase gebraucht, aber das ausgelieferte Produkt soll den entsprechenden Programmcode nicht mehr enthalten.

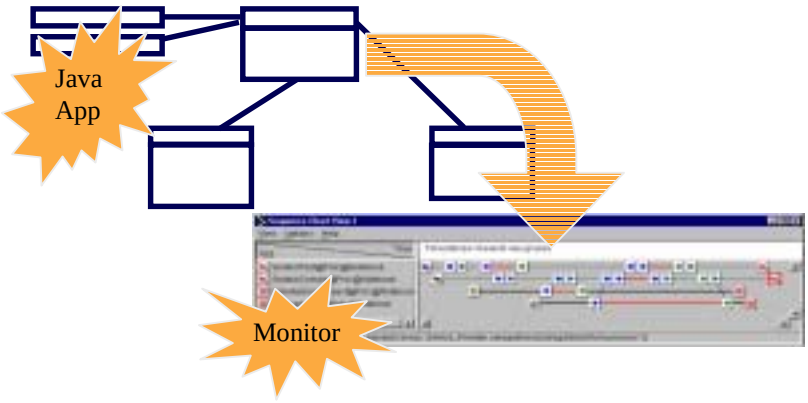


Fig. 1. Die Java-Applikation liefert Laufzeit-Informationen an das Monitoring-Tool

Der Aspekt wird der Java-Applikation hinzugefügt. Monitoring ist jetzt ein Aspekt der Applikation. Die Java-Applikation stellt dem Monitoring-Werkzeug zur Laufzeit geeignete Daten zur Verfügung. Der Benutzer hat das Einfügen des Aspekts Monitoring aber nicht selber programmiert. Vielmehr hat er das Werkzeug für das "Einweben" des Aspekts konfiguriert, um Applikation und Aspekt miteinander zu verbinden. Gleichzeitig könnte er die Menge an Monitoring-Information beeinflussen, um die Auswertung zu beschleunigen,
Die Hersteller von Monitoring-Werkzeugen liefern das Werkzeug zum Einweben des Monitoring-Aspekts ggf. gleich mit. Das bedeutet dann aber, dass in diesem Fall nur dieser eine Aspekt mit einer Applikation verknüpft, also eingewoben, werden kann. Es gibt aber auch Aspekte, die schon während des Designs und der Implementierung

einer Applikation vorliegen. Sollen diese Aspekte eingewoben werden, wird ein universelles Werkzeug benötigt.

3 Der Aspekt "Thread-Safety"

Als weiteres Beispiel betrachten wir deshalb eine Java-Klasse *SimpleMessageQueue*, die eine Warteschlange für Nachrichten implementiert. Stark vereinfacht benötigt diese Klasse die Methoden *put(), get(), isFull()* und *isEmpty()*.

```
public class SimpleMessageQueue {
        public bool put( Object element, TimeValue timeout ) {…}
        public Object get(TimeValue timeout ) {…}
        public boolean isFull() {…}
        public boolean isEmpty() {…}
}
```

Die Implementierung der Methoden soll nicht betrachtet werden. Die Klasse hat in der jetzigen Form folgende Merkmale:

- Sie ist ein abstrakter Datentyp.
- Sie verbirgt hinter ihrer Schnittstelle die grundlegende Implementierung für eine Warteschlange.
- Sie ist für den Gebrauch innerhalb eines Threads eingerichtet, also noch nicht multi-thread-safe.

Bis hierher wurden ausschließlich Prinzipien der OOP angewendet. Diese Prinzipien reichen auch dann noch aus, wenn die Schnittstelle der Klasse erweitert wird, weil Methoden wie *dequeueHead(), dequeueTail()* oder *unget()* und *unput()* benötigt werden. Die funktionale Dekomposition des Problems und der Mechanismus der Vererbung genügen, damit diese zusätzlichen Anforderungen erfüllt werden können.
Wenn die Klasse *SimpleMessageQueue* aber für den Nachrichtenaustausch zwischen Threads eingesetzt wird, müssen weitergehende Anforderungen erfüllt werden.

- Auf den internen Daten von *SimpleMessageQueue* dürfen keine Race-Conditions enstehen, also müssen Zugriffe auf die Methoden serialisiert werden. *SimpleMessageQueue* erhält einen Lock, und jeder Methodenaufruf versucht, zuerst den Lock zu bekommen, bevor er seine eigentliche Funktion erfüllt.
- Beim Lesen auf der leeren Warteschlange soll eine gewisse Zeit gewartet werden, ob nicht doch noch ein anderer Thread eine Nachricht in die Warteschlagen schreibt. Hierzu dient eine NotEmpty-Condition, auf der *SimpleMessageQueue* wartet.
- Beim Schreiben in die volle Warteschlange soll eine gewisse Zeit gewartet werden, ob andere Threads nicht doch noch durch Lesen von Nachrichten wieder genügend Platz für die neue Nachricht schaffen. Hierzu dient eine NotFull-Condition, auf der *SimpleMessageQueue* wartet.

Diese Forderungen sind unabhängig von der Klasse *SimpleMessageQueue*. Sie werden zwar mittels *Lock* und *Conditions* innerhalb der Klasse *SimpleMessageQueue* realisiert, aber sie sind nicht Teil der Funktionalität von *SimpleMessageQueue*. Genau

genommen ist die Erfüllung dieser Forderungen optional und nur abhängig vom Einsatz von *SimpleMessageQueue*.
Ferner wird *SimpleMessageQueue* nicht die einzige Klasse sein, die thread-safe werden soll. Die gleichen Forderungen gelten analog für alle Arten von Containern.
Bei der Realisierung mittels OOP stößt man jetzt aber an gewisse Grenzen.
SimpleMessageQueue benötigt einen *Lock* und zwei *Conditions*. Alle Methodenaufrufe müssen abgefangen werden (per Interzeption), damit zuerst der *Lock* angefordert und dann ggf. noch die richtige *Condition* abgefragt werden kann. Es soll hier nicht Sinn der Erweiterung sein, den Code von *SimpleMessageQueue* direkt zu verändern, auch sollen die Schnittstelle und der Klassennamen erhalten bleiben, da *SimpleMessageQueue* vielleicht schon vielfach eingesetzt wird. Diese Randbedingungen klingen für das sehr kleine und einführende Beispiel etwas gewollt, spiegeln aber letztlich die Anforderungen der Praxis wieder.
Die OOP bietet in einer solchen Situation das Mittel der Vererbung an. Die abgeleitete Klasse enthält *Lock* und *Conditions* und fängt die Aufrufe an die Basisklasse ab. Aber dann hat sich der Klassennamen gerändert. Der Code von *SimpleMessageQueue* ist zwar unverändert, aber dafür muss Applikationscode geändert werden, der *SimpleMessageQueue* einsetzt. Da mehrere Typen von Containern an mehreren Stellen eingesetzt werden, treten diese Situationen auch mehrfach auf.
AOP schlägt nun eine andere Lösung vor.
Die Fähigkeit von *SimpleMessageQueue*, in einer Umgebung mit mehreren Thread klaglos zu funktionieren, nennt man einen *Aspekt von SimpleMessageQueue*. AOP modelliert diesen Aspekt getrennt von *SimpleMessageQueue*. Das ist möglich, weil Thread-Safety nicht zur Funktionalität von *SimpleMessageQueue* gehört. Klasse und Aspekt werden über ein AOP-Werkezeug (z. B. den Weaver von AspectJ™) zusammengefügt, so dass dieser Aspekt in *SimpleMessageQueue* zur Laufzeit vorhanden ist. Dem Quellcode von *SimpleMessageQueue* sieht man das Vorhandensein von Aspekten nicht an und es bleibt dem AOP-Werkzeug überlassen, wie es seine Aufgabe erfüllt.
Die Wirkung ist insgesamt so, als hätte man die Klasse von Hand verändert, z. B. wie in der folgenden Code-Skizze.

```
public class SimpleMessageQueue
{
  //…more fields
  private Mutex lock_ = new Mutex ();
  private Condition notFull_ = new Condition (lock_);
  private Condition notEmpty_ = new Condition (lock_);
  public SimpleMessageQueue(int size)   { … }
  public void put(Object element,TimeValue timeout)
    throws TimeoutException, InterruptedException
  {
    try {
        lock_.acquire ();
        while (this.isFull ())
          notFull_.Wait (timeout);
        //…core functionality…
        notEmpty_.signal ();
        }
    finally {
        lock_.release ();
        }
  }
  public Object get (TimeValue timeout)
       throws TimeoutException, InterruptedException
  {
    Object return_value = null;
    try {
        lock_.acquire ();
        while (this.isEmpty ())
          notEmpty_.Wait (timeout);
        //…core functionality…
        notFull_.signal ();
        }
    finally {
        lock_.release ();
        }
    return return_value;
  }
  //… rest of methods
}
```

Die Code-Skizze zeigt, welcher und vor allem wieviel Code der Klasse hinzugefügt werden muss. Er stellt keine Kernfunktionalität dar, sondern bläht die Klasse eher unnötig auf. AOP nennt diesen Effekt “code tangling”. Der skizzierte Code wird in zusätzlichen Methoden z.T. genauso aussehen, um die die Klasse ggf. erweitert wird. Wenn das Werkezug auch die neuen Methoden thread-safe macht, muss der Entwickler für Klasse *SimpleMessageQueue* weniger implementieren und sein Code konzentriert sich auf den Zweck der Klasse. Wenn der Aspekt “Klasse ist thread-safe”, der "Thread-Safety"-Aspekt, so spezifiziert wird, dass er nicht nur auf Klasse *SimpleMessageQueue* wirken kann, dann entsteht ein zusätzlicher Einsparungseffekt.

4 Was ist ein Aspekt

Das Ziel von AOP ist es also, “den Programmierer darin zu unterstützen, Komponenten und Aspekte sauber voneinander zu trennen, indem Mechanismen zur Verfügung gestellt werden, die einerseits das Auffinden von Aspekten und andererseits das Zusammenfügen aller Teile zum Gesamtsystem ermöglichen.” [4]
Ein Aspekt kann somit aus zwei Blickwinkeln beschrieben werden. Einerseits verkörpert er eine Anforderung, die das System durchdringt (cross-cutting concern). Andererseits stellt er ein Programmkonstrukt dar, z. B. eine Klasse in Java, die die Anforderung kapselt.
Aspekte lagern also bestimmte Anforderungen in eigene Module aus. Dadurch ist es möglich, im Gesamtsystem eine stärkere Trennung der Anforderungen bei Design und Implementierung durchzuhalten. Dabei werden Modul-, Objekt- und Komponentengrenzen in gewisser Hinsicht aufgehoben, da Aspekte auf Objekte zugreifen können, ohne sich dabei auf die öffentliche Schnittstelle der Objekte beschränken zu müssen. Aber eigentlich werden diese Grenzen nicht wirklich verletzt, da der gleiche Aspekt, der auf mehrere Objekte wirkt, diese Objekte unabhängig voneinander erweitert und der Aspekt integraler Bestandteil jedes einzelnen der betroffenen Objekte wird.
Die beiden einführenden Beispiele haben die grundlegende Idee hinter AOP ausgeleuchtet. Um mit Aspekten modellieren und programmieren zu können, sind aber noch einige Betrachtungen vonnöten.
Aspekte modellieren Cross-Cutting Concerns. Das sind Anforderungen, die systemweit auftreten können und nicht Bestandteil des Fachkonzepts sein müssen.

Beispiele für Aspekte sind:

- End-to-End Quality-of-Service-Anforderungen:
  - Sicherheit
  - Fehlerbehandlung und Fehlertoleranz
  - Performanz (nicht nur algorithmische Optimierung) und Echtzeit
- Tracing/Logging und Monitoring/Diagnose/Profiling
- Synchronisation/Thread-Safety
- Caching-Strategien
- Resource-Sharing
- History
- Optimierungen
- Transaktionen
- Konfiguration
- Testanbindung
- Verteilbarkeit und Fernaufrufe
- Common Look&Feel von Benutzungsschnittstellen

Aspekte werden in einem eigenen Designschritt gestaltet und dann z. B. mit den entsprechenden Objekt-Klassen per Deklaration verknüpft. Hierzu benötigt man ein geeignetes Werkzeug, z. B. AspectJ™ [1]. Andere Ansätze mit etwas anderem Verfahren sind z. B. Hyper/J [5] für Java, oder Template Meta-Programming für C++

[15] und Aspekt-Orientierte Programmierung für C++ [25]. Wir beschränken uns hier auf einen kurzen Abriss zu AspectJ™ und verweisen für C++ auf [25].

5 Was ist AspectJ™

AspectJ™ ist eine aspekt-orientierte Erweiterung von Java™. Aspekte werden als Klassen programmiert. Der Unterschied zwischen einer Aspekt-Klasse und der Java-Klasse sind einige neue Schlüsselwörter. Aspekt-Klassen liegen in eigenen .java-Dateien.
Der mitgelieferte AspectWeaver™ ajc ist ein Compiler für Aspekte. Er läuft vor dem Java-Compiler. Anhand der für ihn bestimmten Schlüsselwörter fügt er Aspekt- und Objekt-Klassen zusammen. Beim Aufruf von ajc wird eine Datei angegeben, die alle für den ajc-Lauf benötigten Klassen auflistet. Jeder Aspekt enthält selbst eine Liste von Klassen, auf die er angewendet werden soll. Das Ergebnis des ajc-Laufs sind wiederum .java-Dateien, in der keine AspektJ-Schlüsselwörter mehr vorkommen und die deshalb direkt vom Java-Compiler übersetzt werden können.
Was beinhalten nun die neuen Schlüsselwörter von AspectJ™? Sie beschreiben die Schnittstelle zwischen Aspekt und Objekt. Schnittstelle sind einerseits Methodenaufrufe an ein Objekt, andererseits kann die Klassendefinition durch Hinzufügen von Code erweitert werden.
Wird dann die Methode eines Objekts aufgerufen, kann sich der Aspekt z. B. zuerst aufrufen lassen. Dadurch läuft zuerst die Funktionalität des Aspekts und dann die des Objekts Die umfangreiche Sprachdefinition von AspectJ™ steht unter [1] zur Verfügung. Seit August 2001 ist die Version 1.0 alpha verfügbar, so dass der Einsatz in Produkten endlich in Sicht gerät.

6 Nutzen-"Aspekte"

Damit Aspekte programmiert und eingestzt werden können, wird eine Technologie benötigt. Für Java verwenden wir AspectJ™ [1] und für C++ [25], letzteres ist eine Sammlung von Programmiertechniken, die auf GP-Techniken beruhen und nur den C++-Compiler voraussetzen. Ein Weaver für C++ stand nicht zur Verfügung.
Können Aspekte wie z. B. *Sicherheit* mit AOP anstatt konventionell implementiert werden, entsteht ein Zeitvorteil im Produkt-Lebenszyklus. Die Einbindung von *Sicherheit* in das System muss nicht mehr programmiert werden, sondern kann durch das zeitsparendere Konfigurieren eines Werkzeugs erfolgen. Bei Änderungen fügt das Werkzeug *Sicherheit* einfach erneut ein. Die Arbeitsschritte hierfür sind bei jedem System recht ähnlich und der Aspekt zum Einfügen von *Sicherheit* enthält bereits das Vorwissen, das jetzt im Aspekt dokumentiert ist und auf maximal effiziente Weise wiederverwendet wird. Die Funktionalität von *Sicherhiet* selber muss natürlich weiterhin programmiert werden.
Betrachten wir den Einsatz von Komponenten, so ergibt sich in der Zusammenfassung das folgende Bild. Der Einsatz von Komponenten (ob EJB, COM oder CORBA components) steht und fällt mit der in der Komponente enthaltenen

Funktionalität. Ggf. müssen der Komponente vor ihrem Einsatz im Produkt sowohl Aspekte als auch Funktionalität hinzugefügt werden. Bei uns laufen Arbeiten über die Korrelation von Aspekten und Komponenten, z. B. betreffs Erweiterung von Komponenten mittels AOP-Technologien ([19], [23]). Die Erfahrung zeigt, dass der Aufwand für die Vorausplanung von Erweiterbarkeit von Komponenten (z. B. für User Interfaces) und die zugehörige Implemntierung eines Framework-Produkts sehr groß sind. AOP und GP bieten hier eine Alternative an.
Der Effekt, der durch das Substituieren von Framework-Anteilen durch AOP/GP entsteht, wird als "Architecture Tunneling Effect" in [26] beschrieben. Langfristig entsteht dadurch die Perspektive, die Last einer stets ungenauen Vorausplanung bei (generischen) Frameworks deutlich zu mindern und die Kosten für Bau und Einsatz dieser Frameworks zu senken. Dies wird entsprechend für bestimmte Teile bei Komponenten-Architekturen gelten.

7 Offene Punkte

Webt man einen oder mehrere Aspekte ein, dann stellt sich die Frage nach dem Laufzeitverhalten. Aspekte benötigen Laufzeit für den zusätzlichen Code und einen zusätzlichen Methodenaufruf, da AspectJ mit Interzeption arbeitet. Bei mehreren Aspekten muss die Reihenfolge ihrer Ausführung festgelegt werden, sofern sie das gleiche Objekt betreffen. Die Entscheidung, die beim Design mit Aspekten getroffen wird, muss durch ein Tests untermauert werden.
Von Hand eingefügter Code enthält stets die richtigen Reihenfolge der Methodenaufrufe. Der Weaver-Lauf selbst benötigt keine bestimmte Reihenfolge der Aspekt-Klassen oder der Java-Klassen, um arbeiten zu können. Aber zur Design-Zeit ist ohne Vorgaben an den Weaver nicht immer bestimmbar, in welcher Reihenfolge die Aspekte zur Laufzeit an einem Objekt ausgeführt werden. Der Weaver legt ohne spezielle Vorgaben eine Reihenfolge fest. Für dieses Problem ist die optimale Lösung noch nicht gefunden.
Das Auslagern von Anforderungen in Aspekte kann die Komplexität der Verständlichkeit des Gesamtsystems erhöhen. Diese steigt weiter durch bewusste Abhängigkeiten zwischen Aspekten. Für das Geflecht zwischen Objekten und Aspekten wird eine IDE-Unterstützung benötigt. Für den Programmierer ist das Maß an code tangling in jedem Fall aber sichtbar geringer geworden.
Ein Test des Gesamtsystems wird umso wichtiger, je mehr Aspekte eingewoben werden, und das galt natürlich schon immer auch für die von Hand kodierten Aspekte.
Die Auslieferung von AspectJ™ [1] enthält eine aspekt-orientierte Version des Observer-Patterns. Dies leitet zu der Frage über, in welcher Beziehung Aspekte und Design Patterns zueinander stehen und welche Idiome es für den Einsatz von Aspekten gibt. Auch hierüber laufen Untersuchungen.
Eine ganz anderes Diskussionsthema ist der Zeitpunkt des Weaving selbst. Wir haben Compile-Time-Weaving betrachtet, in der Java-Code als Zwischenprodukt erzeugt werden kann. Für C++ geht [25] ebenfalls von Compile-Time-Weaving aus. Es gibt aber auch die Möglichkeit, zur Lade- oder Laufzeit Aspekte einzuweben. Hierfür ist

dann eine andere Werkzeugunterstützung nötig, sowohl ein anderer Weaver als auch eine andere IDE-Unterstützung.

8 Ausblick

Die Aspekt-Orientierte Programmierung steht noch am Anfang. Die Erfahrungen aus vielen Projekten "vor AOP" lassen hinter AOP aber eindeutig Vorteile vermuten, zumal Aspekte an sich nicht neu sind. Verschiedene Fallstudien ([12], [13]) und andere Arbeiten (z. B. [7], [15], [19], [20], [21], [23], [26], [27], [28]) zeigen bereits ein Spektrum von Erkenntnissen und lassen eine detaillierte Diskussion anhand von Erfahrungen aus der Praxis als sehr sinnvoll erscheinen. Der Weg zum Nutzen durch Einsatz von AOP/GP zeigt sich deutlicher durch wichtige Anwendungsfälle und identifizierte Nutzen-Effekte.
Bei den Untersuchungen hat sich die vorhandene Werkzeugunterstützung als hilfreich erwiesen. AspectJ™ z. B. definiert eine universelle Spracherweiterung, eine IDE-Anbindung für z. B. Microsofts JBuilder wird mit ausgeliefert. Viele Fragen nach geeigneten Analyse-, Design-, Codierungs-, Debug- und Testmethoden und passenden Werkzeugen müssen aber weiterhin genauso beantwortet werden wie die Frage nach dem optimalen Einsatz in Produkten und die optimale Anbindung an den Entwicklungsprozess.
Ein sorgfältiger Vergleich mit den anderen neuen Paradigmen wie z. B. in [12], [14], [19] und insbesondere in [15], wird ebenfalls zur Weiterentwicklung von AOP beitragen.

9 Literatur

[1] AspectJ™ http://aspectj.org und Links zu vielen Veröffentlichungen unter http://www.parc.xerox.com/csl/projects/aop/

[2] C.Lopes, G. Kiczales, A Language Framework for Distributed Programming, Technical Report SPL97-010, P9710047 Xerox Palo Alto Research Center, Palo Alto, CA, February 1997

[3] K.Mehner, A.Wagner, An assessment of Aspect Language Design, STJA99, Erfurt, Germany, 1999

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin, Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, Palo Alto, CA, February 1997.

[5] Peri Tarr, Harold Ossher, "Hyper/J user and installation manual", http://www.research.ibm.com/hyperspace/, 1999

[6] Adaptive Programming http://www.ccs.neu.edu/research/demeter/

[7] Lutz Dominick, Aspekt-Orientierte Programmierung im Überblick, In: JavaSpektrum 4/2000, S. 43-48

[8] Generative Programmierung http://home.t-online.de/home/Ulrich.Eisenecker/, http://www.generative-programming.org and http://www.prakinf.tu-ilmenau.de/~czarn/
[9] Composition Filters Object Model http://wwwtrese.cs.utwente.nl/sina/cfom/index.html
[10] Intentional Programming http://www.research.microsoft.com/ip
[11] Harold Ossher, Peri Tarr. Operation-level composition: A case in (join) point, position paper at the ICSE '98 workshop on Aspect-Oriented Programming, 1998
[12] L. Bergmans, M. Aksit. Analyzing Multi-Dimensional Programming in AOP and Composition Filters, position paper at the OOPSLA99, workshop on Multi Paradigm Design, 1999, Webseite siehe [9]
[13] Gail Murphy (Home-Page), University of British Columbia, Kanada, Gail Murphy hat mehrere Studien zu AOP durchgeführt http://www.cs.ubc.ca/spider/murphy/index.html
[14] Karl Lieberherr, Northeastern University, Boston, MA, Conncections between Demeter/Adaptive Programming and Aspect Oriented Programming (AOP) http://www.ccs.neu.edu/home/lieber/connection-to-aop.html
[15] K. Czarnecki, U. Eisenecker. Generative Propramming. Methods, Tools, and Applications. Addison-Wesley 2000
[16] U. Eisenecker, F. Blinn. K. Czarnecki. A solution to the constructor-problem in mixin-based programming in C++. Workshop "Template-Based Programming", Erfurt 2000, see http://oonumerics.org/tmpw00/
[17] B. McNamara, Y. Smaragdakis, Static interfaces in C++. Workshop "Template-Based Programming", Erfurt 2000, see http://oonumerics.org/tmpw00/
[18] B. Stroustrup, Wrapping C++ Member Function Calls, In: C++ Report 6/2000, S. 9-16, 45
[19] L. Dominick, K. Ostermann, Supporting Extension of Components with new Paradigms, workshop paper, OOPSLA 2000
[20] S. Herrmann, M. Mezini, PIROL: A Case-Study for Multi-Dimensional Separation of Concerns in Software Engineering Environments. In *Proc. of OOPSLA 2000*, pp. 188-208.
[21] M. Mezini, L. Seiter, K. Lieberherr, Component Integration with Pluggable Composite Adapters. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer Academic Publishers, 2000
[22] Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer Academic Publishers, 2000
[23] K. Ostermann, Independent Extensibility - an open challenge for AspectJ and Hyper/J, workshop paper, ECOOP2000
[24] B. Tekinerdogan. Synthesis-Based Software Architecture Design, PhD Thesis, Dept. of Computer Science, University of Twente, Kluwer Academic Publishers, 2000

[25] K. Czarnecki, L. Dominick, U. Eisenecker, Multiple Aussichten - Aspekt-Orientierte Programmierung in C++, dreiteilige Artikelserie, iX Ausgaben 8-10, Heise Verlag, 2001
[26] L. Dominick, Tunneling, Aspects and GP for C++,, workshop paper, ECOOP2001
[27] S. Herrmann, M. Mezini, K. Ostermann, Joint efforts to dispel an approaching modularity crisis, workshop paper, ECOOP2001
[28] K. Ostermann, M. Mezini, Object-Oriented Composition is Tangled, workshop paper, ECOOP2001
[29] TechReview, http://www.techreview.com/articles/jan01/TR10_kiczales.html