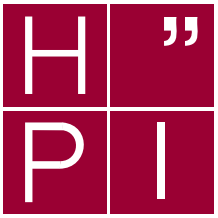
HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam



Aspektorientierte Programmierung

Überblick über Techniken und Werkzeuge

Janin Jeske, Bastian Brehmer, Falko Menge, Stefan Hüttenrauch, Christian Adam, Benjamin Schüler, Wolfgang Schult, Andreas Rasche, Andreas Polze

Technische Berichte Nr. 14

des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 14

Aspektorientierte Programmierung

Überblick über Techniken und Werkzeuge

Janin Jeske, Bastian Brehmer, Falko Menge, Stefan Hüttenrauch, Christian Adam, Benjamin Schüler, Wolfgang Schult, Andreas Rasche, Andreas Polze

Potsdam 2006

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Redaktion: Janin Jeske, Bastian Brehmer, Falko Menge, Stefan Hüttenrauch, Christian Adam, Benjamin Schüler, Wolfgang Schult, Andreas Rasche, Andreas Polze

Email: {janin.jeske; bastian.brehmer; falko.menge; stefan.huettenrauch; christian.adam; benjamin.schueler}@student.hpi.uni-potsdam.de
{wolfgang.schult;andreas.rasche; andreas.polze}@.hpi.uni-potsdam.de

Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@uni-potsdamde
http://info.ub.uni-potsdam.de/verlag.htm

Druck: allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

Heft 14 (2006)
ISBN 3-939469-23-8
ISBN 978-3-939469-23-0
ISSN 1613-5652

Aspektorientierte Programmierung - Überblick über Techniken und Werkzeuge

B.Sc. Janin Jeske, B.Sc. Bastian Brehmer, B.Sc. Falko Menge,
B.Sc. Stefan Hüttenrauch, B.Sc. Christian Adam, Benjamin Schüler,
Dipl.-Inf. Wolfgang Schult, Dipl.-Inf. Andreas Rasche, Prof. Dr. Andreas Polze

{janin.jeske|bastian.brehmer|falko.menge|stefan.huettenrauch|christian.adam|
benjamin.schueler|wolfgang.schult|andreas.rasche|andreas.polze}@hpi.uni-potsdam.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

In den vergangenen Jahren hat sich die Objektorientierung als treibende Kraft in der Softwareentwicklung etabliert. Objektorientierte Modellierungssprachen wie *UML* und objektorientierte Programmiersprachen wie *Java*, *C++* und *C#* haben die Softwareentwicklung stark beeinflusst und werden wohl auch in Zukunft eine Schlüsselrolle spielen. Allerdings hat sich auch gezeigt, dass der objektorientierte Entwurf seine Grenzen hat. Insbesondere das Auftauchen von modulübergreifenden Belangen zerstört unweigerlich jede wohl definierte Modularität. So stellt man schnell fest, dass Anforderungen wie Logging oder Sicherheit bzw. Konzepte wie Fehlerbehandlung und Synchronisation nicht sauber in eine objektorientierte Struktur integriert werden können, sondern im Ergebnis vielmehr über den gesamten Code verteilt sind. Insbesondere für die Wiederverwendbarkeit, Wartung und Erweiterbarkeit der Softwaresysteme bedeutet dies einen nicht unerheblichen Mehraufwand.

Die Ansätze zur Aspektorientierten Programmierung versuchen durch neue Modularisierungskonzepte genau dies zu vermeiden. Modulübergreifende Belange können als Einheit erfasst und implementiert werden. Dabei werden objektorientierte Techniken nicht ersetzt - vielmehr baut die *Aspektorientierte Programmierung* (kurz *AOP*) auf die *Objektorientierte Programmierung* auf und erweitert sie um neue Konzepte.

Der vorliegende Report soll einen kurze Einführung in das Thema *Aspektorientierte Programmierung* und einen Überblick über den aktuellen Stand der Forschung geben. Hierzu werden aktuelle *AOP* Projekte und Produkte zum Teil ausführlicher betrachtet und bewertet werden. Der Report kann und soll keine allumfassende Bestandsaufnahme zum Thema *AOP* sein. Vielmehr sollen hier aktuelle Trends der Entwicklung und herausragende Technologien dargestellt werden.

2 Aspektorientierte Programmierung

2.1 Ein System als Menge von Eigenschaften

Ein Entwickler entwirft ein System aus einer Menge von Anforderungen. Man kann diese in Anforderungen auf Modulebene (Geschäftslogik) und Anforderungen auf Systemebene klassifizieren. Die Anforderungen auf Systemebene sind meist orthogonal zueinander, überschneiden sich aber mit Anforderungen auf der Modulebene. Zum Beispiel erfordert die Geschäftslogik an verschiedenen Stellen eine Authentifizierung des Nutzers, um Operationen ausführen zu dürfen.

Betrachtet man nun die Implementierung dieses Systems, so kann man sagen, dass die Anforderungen auf Modulebene die funktionalen Eigenschaften des Systems wieder spiegeln. Auf der anderen Seite sind die Anforderungen auf Systemebene durch die modulübergreifenden Eigenschaften oder auch nicht funktionalen Eigenschaften des Systems reflektiert.

Gängige Programmiermethoden erlauben es aber nur, die Anforderungen in einer Dimension zu implementieren. In dieser Dimension wird die Modulebene implementiert. Alles was übrig bleibt (Systemebene), wird zusätzlich in dieser Dimension verteilt. Mit anderen Worten, der Anforderungsraum ist n-dimensional wohingegen der Implementierungsraum nur eindimensional ist. (siehe Abbildung 1)

Typische Beispiele für solche modulübergreifenden Eigenschaften sind:

- Fehlertoleranz,
- Sicherheit,
- Logging,
- Tracing,
- Synchronisation,
- Testen und
- Fehlerbehandlung.

2.1.1 Symptome

Folgende Symptome lassen sich identifizieren, wenn man versucht, die oben beschriebenen Probleme mit gängigen Programmiermethoden zu implementieren:

Fehlende funktionale Kapselung Module in einem Softwaresystem müssen oft verschiedenen Anforderungen genügen. Als Resultat müssen sich die Entwickler gleichzeitig Gedanken um die Implementierung der Geschäftslogik, und z.B. der Sicherheit und der Fehlertoleranz machen. Dadurch entsteht Quellcode, der zwar allen Anforderungen gerecht wird, wo die einzelnen Anforderungen aber nicht mehr als Einheit erkannt werden können. Der Quellcode ist durchsetzt (im Englischen *code tangling*).

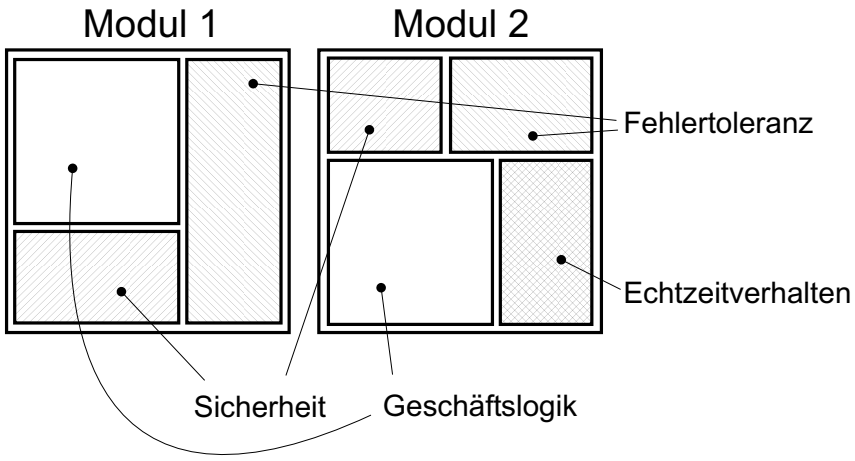


Abbildung 1: Objektorientiertes Modularisierungskonzept

Verstreuter Code Da sich nicht funktionale Eigenschaften über mehrere Module verteilen, ist deren Implementierung auch über mehrere Module verteilt (im Englischen *code scattering*).

Verborgene Eigenschaften Die Eigenschaften der Systemebene (nicht funktionale Eigenschaften) spiegeln sich nicht im Interface der Implementierung wieder.

2.1.2 Schlussfolgerungen

Code tangling und *code scattering* beeinflussen das Softwaredesign und die Softwareentwicklung auf verschiedenste Weise. Hier sind einige Schlussfolgerungen, die daraus gezogen werden können:

Schlechte Wartbarkeit Da der Code für eine Eigenschaft in einem Modul an verschiedenen Stellen implementiert ist (*code tangling*), ist es für den Entwickler schwer, diesen als Einheit zu erfassen und entsprechend zu warten. Auf der anderen Seite, wenn der Code über mehrere Module verteilt ist (*code scattering*), müssen Änderungen parallel an allen Modulen erfolgen.

Kaum Wiederverwendungsmöglichkeit Code, der verteilt ist, lässt sich schlecht für andere Anforderungen wiederverwenden, da er keine eigene Einheit darstellt.

Schlechte Produktivität Könnte man jede Eigenschaft einzeln betrachten und als Einheit implementieren, so entsteht kein zusätzlicher Aufwand durch Probleme, die aus der Verteilung auf und in die einzelnen Module resultieren.

Schlechte Erweiterbarkeit Ein Softwaresystem wird mit den Erkenntnissen von heute implementiert. Sollten jedoch in Zukunft weitere Eigenschaften hinzukommen, gestaltet es sich sehr schwierig, diese zusätzlich zu implementieren.

2.1.3 Ein Lösungsansatz

In der vorangegangenen Diskussion wurde klar, dass es durchaus von Nutzen ist modulübergreifende Eigenschaften als eigenständige Einheiten zu betrachten und zu implementieren. Dies wird in der Literatur häufig als *separation of concerns* bezeichnet. *Aspektorientierte Programmierung* ist eine Methode hierfür und stellt einen Weg dar, um die aufgezeigten Probleme zu lösen. *AOP* bietet die Möglichkeit, modulübergreifende Eigenschaften als eigenständige Einheit (so genannte Aspekte) zu implementieren und aus diesen Implementierungen das Gesamtsystem zu formen. Erst ein Aspektweber fügt funktionale Eigenschaften und Aspekte zusammen.

2.2 Aspekte

Bei der *Objektorientierten Programmierung* sind die meisten nicht funktionalen Eigenschaften eines Softwaresystems über mehrere Methoden oder Klassen im Programmcode verteilt und lassen sich nur schwer getrennt von den funktionalen Eigenschaften kapseln. Dies lässt sich mit den folgenden Begriffen beschreiben:

Überschneidende Belange sind die über mehrere Module im Programmcode verteilten nicht funktionalen Belange an ein Softwaresystem. (im Englischen *cross-cutting concerns*)

Verstreuter Code ist die über mehrere Module im Programmcode verteilte Implementierung der nicht funktionalen Eigenschaften. (im Englischen *scattered code*)

Verworrener Code ist die Implementierung von einem funktionalen und einem oder mehreren nicht funktionalen Belangen an ein Softwaresystem in einem Modul. (im Englischen *tangled code*)

Mit Hilfe der *Aspektorientierten Programmierung* lassen sich die verteilten nicht funktionalen Belange an ein Softwaresystem in Aspekten kapseln. Dadurch wird verstreuter und verworrener Code vermieden.

Aspekte kapseln also die nicht funktionalen Eigenschaften an ein Softwaresystem. Zu einem Aspekt gehört die Definition der Verwebungspunkte (siehe Kapitel 2.2.1) und der Aspektcode.

2.2.1 Verwebungspunkte

Ein Verwebungspunkt (im Englischen *join point*) ist ein Ereignis, dass bei der Ausführung eines Softwareprogramms eintritt. Dabei kann es sich z.B. um einen Methodenaufruf, das Fangen einer Exception oder die Zuweisung eines Wertes handeln. Ein Methodenaufruf zwischen zwei Softwarekomponenten besitzt sechs mögliche Verwebungspunkte:

- vor einem Methodenaufruf,
- anstelle eines Methodenaufrufes,

- nach einem Methodenaufruf,
- vor dem Eintritt in eine Methode,
- anstelle einer Methode und
- nach der Beendigung einer Methode.

Die Definition der Verwebungspunkte finden in der Regel im Aspektcode statt. Dadurch muss der funktionale Programmcode nicht verändert werden.

Verwebungspunkte kennzeichnen die Stellen im Programmcode an denen Aspektcode vom Aspektweber (siehe Kapitel 2.3) eingewoben werden soll. Kommt ein Programm bei seiner Ausführung an einen definierten Verwebungspunkt, dann wird an dieser Stelle der Aspektcode ausgeführt.

2.3 Aspektweber

Aspektweber verweben den funktionalen Programmcode oder Bytecode mit dem Aspektcode an den definierten Verwebungspunkten. (siehe Abbildung 2)

Es gibt vier verschiedene Arten von Aspektwebern:

Quellcode Weber verweben den funktionalen Programmcode mit dem Aspektcode vor der Übersetzung des Programmcodes. (im Englischen *source code weaver*)

Bytecode Weber verweben den Aspektcode mit dem compilierten funktionalen Programmcode.(im Englischen *byte code weaver*)

Ladezeit Weber verweben den funktionalen Bytecode eines Programmes mit dem Aspektcode zur Ladezeit. (im Englischen *link time weaver*)

Laufzeit Weber verweben den Aspektcode mit dem geladenen funktionalen Bytecode dynamisch zur Laufzeit. (im Englischen *run time weaver*)

Die Quellcode und die Bytecode Weber werden in dem Oberbegriff statische Aspektweber zusammengefasst. Die Ladezeit und Laufzeit Weber werden auch dynamische Aspektweber genannt.

2.4 Vorteile Aspektorientierter Programmierung

Die *Objektorientierte Programmierung* ermöglicht die Modularisierung von funktionalen Belangen an ein Softwaresystem in Klassen. Die *Aspektorientierte Programmierung* ermöglicht zusätzlich die Modularisierung von nicht-funktionalen Belangen in Aspekten. Mit Hilfe von *AOP* ist also eine mehrdimensionale Modularisierung möglich. Durch die Modularisierung von nicht-funktionalen Eigenschaften soll der strukturelle Aufbau eines Softwaresystem übersichtlicher werden, wodurch die Wartbarkeit und Wiederverwendbarkeit von Programmcode verbessert werden soll. Außerdem ist es möglich funktionalen und nicht funktionalen Programmcode separat voneinander zu entwerfen, zu implementieren und zu testen.

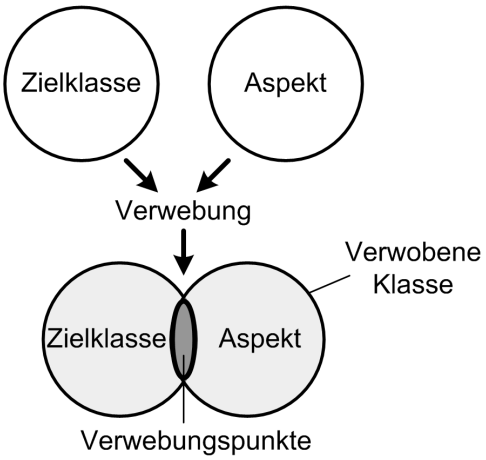


Abbildung 2: Verwebung von Aspekt und Zielklasse

2.5 Kategorisierung der Techniken und Werkzeuge für Aspektorientierte Programmierung

Die verschiedenen zur Verfügung stehenden Techniken und Werkzeuge zur *Aspektorientieren Programmierung* lassen sich entsprechend ihrer Anwendbarkeit in den verschiedenen Phasen der Softwareentwicklung diesen zuteilen. Die Abbildung 3 kategorisiert die im Rahmen dieses Dokumentes untersuchten Techniken und Werkzeuge.

Für die Analysephase wurden einige unterstützende Werkzeuge zur Identifizierung und Visualisierung von Aspekten in großen Softwaresystemen untersucht. Für die Designphase wurden verschiedene Programme untersucht, mit deren Hilfe unter anderem automatisch aus dem Entwurf eines Softwaresystems Quellcode generiert werden kann. Außerdem wurden Techniken untersucht, die das Design von aspektorientierten Softwareprogrammen erleichtern. Für die Implementierungsphase wurden viele verschiedene Werkzeuge untersucht, die unterschiedliche Ansätze verfolgen hinsichtlich des Zeitpunktes der Verwebung. (siehe Kapitel 2.3)

Des weiteren lassen sich die untersuchten Werkzeuge für *AOP* entsprechend den Programmiersprachen die sie unterstützen unterteilen (siehe Abbildung 4).

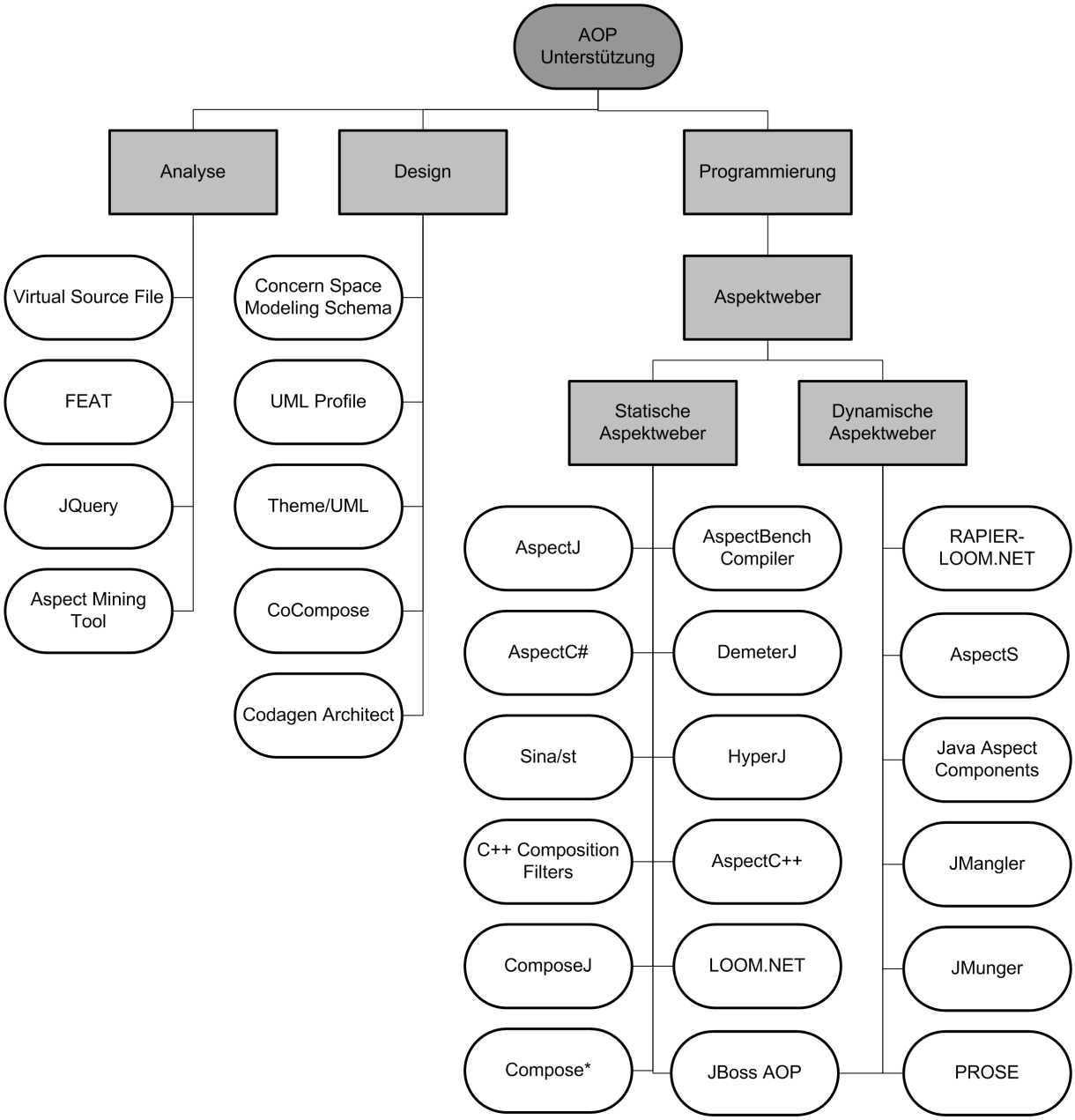


Abbildung 3: Kategorisierung der Techniken und Werkzeuge für Aspektorientierte Programmierung anhand des Prozesses der Softwareentwicklung

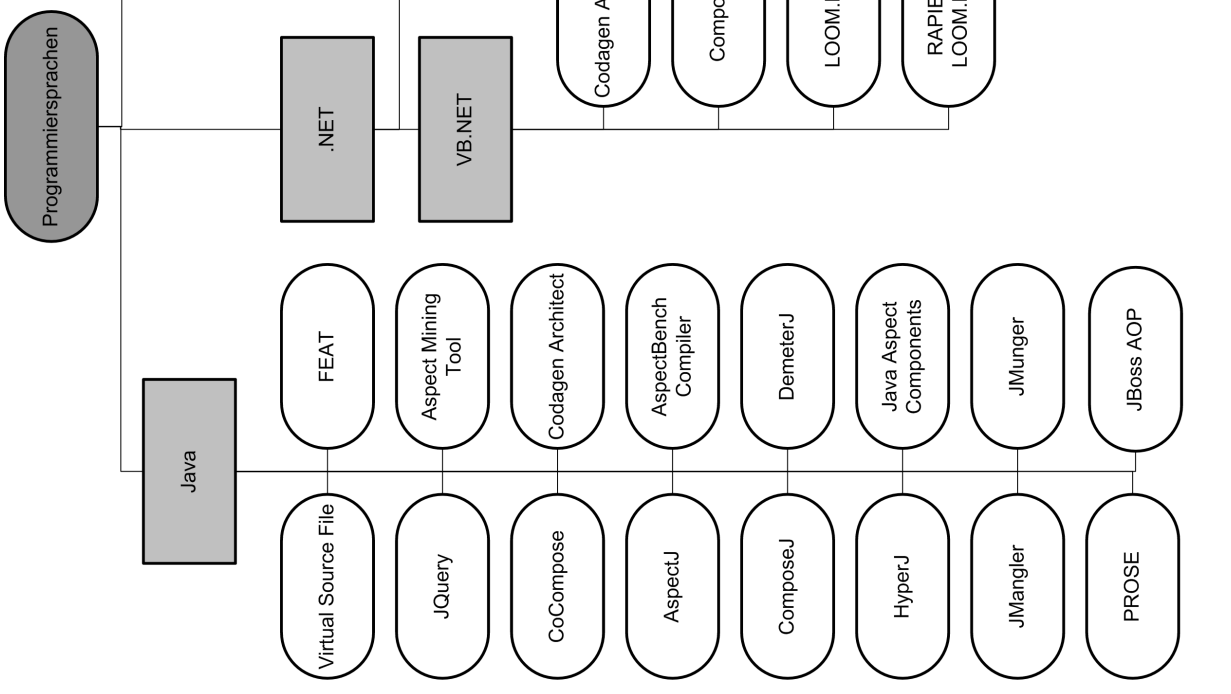


Abbildung 4: Kategorisierung der Werkzeuge Aspektorientierter Programmierung anhand der unterstützten Programmiersprache

3 Techniken und Werkzeuge zur Analyse Aspektorientierter Softwareprogramme

„Die Analysephase in der Softwareentwicklung hat das Ziel ein konsistentes, vollständiges, eindeutiges und realisierbares Modell des Fachkonzepts eines Softwaresystems zu erstellen.“ [1] Dazu müssen die funktionalen Anforderungen und die nicht funktionalen Belange an ein Softwaresystem ermittelt werden. Dabei können entsprechende Werkzeuge insbesondere zur Ermittlung von nicht funktionalen Belangen an ein Softwaresystem sehr hilfreich sein, von denen einige im Folgenden vorgestellt werden.

3.1 Virtual Source File

Virtual Source File ist ein Werkzeug zur Konfiguration und Verwaltung von Softwaresystemen. Es unterstützt die Verwaltung von Änderungen und die Versionierung von Softwareprojekten. Außerdem kann Quellcode nach bestimmten Kriterien untersucht werden. Somit ist es möglich die nicht funktionalen Belange eines Softwaresystems (siehe Kapitel 2.2), die über mehrere Module im Programmcode verteilt sind, zu identifizieren. Außerdem kann nach bereits implementierten Aspekten eines Softwaresystems im gesamten Quellcode gesucht werden. Die Ergebnisse einer Untersuchung werden in virtuellen Quelldateien gespeichert.

Virtual Source File ist aus dem *Eclipse Stellation Projekt* entstanden. Als Plugin für *Eclipse* ist es in der Version 0.5.0 verfügbar, die Mitte 2002 veröffentlicht wurde.

Virtual Source File ist nur begrenzt einsetzbar. Es gibt keine Editierungsmöglichkeit der virtuellen Quelldateien. Außerdem ist eine nachträgliche Anpassung der Suchkriterien nicht möglich.

3.2 FEAT

FEAT unterstützt die Identifizierung, Beschreibung und Analyse von Aspekten. Es ermöglicht das manuelle Hinzufügen einzelner Klassen, Methoden oder Felder zu Aspekten. Des Weiteren werden die Elemente eines Aspektes angezeigt und es können andere Elemente eine Softwaresystems ermittelt werden, die mit den Elementen eines Aspektes interagieren. Außerdem ermöglicht *FEAT* die Darstellung von Aspekten in einem Aspektgraphen. Darin werden die Aspekte selbst mit ihren Elementen und ihre Beziehungen zu anderen Elementen dargestellt. Außerdem werden die Elemente eines Aspektgraphen auf den zugrunde liegenden Quellcode abgebildet.

FEAT wurde in einem Projekt an der University of British Columbia entwickelt. Es ist in der Version 2.5.4 als Plugin für *Eclipse* 3.0 verfügbar.

FEAT besitzt ausführliche Hilfe über das *Eclipse* Hilfe-Menü. Ein Nachteil ist, dass es nicht möglich ist den Quellcode aus dem Aspektgraphen heraus zu editieren.

3.3 JQuery

JQuery ist ein Werkzeug zur anfragebasierten Suche im Quellcode. Es arbeitet auf Basis der logischen Anfragesprache *TyRuBa*. Der Quellcode wird nach den Kriterien einer Anfrage untersucht. Dabei kann es sich unter anderem auch um nicht funktionale Belange (siehe Kapitel 2.2), die über mehrere Module im Programmcode verteilt sind, handeln. Außerdem kann nach bereits implementierten Aspekten im Quellcode gesucht werden. Die Ergebnisse einer Anfrage werden automatische an den Quellcode gebunden. Dadurch ist die einfache Navigierbarkeit zwischen den Anfrage Ergebnissen und dem Quellcode möglich und der Quellcode kann gegebenenfalls direkt geändert werden. Außerdem ist das Speichern von Anfragen möglich. Die Ergebnisse einer Anfrage können nicht gespeichert werden. Das bedeutet, dass eine Anfrage erneut ausgeführt werden muss, um die Ergebnisse der Suche anzeigen zu können. Dadurch wird aber auch sichergestellt, dass immer der aktuellste Code zur Auswertung verwendet wird.

JQuery ist seit Oktober 2004 in der Version 3.1.3 als Plugin für *Eclipse* 3.0 verfügbar.

3.4 Aspect Mining Tool

Aspect Mining Tool analysiert die Beziehungen zwischen Aspekten im Quellcode. Die Ergebnisse der lexikalischen und typbasierten Analyse werden grafisch dargestellt. *Aspect Mining Tool* basiert auf dem *AspectJ* Compiler.

Aspect Mining Tool wurde an der University of British Columbia entwickelt. Es ist in der Version 0.6a vom Oktober 2001 verfügbar.

4 Techniken und Werkzeuge zum Entwurf Aspektorientierter Softwareprogramme

„Die Entwurfsphase in der Softwareentwicklung hat das Ziel für das zu entwerfende Softwaresystem eine Softwarearchitektur zu erstellen, die die funktionalen und nichtfunktionalen Produktanforderungen sowie allgemeine und produktspezifische Qualitätsanforderungen erfüllt und die Schnittstellen zur Umgebung versorgt.“ [1] Die Softwarearchitektur soll die Struktur des Softwaresystems durch Systemkomponenten und ihre Beziehungen untereinander beschreiben.

In den nachfolgenden Kapiteln werden einige Werkzeuge vorgestellt, die insbesondere das Entwerfen von nicht funktionalen Belangen an ein Softwaresystem in einer Softwarearchitektur ermöglichen.

4.1 Concern Space Modeling Schema

Concern Space Modeling Schema (kurz *COSMOS*), ist eine Technik zum Entwurf Aspektorientierter Softwareprogramme. Es ermöglicht die Modellierung von mehrdimensionalen Belangen an ein Softwaresystem. Außerdem wird der Entwurf von gegensätzlichen Aspekten unterstützt. *COSMOS* erweitert bekannte Ansätze zur Modellierung wie *UML* und *Entity-Relationship* Diagramme. Dabei ist *COSMOS* unabhängig von bestehenden Formalismen, Entwicklungsmethoden und Werkzeugen.

Zur Modellierung eines Softwaresystems definiert *COSMOS* einen Anforderungsraum (im Englischen *concern space*). Dieser Anforderungsraum enthält die Belange an ein Softwaresystem und deren Beziehungen untereinander. Die Belange werden klassifiziert in logische und physische Belange. Beide Typen von Belangen können unabhängig voneinander vorkommen.

Logische Belange beschreiben die Anforderungen die ein Softwaresystem erfüllen soll. Beispiele dafür sind Funktionalität, Verhalten, Performanz, Robustheit, Zustand, Bindung, Variabilität, Benutzbarkeit, Größe und Kosten. Logische Belange werden in fünf Typen unterteilt (siehe Abbildung 5):

Klassifizierungen fassen Klassen zu einer Menge zusammen. Sie sind Wurzeln von Hierarchiebäumen von Klassen. Ein logischer Belang kann zu mehreren Klassifizierungen gehören. Ein Beispiel hierfür ist ein Quellcode Segment, das einer Programmiersprache, einer Klasse und auch einer Operation zugeordnet werden kann. (im Englischen classifications)

Klassen kategorisieren Belange. Sie werden durch Klassifizierungen zusammengefasst und können Unterklassen und logische und physische Instanzen beinhalten. Beispielsweise kann ein Dateiobjekt den Klassen Einfügen, Löschen, Verschieben oder Erstellen zugeordnet werden. (im Englischen classes)

Instanzen können Klassen zugewiesen werden oder separat vorkommen. Ein Beispiel für eine Instanz ist das Exemplar KopiereDatei der Klasse Kopiere. (im Englischen instances)

Eigenschaften beschreiben die Eigenschaften der logischen Belange. Sie können Klassifizierungen, Klassen und Instanzen zugewiesen werden. Sie werden von Klassifizierungen zu Klassen, von Klassen zu Unterklassen und von Unterklassen zu Instanzen vererbt. Beispiele hierfür sind Performanz, Größe und Breite. (im Englischen properties)

Themen gruppieren logische Belange. Sie definieren beliebige Gruppen (im Englischen collections) über Klassifizierung und Klassen Grenzen hinweg. Die Gruppenelemente können von den Typen Klassifizierung, Klasse und Instanz sein. (im Englischen topics)

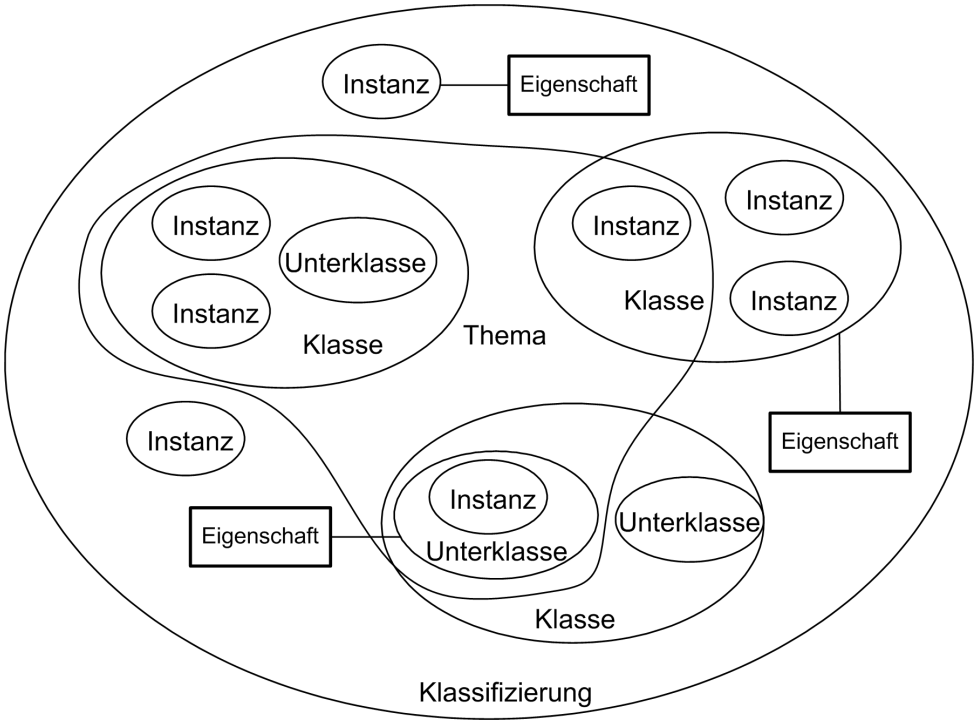


Abbildung 5: COSMOS: logische Belange

Physische Belange beschreiben die Schnittstellen zwischen Softwareprogrammen und den Umgebungen in denen sie ausgeführt werden. Sie beziehen sich auf die logischen Anforderungen. Beispiel für physische Belange sind Softwarekomponenten, Hardwarebauteile und Dienste. Physische Belange werden in drei Typen unterteilt (siehe Abbildung 6):

Instanzen sind Softwarekomponenten, Hardwarekomponenten, Dienste oder bestimmte Systeme. Beispielsweise ist eine Java-Datei eine Instanz. (im Englischen instances)

Sammlungen sind Pakete die Quellcode enthalten und Gruppierungen von Instanzen. Sie können Untersammlungen enthalten. (im Englischen collections)

Attribute beschreiben die Eigenschaften von Instanzen und Sammlungen. Sie sind konkrete Werte von konkreten Objekten. (im Englischen attributes)

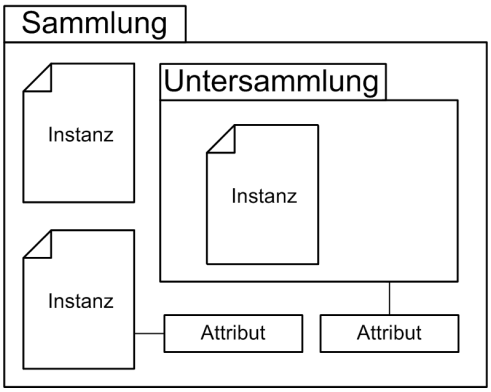


Abbildung 6: COSMOS: physische Belange

Zwischen den verschiedenen Typen von Belangen an ein Softwaresystem können Beziehungen bestehen. Diese können wie folgt klassifiziert werden:

Kategorische Beziehungen setzen die Belange der gleichen Kategorie zueinander in Beziehung. (im Englischen categorical relationships) (siehe Tabelle 1)

Interpretierbare Beziehungen existieren meist zwischen logischen Belangen. Es werden die semantische Bedeutung und die Kontextabhängigkeit zwischen Belangen betrachtet. (im Englischen interpretive relationships)
Beispiele für solche Beziehungen sind:

- hat teil an,
- bedingt oder
- wird implementiert durch.

Physische Beziehungen existieren zwischen physischen Belangen. Beispielsweise besteht eine Kompositionsbeziehung zwischen den Komponenten eines Systems und dem Gesamtsystem. (im Englischen physical relationships)

Abbildende Beziehungen setzen physische und logische Belange zueinander in Beziehung. Ein Beispiel hierfür ist die physische Implementation einer Funktion durch bestimmten Quellcode. (im Englischen mapping relationships)

Die Konstruktion eines Anforderungsraumes besteht aus den folgenden Schritten:

1. Identifizierung, Sammlung und Typisierung von Belangen,
2. Definition von Beziehungen zwischen den Belangen,
3. Definition von Attributen für Belange und deren Beziehungen,
4. Verfeinerung der Belange und ihrer Beziehungen über die Phasen der Softwareentwicklung.

Rolle	**Beziehung**	**Rolle**
Klassifizierung	Klassifizierung	Klasse
Klasse (Oberklasse)	Generalisierung	Klasse (Unterklasse)
Klasse	Instanzierung	logische Instanz, Eigenschaft, Sammlung, physische Instanz, Attribut, Klassifizierung, Klasse oder Instanz,
Eigenschaft	Charakterisierung	Klassifizierung,Klasse oder Instanz
Thema	Thematik	Anforderung
Sammlung	Zugehörigkeit	physische Instanz
Attribut	Attributierung	Sammlung oder physische Instanz

Tabelle 1: COSMOS: Kategorische Beziehungen

COSMOS ist nur ein Ansatz zur Modellierung von Aspekten eines Softwaresystems und wurde zwischen 1999 und 2001 von IBM entwickelt. Bisher gibt es keine direkte technische Unterstützung. Es ist allerdings möglich die Technik mit gängigen Entwurfswerkzeugen anzuwenden.

In der Abbildung 7 ist ein Beispiel für einen Anforderungsraum zu sehen.

4.2 Modellierung von Aspekten mit UML

Zur Modellierung von Aspekten mit *UML* gibt es die beiden Ansätze *UML*-Profile und *Theme/UML*. Beide Techniken erweitern die bestehende graphische Modellierungssprache *Unified Modeling Language* (kurz *UML*) für Objektorientierte Softwareentwicklung (im Englischen *Object-Oriented Software Development*, kurz *OOSD*).

UML-Profil ist der Standard Mechanismus zur Erweiterung des Sprachumfangs von *UML*. Ein *UML*-Profil definiert die Syntax und die Semantik der Modellierungssprache durch Einschränkungen. *UML*-Profile werden mit Hilfe von Stereotypen, Eigenschaftswerten (im Englischen tagged values) sowie Einschränkungen (im Englischen constraints) definiert. Mit Hilfe von *UML*-Profilen wird *UML* an spezifische Einsatzbedingungen, in diesem Falle *AOP*, angepasst. Das im folgenden beschriebene *UML*-Profil modelliert Aspekte mit bereits bestehenden *UML* Konstrukten.

Aspekte werden mit Hilfe des Stereotypen `<<aspect>>` gekennzeichnet. Überschneidende Belange (siehe Kapitel 2.2) werden ebenfalls durch Stereotypes mit `<<crosscut>>` gekennzeichnet. Falls ein Aspekt die Funktionalität einer Klasse beeinflusst, so wird er besonders gekennzeichnet mit `{synchronous}`. Die Kennzeichnung eines Aspektes mit `{synchronous}` erfordert bei der Umsetzung eines Aspektorientierten Softwaremodells die Implementierung von Vor- und Nachbedingungen in Form von Methoden in den betroffenen Aspekten.

```
PreActivation()
PostActivation()
```

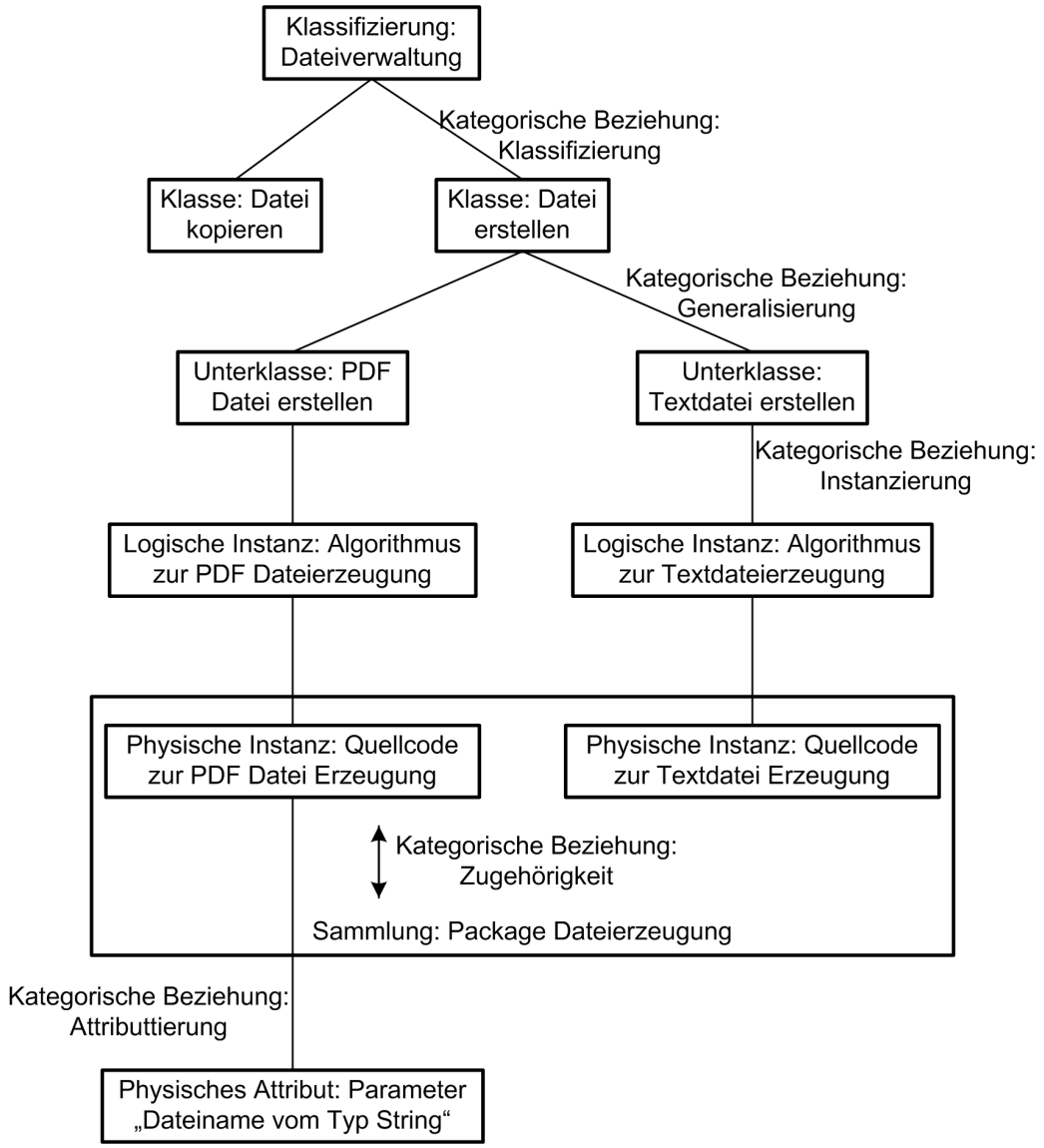


Abbildung 7: COSMOS: Beispiel zur Datei Verwaltung

Das *UML*-Profil kann ohne die Notwendigkeit zusätzlicher Werkzeuge zur Modellierung von Aspekten benutzt werden. Dies ist möglich, da das *UML*-Profil bereits bestehende *UML*-Konstrukte zur Modellierung von Aspekten benutzt.

Ein Beispiel für ein mit dem *UML*-Profil entworfenes Aspektorientiertes Softwaremodel zeigt Abbildung 8.

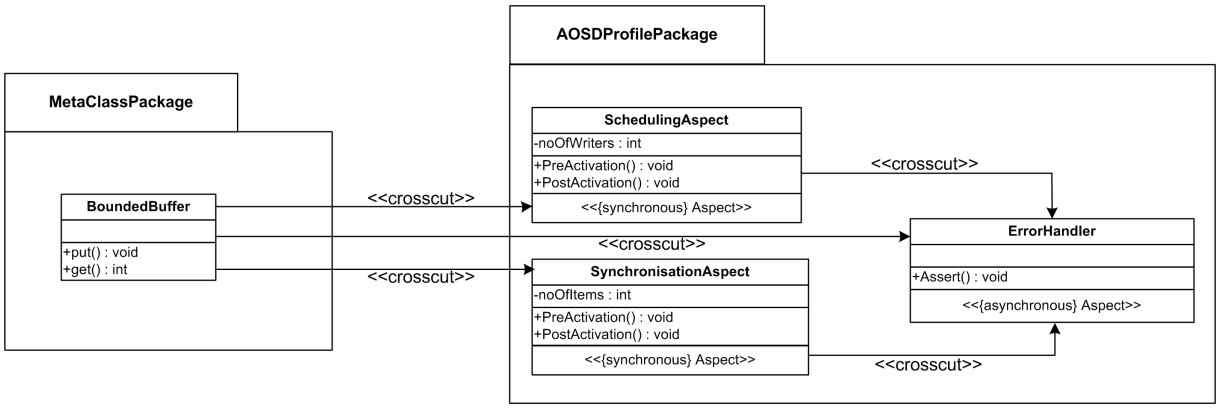


Abbildung 8: Beispiel UML Profil

Theme/UML basiert auf der *Subjektorientierten Programmierung*, welche im Kapitel 5.1.7 genauer beschrieben wird.

Theme/UML löst die verteilten Belange aus einem Gesamtsystem heraus und fügt sie mit Hilfe von *UML*-Schablonen (im Englischen templates) neu in das Gesamtsystem ein. *UML*-Schablonen sind parametriesierte Modellelemente. Die Aspekte werden in Paketen gekapselt und durch *UML*-Schablonen an den Paketen gekennzeichnet. An die *UML*-Schablonen können beliebige reale Modellelemente mit dem Stereotype *bind* gebunden werden.

Ebenso wie das *UML*-Profil kann auch *Theme/UML* ohne zusätzliche Werkzeuge zur Modellierung von Aspekten benutzt werden, da bereits bestehende *UML*-Konstrukte zur Modellierung der Aspekte benutzt werden.

Ein Beispiel für ein Aspektorientiertes Softwaremodel entworfen mit *Theme/UML* zeigt Abbildung 9.

4.3 CoCompose

CoCompose ist ein Werkzeug zum Design Aspektorientierter Softwareprogramme. Mit Hilfe von *CoCompose* können Modelle Aspektorientierter Softwareprogramme entworfen werden. Für jedes Designelement eines Modells können mehrere mögliche Verfeinerungen definiert werden. *CoCompose* ermöglicht die automatische Quellcode Generierung aus dem Entwurf heraus in die Programmiersprachen *Java* und *ConcernJ*.

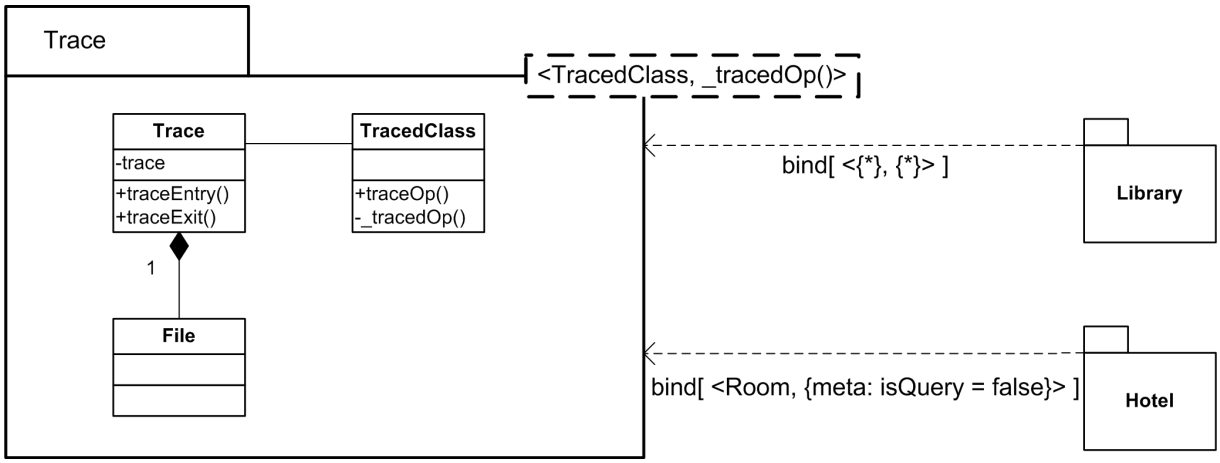


Abbildung 9: Beispiel Theme/UML

CoCompose2 ist eine neuere Version von *CoCompose*. Da sich *CoCompose2* noch in einem frühen Entwicklungsstadium befindet sind noch nicht alle geplanten Funktionen umgesetzt. Zur angestrebten Funktionalität von *CoCompose2* gehört unter anderem der Import und Export von *UML* Modellen, die Automatische Verfeinerung von *CoCompose* Modellen, die automatische Quellcode Generierung für *Java* und mindestens noch eine andere Programmiersprache.

CoCompose ist als eigenständiges Werkzeug in der Version 0.1.22 verfügbar. *CoCompose2* ist als *Eclipse* Plugin in der Version 0.2.14 verfügbar.

4.4 Codagen Architect

Codagen Architect ist ein kommerzielles Werkzeug zum Design Aspektorientierter Softwareprogramme. Es unterstützt den Ansatz *Model Driven Architecture* (kurz *MDA*). Außerdem bietet es die Möglichkeit der automatischen Quellcode Generierung für die Zielplattformen *J2EE*, *.NET* und die Sprachen *Java*, *C++*, *C#*, *Visual Basic* aus dem Entwurf heraus. Die definierten Aspekte werden bei der Quellcode Generierung verwoben.

Codagen Architect besitzt kein eigenes integriertes *UML* Werkzeug, sondern wird als Add-In für bestehende *UML* Werkzeuge ausgeliefert. Unterstützt werden unter anderem *Rational Rose*, *Rational XDE*, *MagicDraw*, *Borland Together Control Center* und *Microsoft Visio*.

5 Techniken und Werkzeuge zur Implementierung Aspektorientierter Softwareprogramme

In der Implementierungsphase der Softwareentwicklung wird die in der Entwurfsphase erstellte Softwarearchitektur umgesetzt. Dabei werden die in der Softwarearchitektur spezifizierten Softwarekomponenten implementiert.

Im Folgenden werden einige Werkzeuge vorgestellt, welche die Implementierung von nicht-funktionalen Eigenschaften und die Verwebung von Aspektcode mit funktionalem Programmcode ermöglichen.

5.1 Statische Aspektweber

Statische Aspektweber verweben Aspektcode mit funktionalem Programmcode vor bzw. nach der Compilierung des Programmcodes, bevor das Programm zur Ausführung kommt. (siehe 2.3)

Im den nachfolgenden Kapiteln werden einige statische Aspektweber vorgestellt.

5.1.1 AspectJ

AspectJ ist eine Erweiterung der Programmiersprache *Java* um Konstrukte zur *Aspektorientierten Programmierung*. Aspekte werden durch das Schlüsselwort *aspect* definiert. Sie definieren die Verwebungspunkte (siehe Kapitel 2.2.1) und implementieren den Aspektcode.

Ein *pointcut* in *AspectJ* spezifiziert eine Menge von Verwebungspunkten und wählt aus dem Programmcode die passenden aus. Mögliche Verwebungspunkte in *AspectJ* sind im Folgenden aufgelistet, wobei in Klammern die jeweiligen Schlüsselwörtern für *AspectJ* aufgeführt sind:

- Aufruf und Ausführung von Methoden (*call*, *execution*)
- Zugriff auf Felder (*set*, *get*)
- Instanzierung, vor der Instanzierung, Aufruf und Ausführung von Objekten (*initialization*, *preinitialization*, *call*, *execution*)
- statische Instanzierung von Klassen (*staticinitialization*)
- Ausnahmebehandlungen (*handler*)
- Ausführung von Aspektcode (*adviceexecution*)
- Codeausführung innerhalb einer Klasse, eines Konstruktors oder einer Methode(*within*, *withincode*)
- Typüberprüfung des aktuellen Objektes, eines gerufenen Objektes oder der Argumente (*this* , *target* , *args*)

- Kontrollfluss eines Verwebungspunktes oder der Kontrollfluss nach dem Verwebungspunkt(*cflow* , *cflowbelow*)

Zur Spezifikation mehrerer Verwebungspunkte in einem *pointcut* können Platzhalter verwendet werden. Als Platzhalter dienen .. und *. Der Platzhalter .. steht für eine Zeichenkette, die mit einem Punkt beginnt und mit einem Punkt endet oder für eine beliebige Anzahl von Parametern. * steht für 0 oder n Zeichen, außer dem Punkt. Außerdem können Verwebungspunkte durch die logischen Operatoren und (&&), oder (||), nicht (!) verknüpft werden. Auch können für *pointcuts* Bedingungen (if(...)) definiert werden. Zusätzlich können *pointcut* Parameter benutzt werden. Diese müssen in Bezug zu den vom *pointcut* spezifizierten Verwebungspunkten stehen. In einem *advice* können die *pointcut* Parameter benutzt werden. Ein Beispiel für einen Pointcut ist:

```
pointcut add():call(public * add(..));
```

In diesem Beispiel wird ein Pointcut mit dem Namen *add* definiert. Dieser Pointcut spezifiziert einen Verwebungspunkt für den Aufruf der öffentlich deklarierten Methode *add* , die einen beliebigen Rückgabewert und eine beliebige Anzahl und Typen von Argumenten besitzen darf.

Ein *advice* definiert den nicht-funktionalen Programmcode, der an einem Verwebungspunkt eingewoben werden soll und beim Erreichen eines Verwebungspunktes während eines Programmdurchlaufs ausgeführt werden soll. Ein *advice* kann vor einem Verwebungspunkt (*before*), nach einem Verwebungspunkt (*after* , *after returning* , *after throwing*) oder anstatt eines Verwebungspunktes (*around*) ausgeführt werden. Für den Fall, dass ein *advice* anstatt eines Verwebungspunktes ausgeführt werden soll besteht die Möglichkeit über Kontextinformationen innerhalb des *advice* auf Informationen über den Verwebungspunkt zuzugreifen. Dazu stehen die drei Variablen *thisJoinPoint* ,*thisJoinPointStaticPart* und *thisEnclosingJoinPointStaticPart* zur Verfügung. *thisJoinPoint* ist ein Objekt, das den aktuellen Verwebungspunkt repräsentiert. Mit *thisJoinPointStaticPart* hat man Zugriff auf statische Kontextinformationen des Verwebungspunktes. Mit *thisEnclosingJoinPointStaticPart* hat man Zugriff auf statische Kontextinformationen der rufenden Methode. Außerdem kann innerhalb eines *around advice* die Ausführung eines Verwebungspunktes explizit erzwungen werden mit Hilfe der Methode *proceed* . Ein Beispiel für ein *advice* ist:

```
before():add(){
  System.out.println("Before: " + thisJoinPoint.getSignature());
}
```

Das *advice before* definiert den Programmcode, der vor der Ausführung des zugehörigen Verwebungspunktes, der durch den *pointcut add* spezifiziert wurde, ausgeführt werden soll. In diesem Beispiel wird die Signatur des Verwebungspunktes ausgegeben.

Mit Hilfe von *inter-type declarations* kann der funktionale Programmcode statisch modifiziert werden. Es ist möglich Programmcode in Form von Methoden, Konstruktoren, Feldern, Klassen oder Interfaces und Vererbungsbeziehungen hinzuzufügen. Außerdem können Warnungen, Fehler und Ausnahmen beim Erreichen von *pointcuts* ausgegeben werden und ein bevorzugt bzw. als erstes auszuführender *advice* ange-

geben werden, wenn zu einem *pointcut* mehrere *advices* definiert sind. Ein Beispiel für das Hinzufügen einer Vererbungsbeziehung ist das Folgende:

```
declare parents: Calculator extends Multiplier;
```

In diesem Beispiel wird definiert, dass die Klasse *Calculator* von der Klasse *Multiplier* erbt.

AspectJ verwebt einen *aspect* statisch an den definierten Verwebungspunkten mit dem funktionalen Programmcode. Nach der Verwebung ist der Aspektcode als zusätzlich definierte Methoden im Programmcode zu finden.

AspectJ wurde ab 1996 von einer Gruppe um Georg J. Kiczales am Xerox Palo Alto Research Center entwickelt. Seit Ende 2002 ist *AspectJ* Teil des *Eclipse* Projektes.

Ende 2002 wurde die *AspectJ* Version 1.0 unter dem *Eclipse* Projekt veröffentlicht. *AspectJ* 1.0 enthält als Kommandozeilen Werkzeuge eine Compiler (*ajc*), einen Debugger (*ajdb*), einen Dokumentations-Generator (*ajdoc*), als grafisches Werkzeug den *AspectJ Browser* und unterstützt *Ant*, zur automatischen Erstellung von Programmen. *AspectJ* 1.0 ist ein Quellcode Weber. Die Verwebung einer *Java* Klasse mit einem Aspekt erfolgt statisch auf Quellcode-Basis mit Hilfe des Kommandozeilen Compilers *ajc* . Mit Hilfe des *AspectJ Browser* kann man die verteilten Belange in einem Softwareprogramm identifizieren, die Beziehungen zwischen Verwebungspunkten und Aspektcode besser überblicken und die Verwebung von *Java* Klassen mit Aspektcode grafisch durchführen und visualisieren.

2003 wurde die *AspectJ* Version 1.1 veröffentlicht. Ab der Version 1.1 ist *AspectJ* eine Bytecode Weber. Der Compiler *ajc* setzt auf dem Open-Source *Eclipse* Compiler auf. *ajc* compiliert zuerst die *Java* Klassen und die Aspekte und verwebt diese dann auf Bytecode Basis.

2004 wurde die *AspectJ* Version 1.2 veröffentlicht.

Ende 2005 wurde die aktuellste *AspectJ* Version 5 veröffentlicht. *AspectJ 5* unterstützt die mit *Java 5* eingeführten Features, wie z.B. *Annotations*, *Generics*, *Autoboxing* und *Unboxing*. *Annotations* können in *AspectJ 5* dazu benutzt werden Aspekte und innerhalb von Aspekten Methoden, Felder, Konstruktoren, *advices*, *inter-type declarations*, Methoden- und *advice*-Parameter mit Metadaten zu kennzeichnen. *Annotations* werden im Quellcode mit dem Zeichen @ markiert. Sie können zur Laufzeit explizit mittels Reflection abgefragt werden. Außerdem können sie von Aspekten mit `@Inherited` an abgeleitete Aspekte vererbt werden. *Annotations* können in *pointcuts* als zusätzliche Bedingungen für passende Verwebungspunkte mit angegeben werden. Des Weiteren unterstützt *AspectJ 5* generische und parametrisierte Typen in *pointcuts*, *inter-type declarations* und bei der Vererbung. Außerdem unterstützt *AspectJ 5* die automatische Konvertierung von Primitiven Datentypen in ihre äquivalenten Objekt Datentypen (*Autoboxing*) und umgekehrt (*Unboxing*) in *pointcuts*.

Parallel zum *AspectJ* Projekt läuft unter dem *Eclipse* Projekt auch noch das *AspectJ Development Tools* (kurz *AJDT*) Projekt. *AJDT* entwickelt *Eclipse* Plugins für *AspectJ*. Durch die Integration von *AspectJ* als Plugin in die Entwicklungsumgebung *Eclipse* sind die folgenden Funktionen nutzbar:

- Spezieller Editor für *AspectJ*,

- Vervollständigung von Quellcode,
- Grafisches Debuggen,
- Konfiguration von Aspekten mit Hilfe eines Wizards,
- Verwaltung und Bearbeitung der Build-Dateien,
- Anzeige der Verwebungspunkte im Quellcode,
- Direkte Erstellung von Jar-Archiven und
- Generierung von *AspectJ* Dokumentation (*ajdoc*).

Außerdem gibt es *AspectJ* Unterstützung für die Entwicklungsumgebungen *Emacs*, *JBuilder* und *Netbeans*.

Das folgende Beispiel zeigt die Verwebung eines einfachen Tracing Aspektes. Die Klasse *Calculator* ist die zu verwebene Zielklasse. *Calculator* implementiert eine Methode zur Addition.

```
public class Calculator{
  public int add(int a, int b){
    return a+b;
  }
}
```

Der Aspekt *Trace* definiert einen *pointcut* für den Aufruf aller Methoden mit Namen `add`, mit beliebigen Argumenten und beliebigem Rückgabewert. Außerdem definiert *Trace* zwei *advices* , von denen einer vor und einer nach dem Eintritt des *pointcut* ausgeführt werden soll. Beide *advices* geben die Signatur des Verwebungspunktes aus.

```
public aspect Trace{
  pointcut add():call(public * add(..));

  before():add(){
    System.out.println("Before: " + thisJoinPoint.getSignature());
  }

  after():add(){
    System.out.println("After: " + thisJoinPoint.getSignature());
  }
}
```

CalculatorApplication ist eine Konsolenanwendung, welche die Klasse *Calculator* instanziert und die Methode `add` aufruft.

```
public class CalculatorApplication{
  public static void main(String[] args){
    Calculator calc = new Calculator();
    System.out.println("Result of 2 + 3 = " + calc.add(2,3));
  }
}
```

Wird die Klasse *Calculator* mit dem Aspekt *Trace* mit Hilfe von *AspectJ* verwoben und danach die Konsolenanwendung *CalculatorApplication* gestartet, erfolgt die folgende Konsolenausgabe:

```
Before: int Calculator.add(int, int)
After: int Calculator.add(int, int)
Result of 2 + 3 = 5
```

Das Beispiel wird nun um die Klasse *Multiplier* erweitert. *Multiplier* implementiert eine Methode zur Multiplikation.

Des Weiteren wird der Aspekt *Trace* durch den Aspekt *Inheritance* ersetzt. *Inheritance* erweitert die Vererbungsbeziehungen der Klasse *Calculator* . Nach der Verwebung von *Calculator* mit *Inheritance* erbt *Calculator* von *Multiplier*. Der Aspekt *Inheritance* definiert einen *pointcut* für den Aufruf aller Methoden mit Namen `add`, mit beliebigen Argumenten und beliebigem Rückgabewert. Außerdem definiert *Inheritance* ein *arround advice* , das auf einem Objekt der Klasse *Calculator* die geerbte Methode `mult` aufruft und danach explizit die Ausführung des Verwebungspunktes erzwingt.

```
public class Multiplier {
  public int mult(int a, int b){
    return a*b;
  }
}
public aspect Inheritance {
  declare parents: Calculator extends Multiplier;

  pointcut add():call(public * add(..));

  Object around():add(){
    Calculator calc = (Calculator)thisJoinPoint.getTarget();
    System.out.println("Result of 2 * 3 = " + calc.mult(2,3));
    Object o = proceed();
    return o;
  }
}
```

Wird nun die Klasse *Calculator* mit dem Aspekt *Inheritance* mit Hilfe von *AspectJ* verwoben und danach die Konsolenanwendung *CalculatorApplication* gestartet, erfolgt die folgende Konsolenausgabe:

```
Result of 2 * 3 = 6
Result of 2 + 3 = 5
```

AspectJ war die erste Aspektorientierte Programmiersprache, besitzt große Akzeptanz und eine weite Verbreitung. *AspectJ* wird ständig weiterentwickelt und ist gut in gängige Entwicklungsumgebungen integrierbar. Allerdings werden nicht alle Features der Entwicklungsumgebungen unterstützt. Auch einige *Java* Fehler werden vom Compiler nicht korrekt weitergereicht.

5.1.2 AspectBench Compiler

Der *AspectBench Compiler* (kurz *abc*) ist neben dem *AspectJ Compiler* (kurz *ajc*) eine weitere Implementierung der *Java* Spracherweiterung *AspectJ*. *abc* existiert als Referenzcompiler zu *ajc* und wird als ein erweitertes Framework zur Implementierung von *AspectJ* Erweiterungen und deren Optimierungen benutzt was im Folgenden aufgezeigt wird. Dazu wird zunächst die grundlegende Architektur des Compilers näher erläutert. Danach wird speziell auf schon implementierte Standard Erweiterungen Bezug genommen und deren Implementierung kurz beschrieben.

Beim *AspectBench Compiler* handelt es sich um ein Joint-Projekt, welches aus der Programming Tools Group der Oxford Universität in den USA, der Sable Research Group der McGill Universität in Kanada und später der BRICS (Basic Research in Computer Science) der Universität von Aarhus in Dänemark hervorgegangen ist. Es handelt sich bei *abc* um die komplette Implementierung der Sprachdefinition *AspectJ*, welche sowohl das Ziel der Erweiterung als auch der Optimierung der Kernsprache verfolgt. Dabei baut es auf den beiden eigenständigen Frameworks *Polyglot* und *Soot* auf.

Im Bezug zu *ajc*, ist *abc* in erster Linie als Neuimplementierung der Sprache *AspectJ* zu sehen. Es bringt einige kleine Unterschiede mit sich, obwohl *abc* mit Hilfe von *ajc* und der gleichen Testsuite als Referenz entwickelt wurde. Auch ist sogar die komplette Laufzeitbibliothek aus *ajc* entnommen.

Unterschiede zwischen *abc* und *ajc* kann man nur durch Betrachtung zweier nebenläufig entwickelter Versionen vornehmen. Dabei sind die *abc* Version 1.0.1 und *ajc* in der Version 1.2.1 unterscheidbar und gewisse Abweichungen aufführbar.

Zum einen sei als ein Unterschied die Nichtunterstützung des separaten Compilieren von Aspekten zu nennen. Das meint, es können durch *abc* nur Dateien mit den Endungen .java oder .aj also Java Quellcode, compiliert werden. Hingegen ist es möglich mit *ajc* auch *Java* Bytecode zu verweben.

Als einen zweiten wesentlichen Unterschied zu *ajc* benötigt *abc* wesentlich mehr Zeit zur Compilierung, es ist in dieser Version ungefähr 4-mal so langsam wie *ajc*. Doch liegen die Stärken von *abc* in der einfachen Implementierung von Erweiterungen und dem aggressiven Verhalten bei Optimierungsverfahren der Bytecodeerzeugung. Somit stellt sich die Frage, ob mit *abc* compilierte *AspectJ* Programme durch derzeitige Optimierungsverfahren wirklich schneller laufen. Als Antwort sei genannt, dass es davon abhängt, welche Sprachfeatures benutzt werden, so erzeugt *abc* derzeit für z.B. `cflow()` einen 45-mal schnelleren Code als *ajc*. Dabei unterstützt *abc* kein *Load-Time*

Weaving und geht auch bei den durch *AspectJ* neu eingeführten Schlüsselwörtern einen anderen Ansatz. So folgt *abc* dem Design von *Java* klar zwischen „Identifiers“ und „Keywords“ zu unterscheiden. Dabei hängt es vom jeweiligen Kontext ab:

***Java* Klasse** normale Schlüsselwörter, plus *aspect, privileged* und *pointcut*

Aspekt alle Schlüsselwörter aus *Java*, plus *after, around, before, declare, issingleton, percflow, percflowbelow, pertarget, perthis, proceed*

Pointcut nur *adviceexecution, args, call, cflow, cflowbelow, error, execution, get, handler, if, initialisation, parents, precedence, preinitialization, returning, set, soft, staticinitialization, target, this, throwing, warning, within, withincode*

abc bietet des Weiteren zusätzliche Features, wie die Unterstützung von geschachtelten Kommentaren (/* ... /* ... */ ... */). Dazu wird als Übergabeparameter des Compilers

```
-nested-comments(:on)|(:true)
```

benötigt. Auch erlaubt es alternative Argumentbindungen in *Pointcuts*. So ist es z.B. möglich

```
( args(x, ..) || args(.., x) ) && if(x==0)
```

zu schreiben. Dies wird derzeit von *ajc* noch nicht unterstützt.

Abbildung 10 zeigt das Zusammenspiel von Aspekten und Komponenten unter Bezug der wesentlichen Begriffe. So können Aspekte in eigenen aj-Klassen definiert werden, müssen es aber nicht. Ein *Pointcut* ist immer ein Pattern von Events, bzw. ein Set aus Knoten in einem Aufrufgraphen, an welchem sich *Advices* anhängen lassen, die als Anweisungen extra Code enthalten. Dieser wird durch Matchen des *Pointcuts* auf einen *Join-Point*, also einen Event Knoten in einem dynamischen Aufrufgraphen, in einen so genannten *Shadow*, ein Programmpunkt der einem *Join-Point* entspricht, eingewoben.

Im Folgenden wird die Darstellung und die Funktionsweise von *abc* näher betrachtet.

Jeder Compiler verspricht einen geregelten Aufbau, der der Abbildung 11 folgt. So kann ein Compiler in ein Frontend und ein Backend unterteilt werden. Das Frontend übernimmt die gesamten Schritte der Analyse, darunter fallen die lexikalische, syntaktische und semantische Analyse. Das Backend übernimmt dann das Zusammenbauen in die Zielsprache. So wird aus dem Zwischencode durch Codeoptimierungen mit anschließender Codeerzeugung der Zielcode generiert.

In *abc* übernimmt *Polyglot* die Aufgabe des Frontends des Compilers, mit welchem es einfach möglich ist Erweiterungen (im Englischen: Extensions) zur Grundsprache zu implementieren. Der Ablauf der Analyse gestaltet sich wie folgt. Als erstes wird der Quellcode, gegeben in *Java* Sourcecodedateien, zu abstrakten Syntaxbäumen (im Englischen: abstract syntax tree, kurz AST) geparst. Danach werden alle statischen Checks, welche für die Sprache *Java* benötigt werden, durchgeführt. *Polyglot* führt also „compile-time Checks“ durch, um sicher zu stellen, dass sich keine Fehler mehr im Code befinden. Somit ist es mit der Ausgabe, die *Polyglot* als eigenständiges Framework

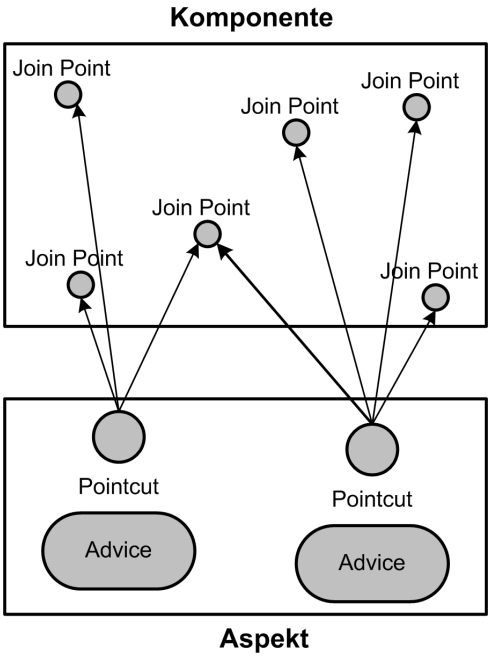


Abbildung 10: abc: Begriffsklärung

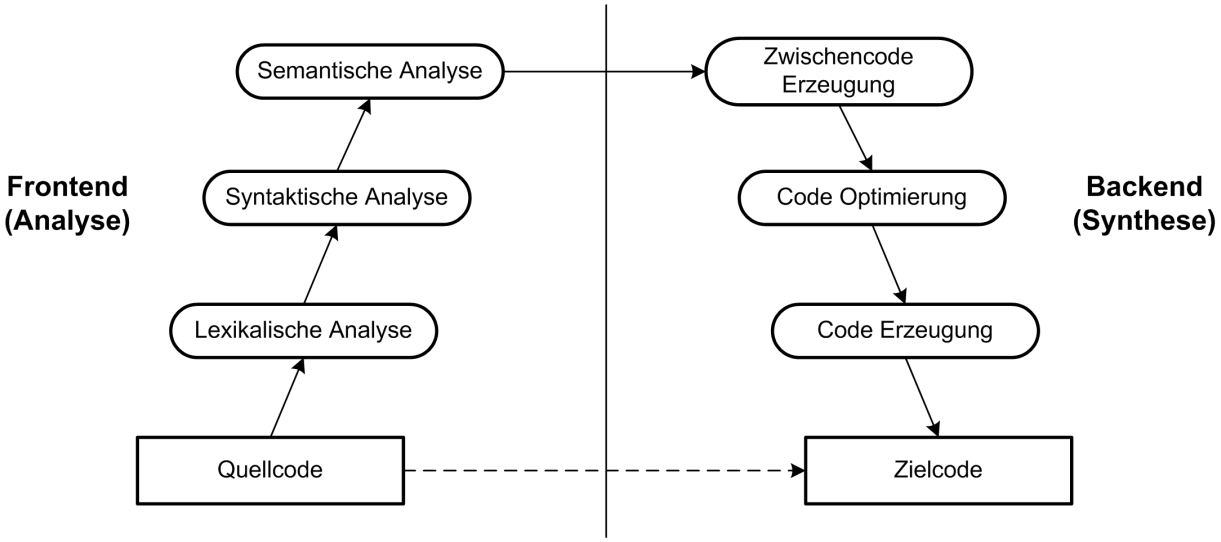


Abbildung 11: abc: Compiler allgemein

geben würde, möglich diese durch jeden Standard – *Java* Compiler ohne Fehler zu compilieren und auszuführen. Da *Polyglot* in *abc* aber die benötigte Compilervorarbeit leistet, welche später dann in *Soot* gebraucht wird, wird dieser AST nicht zurück in eine Datei geschrieben, sondern stattdessen noch ein zweites Mal von *Polyglot* bearbeitet. Und zwar liefert *Polyglot* für *Soot*, dem Backend des *abc* Compilers, eine Separation des AST in einen reinen *Java* Quellcode AST und Instruktionen für den Aspektweber, welcher später mit diesen Informationen die Verwebung der Aspekte vornehmen kann.

Soot gilt auch als ein eigenständiges Analyse- und Transformierungs-Framework für *Java*-Bytecode. Es ist in *abc* als Backendsystem für die Codegenerierung und die Aspektverwebung zuständig. *Soot* kann als Input sowohl mit *Java*-Bytecode als auch mit *Java*-Sourcecode umgehen. In *abc* wird aber nur der Umgang mit dem *Java*-Quellcode AST, welcher durch *Polyglot* übergeben wird, vorausgesetzt. Der wohl wichtigste Vorteil, den *Soot* bietet, ist *Jimple*, *Soots* „Intermediate Representation“ des Codes. *Jimple* ist ein typisierter, stackloser, „drei-adressen - Code“, welcher explizite Manipulation spezifizierter, lokaler Variablen ermöglicht. Dies geschieht, nicht, wie sonst, über einen Stack. Ein weiter Vorteil, den *Soot* bietet, sind seine diversen Module, mit denen es möglich ist zwischen *Jimple*, *Java*-Bytecode und *Java*-Sourcecode zu konvertieren. Damit ist es mit dem in *Soot* eingebauten Decompiler *Dava* auch möglich den Bytecode wieder zurück in Sourcecode zu transformieren. Dies ermöglicht jedem nachzuvollziehen, in wie fern Aspekte verwoben wurden, wie sich ein Programm mit Aspekten verhält und welche Effekte dabei auftreten. Weiterhin beinhaltet *Soot* schon Implementierungen für Standard Compiler Optimierungen, welche nach dem Weben in *abc* angewendet werden können. Auch bietet es zusätzlich Tools zum Schreiben von Erweiterungen an:

- Control Flow Graph Builder,
- Definition / Use-Chains,
- Ein Fixed-Point Flow Analyse Framework und
- Einen Method Inliner.

Wie im Abschnitt über *Polyglot* und *Soot* schon beschrieben, zeigt Abbildung 12 die Compilerarchitektur von *abc*. Dabei ist der Seperator die Schlüsselkomponente dieser Architektur. Diese teilt den *AspectJ* AST, welcher die Typinformationen beinhaltet, in den oben genannten reinen *Java* AST und die `AspectInfo` Struktur, welche die aspektspezifischen Informationen für das Weben bereithält. Die AspectInfo Struktur beinhaltet alle für das Backend benötigten Informationen zum Verweben der Aspekte. Die Hauptkomponenten dieser `AspectInfo` Struktur sind u.a.:

- *AspectJ* spezifische Sprachkonstruktionen,
- die interne Repräsentation der Klassenhierarchie und interne Klassenbeziehungen,
- eine Liste webbarer Klassen sowie
- weitere Komponenten.

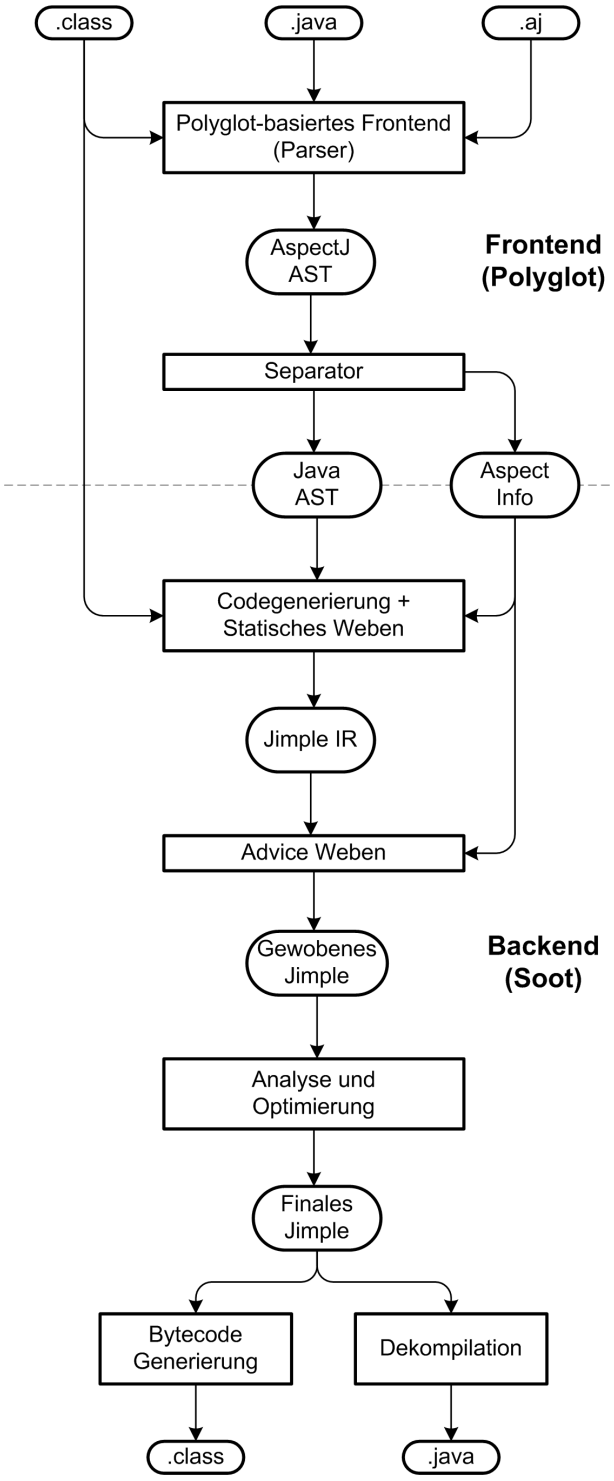


Abbildung 12: abc Architektur

Das Weben geschieht bei *abc* in zwei Stufen. Zum einen wird durch statisches Weben, in der Codegenerierungsstufe, aus dem reinen *Java* AST die so genannte „Jimple Intermediate Representation“ generiert. Unter der Verwendung der `AspectInfo` werden innerhalb dieser Codegenerierung die später beschriebenen *Shadows* eingewoben. In der zweiten Stufe vollzieht *abc* das so genannte *Advice* Weben, bei welchem die separat gehaltenen Aspekte, wiederum durch die `AspectInfos`, in den Code eingewoben werden. Schließlich ist aus der Architektur zu erkennen, dass es mittels *Soot* möglich ist aus der *Jimple IR* in *Java*-Quellcode bzw. in *Java*-Bytecode zurück schließen zu können.

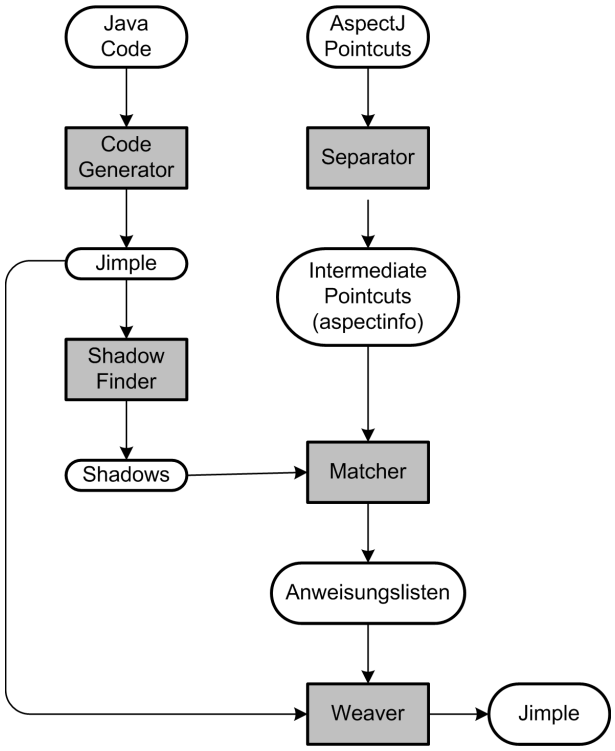


Abbildung 13: abc: Verweben von Pointcuts

Das Verweben von Pointcuts zeigt Abbildung 13. Diese Abbildung zeigt die Verarbeitungsstufen der *abc* - Architektur, wie aus dem *Java*-Quellcode und den Aspekt-Dateien die verwobenen *Jimple*-Dateien entstehen. Auf der einen Seite wird aus dem *Java* Code durch einen Code Generator die *Jimple*-Repräsentation erstellt. In dieser werden die *Shadows* bekannt gemacht und an den Matcher weitergereicht. Bei den *Shadows* handelt es sich um Rahmen, welche um die möglichen *Join-Points* gelegt werden, da bis dato nicht bekannt ist, an welcher Stelle die Anweisungen eines Aspekts auszuführen sind. Durch die *AspectJ Pointcuts* auf der anderen Seite erhalten wir mittels eines Separators die `AspektInfos`. Sind nun beide Informationen dem Matcher übergeben, so ist es diesem möglich die Aspekte in den *Shadows* zu identifizieren und Anweisungslisten für den Aspektweber, welche spezifische Informationen über das Einweben der Aspekte beinhalten, an den Weber weiterzureichen. Dieser webt nun die Anweisungen mittels der zusätzlich gegebenen Informationen über die Position in das *Jimple*-File ein.

Ein besseres Verständnis über das Einweben eines Aspekts in einen *Shadow* gibt die Abbildung 14. Diese veranschaulicht das Problem der Entscheidung über die Positionierung der Anweisungen eines Aspekts. Aus dem Quellcode im linken oberen Teil wird im ersten Schritt die Generierung in das *Jimple*-File vollzogen, welches sich rechts daneben befindet. In diesem speziellen Fall geht es um die Methode `eval()`, um welche hier das *Shadow*, mittels nop (no Operation), gelegt wurde. Durch die nop - Operationen wird der Rahmen um den *Join-Point* spezifiziert. Trifft es nun wie in diesem Beispiel zu, dass ein *Pointcut* existiert, welcher auf genau das `eval()` matcht welches sich in der Klasse `Main` befindet, so ist an dieser Stelle nur noch zu prüfen, wo genau die zusätzlich auszuführenden Anweisungen zu platzieren sind. Auch diese Informationen finden sich im Aspekt wieder. Zum einen existiert ein Schlüsselwort `before()`, wie in diesem Beispiel, durch welches bekannt wird, dass die an den Aspekt geknüpften Anweisungen vor Eintritt der gematchten Methode auszuführen sind, zum anderen existiert ein weiteres Schlüsselwort `after()`, durch welches die gefolgten Anweisungen nach Eintritt auszuführen sind und schließlich existiert ein `around`-Schlüsselwort. Die *around*-Variante ist die wohl Komplizierteste, da durch einen *around-Advice* entschieden werden kann, ob die folgenden Anweisungen bei Ausführung des Orginalcodes auszuführen sind, oder ob der Orginalcode zu ersetzen ist.

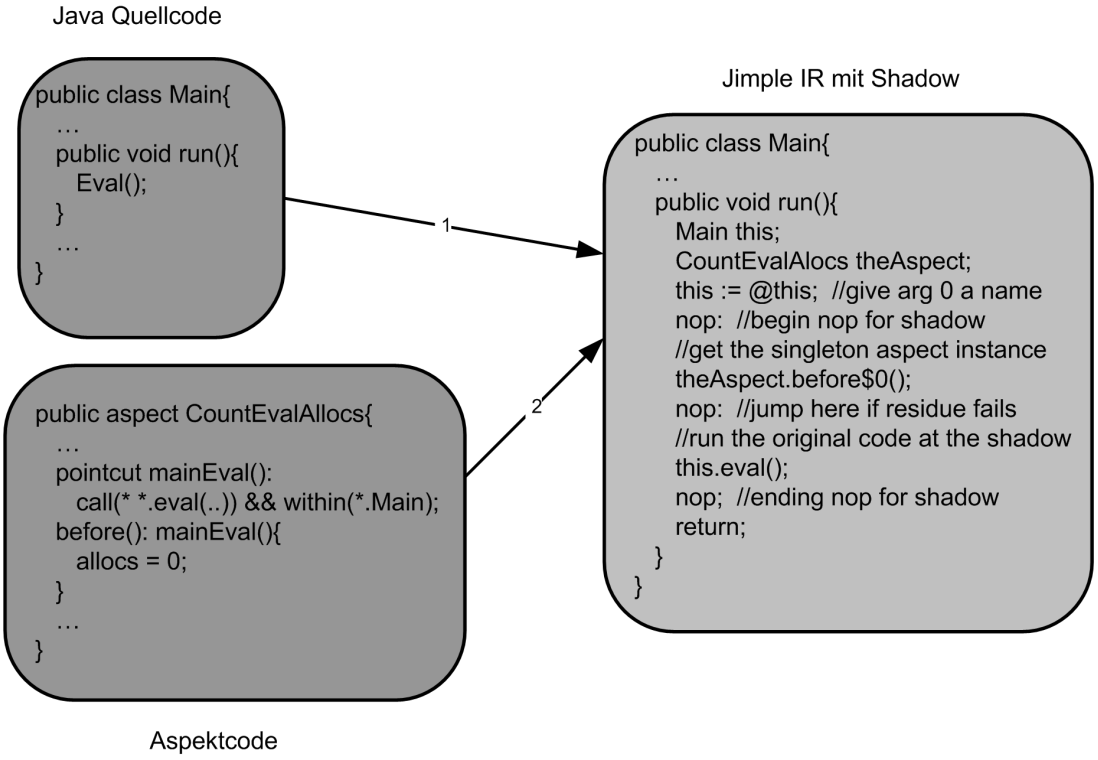


Abbildung 14: abc: Verweben von Quellcode

Der Ansatz zum Aufbau und Verweben eines *Shadows* gestaltet sich wie folgt:

- nop - Anweisungen am Anfang und Ende jedes *Shadows*

- einweben der *Advices* in einer „Inside-out Order“ (somit werden *before-Advices* als erstes eingewoben)

- durch nop wird ein eingefügter Bereich jeweils abgeschlossen

Alle *Advices*, sei es `before()`, `after()`, oder `around()`, besitzen jeweils einen eigenen Weber, welcher den Code an die richtige Stelle des *Shadows* einsetzt.

Im Folgenden werden die derzeit vorhandenen *AspectJ* Erweiterungen und deren mögliche Nutzung beschrieben, welche in der aktuellen *abc* Version 1.0.1 integriert sind. Sie dienen der Anschauung und bieten einen guten Einstieg in die Implementierung eigener Erweiterungen, bzw. der Erforschung von Optimierungsmöglichkeiten der Sprache. Um eigene Projekte mit diesen schon implementierten Erweiterungen zu compilieren, ist es von Nöten dem *abc* Compiler auf der Kommandozeile das Flag `-ext abc.eaj.ExtensionInfo [options and source files]` für die Version 0.9.0 oder 0.9.1, bzw. `-ext abc.eaj [options and source files]` für spätere Versionen zu übergeben.

Der einzige Weg neue Variablen in *Pointcuts* einzuführen, ist sie als explizite Parameter einer benannten *Pointcut* Definition zu übergeben. Beim Schreiben von *Pointcuts* und speziell benannten *Pointcuts* will man eine *Pointcut* Variable binden und dazu nicht den *Pointcut* damit parametrisieren. Da es wie eben beschrieben unnütz ist einen *Pointcut* zu parametrisieren, da man mit der Referenz auch gar nichts anfangen kann, denn man weiß das diese null sein muss, führt man `private` als Bezeichner für eine lokale *Pointcut* Variable ein. Ein Beispiel eines *Pointcuts*, welcher mit einer privaten Integer Variable bzw. einer Double Variable gebunden ist, und auf einen *Join-Point* matchen würde, bei dem das übergebene Objekt ein negatives Vorzeichen besitzen würde ist das Folgende:

```
pointcut negativeFirstArg():
  private(int x) ((args(x) && if(x<0)) ||
  private(double x) (args(x) && if(x<0));
```

Durch *Throw Pointcuts* ist es möglich Erweiterungen der Debug-Informationen vorzunehmen. *Throw Pointcuts* wurden in *ajc* eingeführt und als Referenz nach *abc* übernommen. Das folgende Beispiel zeigt die einfache Erweiterung der Debug-Informationen durch hinzufügen von Zustandsinformationen:

```
before(Debuggable d):this(d) && throw() && args(RuntimeException){
  d.dumpState();
}
```

Eine globale *Pointcut* Deklaration erlaubt es einen *Pointcut* mit den *Pointcuts* für alle Anweisungsdeklarationen, welche innerhalb von Aspekten auf ein *Classname-Pattern* matchen, zu verbinden. Das meint unter anderem auch, dass es möglich ist die Implementation einer Klasse bzw. eines Aspektes gegenüber Aspekten abzuschirmen. Dies veranschaulicht das folgende Beispiel:

```
aspect HiddenImplementation{
  global: * : !within(HiddenImplemenation);
  ...
}
```

Da wie oben genannt Aspekte auch in sich verwebt werden können, muss es also mittels globaler *Pointcut* Deklarationen möglich sein dies zu unterbinden. Das folgende Beispiel veranschaulicht dies:

```
aspect DoesNotApplyToSelf{
  global: DoesNotApplyToSelf: !within(DoesNotApplyToSelf);

  //the following advice would create  a non-termination loop
  //without the global declaration above
  before(): call(* *(...)){
    System.out.println("Entering a method");
  }
  ...
}
```

Würde diese Unterbindung des Einwebens nicht möglich sein, könnte es hierbei zu einem Fehler kommen, welcher eine Schleife erzeugt, die nicht mehr abbrechen würde.

Cast-Pointcuts sind eine wesentliche Vereinfachung des Abfangens möglicher Fehler bei *Cast*-Operationen. Der *Cast-Pointcut* greift dabei auf jeden impliziten oder expliziten *Cast* zu, welcher auf einen Typ mit diesem speziellen Muster (im Englischen *Type Pattern*) passt. Das folgende Beispiel beschreibt die Möglichkeit der Prüfung auf Informationsverlust, bei einem *Cast* von einem Integer-Wert zu einem Short-Wert.

```
aspect CastingBoundsCheck{
  before(int i): cast(short) && args(i) &&
    if(i<Short.MIN_VALUE || i>Short.MAX_VALUE){
      System.err.println("Warning: lossof information casting" + i
        + "to a short.");
    }
  }
}
```

Sollte der übergebene Integer-Wert aus dem Datenbereich von Short herauslaufen, so ist es auf vielerlei Wegen möglich diesem bevorstehenden Datenverlust zu verhindern oder eine Fehlermeldung über den Verlust auszugeben.

Dieser Abschnitt beschreibt die Implementierung der zuvor aufgezeigten Erweiterungen des *abc* Compilers. Diese Erweiterungen sind derzeit Teil von *abc* und liegen im Package abc.eaj. Der erste Anlaufpunkt, der als allgemeiner Startpunkt für die Entwicklung eigener Erweiterungen zu sehen ist, ist die `AbcExtension` Klasse. Deren Standardimplementierung liegt im Package abc.main. Als eine weitere Schlüsseldatei existiert die Klasse `ExtensionInfo`, welche Teil des Erweiterungsmechanismus von *Polyglot* ist. Alle Frontend-Erweiterungen sind hierbei registriert als Unterklassen dieser Klasse. Instanzen der Klasse `ExtensionInfo` werden durch die Unterklassen der Klasse `AbcExtension` erzeugt.

Der *abc* Lexer ist zustandsbehaftet. In ihm existieren vier Hauptzustände:

- *Java*,
- *AspectJ*,
- *Pointcut* und
- PointcutIfExpr.

Das folgende Beispiel, welches in der Datei abc\eaj\AbcExtension.java zu finden ist, zeigt die Bekanntmachung der neu eingeführten Schlüsselwörter `cast` und `global` für alle vier Lexer-Zustände:

```
public void initLexerKeywords(AbcLexer lexer){
  //keyword for the cast pointcut extension
  lexer.addPointcutKeyword("cast", new LexerAction_c(
    new Integer(abc.eaj.parse.sym.PC_CAST)));

  //keyword for the global pointcut extension
  lexer.addGlobalKeyword("global", new LexerAction_c(
    new Integer(abc.eaj.parse.sym.GLOBAL),
    new Integer(lexer.poincut_state())));

  //add the base keywords
  super.initLexerKeywords(lexer);
}
```

Dabei werden beide, mittels der Standardimplementierung der `LexerAction_c`, als *Pointcuts* eingeführt. Hierzu wird ein ein- bzw. zweiargumentiger Konstrutor aufgerufen. Das erste Argument bezeichnet immer das Parser-Token, welches im Anschluss als Schlüsselwort zurückgeliefert wird. Bei dem zweiten Argument, wenn vorhanden, handelt es sich um den Lexer-Zustand, welcher bei einem Aufruf ausgewählt werden soll. Möchte man weitere Logik implementieren, so kann auch die Klasse `LexerAction_c` erweitert werden.

Das folgende Beispiel, welches in der Datei abc\eaj\parser\eaj.ppg zu finden ist, veranschaulicht die Erweiterung des Parsers um die Erkennungsmöglichkeit der neu eingeführten Strukturen:

```
extend basic_pointcut_expr::=
PRIVATE:x LPAREN formal parameter list opt:a RPAREN
LPAREN pointcut_expr:b RPAREN:y
{:
  RESULT=parser.nf.PCLocalVars(parser.pos(x,y),a,b);
:}
|
PC_CAST:x LPAREN type_pattern_expr:a RPAREN:y
{:
  RESULT=parser.nf.PCCAST(parser.pos(x,y),a);
:}
;
```

Hierbei handelt es sich um eine *LALR* grammatikalische Erweiterung der Sprache.

Da *abc* aus Sicht von *Polyglot* nur eine Erweiterung ist, meint *abc* erbt alle Erweiterungsmechanismen von *Polyglot*, ist es somit unabdingbar, wenn neue Compilererweiterungen gebaut werden sollen, neue AST Knoten für den Compiler zu spezifizieren. Als allererstes ist es somit nötig die Definition eines Interfaces für jeden neuen AST Knoten vorzunehmen und die damit verbundene Funktionalität zu implementieren. Ein wichtiger Vorteil, welchen es dabei zu beachten gilt, ist die Veränderungen lokal in den neuen Klassen vorzunehmen. Dies gewährleistet die Robustheit von *abc* auch gegenüber Updates. Die folgenden beiden Beispiele veranschaulicht die Implementierung der globalen *Pointcut* Deklaration, welche im Verzeichnis abc\eaj\ast\GlobalPointcutDecl.java und abc\eaj\ast\GlobalPointcutDecl_c.java zu finden sind:

```
public interface GlobalPointcutDecl extends PointcutDecl{
  public void registerGlobalPointcut(GlobalPointcuts visitor,
    Context context, EAJNodeFactory nf);
}
public GlobalPointcutDecl GlobalPointcutDecl(Position pos,
  ClassnamePatternExpr aspect_pattern, Pointcut pc, String name,
  TypeNode voidn){
    return new GlobalPointcutDecl_c(pos, aspect_pattern, pc, name,
      voidn);
}
```

Um z.B. *Cast-* und *Throw-Pointcuts* implementieren zu können, muss als allererstes die Liste der *Joint-Point* Typen erweitert werden. Dies macht man, indem man diese der Liste von *Factory*-Objekten hinzufügt. Diese wird im Anschluss durch den *Pointcut* Matcher durchiteriert, um später alle *Join-Point Shadows* identifizieren zu können. Im Folgenden wird das Überschreiben der Methode `listShadowTypes()` aus der `AbcExtension` Klasse gezeigt, die in der Datei abc\eaj\AbcExtension.java zu finden ist:

```
protected List /* <ShadowType> */ listShadowTypes(){
  List /* <ShadowType> */ shadowTypes = super.listShadowTypes();
  shadowTypes.add(CastShadowMatch.shadowType());
  shadowTypes.add(ThrowShadowMatch.shadowType());
  return shadowTypes;
}
```

`CastShadowMatch` und `ThrowShadowMatch` sind hierbei die zwei neuen *Join-Point* Typen. Die Methode `shadowType()` gibt ein anonymes *Factory*-Objekt zurück, welches das Finden eines *Join-Points* zu einer statischen Methode delegiert. Die Implementierungen der beiden Match-Funktionen finden sich im Verzeichnis abc\eaj\waeving\matching. Das folgende Beispiel, welches in der Datei abc\eaj\weaving\matching
\CastShadowMatch.java zu finden ist, zeigt die statische Methode für `CastShadowMatch`:

```
public static CastShadowMatch matchesAt(MethodPosition pos){
  if(!(pos instanceof StmtMethodPosition)) return null;

  Stmt stmt = ((StmtMethodPosition)pos).getStmt();
  if(!(stmt instanceof AssignStmt)) return null;

  Value rhs = ((AssignStmt)stmt).getRightOp();
  if(!(rhs instanceof  CastExpr)) return null;

  Type cast_to = ((CastExpr)rhs).getCastType();

  return new CastShadowMatch(pos.getContainer(), stmt, cast_to);
}
```

Die Methode `matchesAt()` nimmt eine Struktur, welche eine Position im Programm zeigt, die eingewebt wurde und gibt entweder ein Objekt zurück, welches einen *Join-Point Shadow* repräsentiert, oder es wird null zurückgegeben. Dabei ist die Absicht durch den Parameter `pos` *abc* zu erlauben durch alle Teile durchzuiterieren, an denen ein *Join-Point Shadow* auftreten kann. Als allererstes ist es speziell in diesem Beispiel wichtig zu überprüfen, ob man sich an einem geeigneten Punkt für einen *Cast* befindet. Danach muss überprüft werden, ob an dieser Stelle wirklich ein *Cast* ausgeführt wird. Mit Hilfe von *Jimple* ist das relativ einfach zu bewerkstelligen, da eine *Cast*-Operation nur auf der rechten Seite einer Zuweisungsoperation stehen kann. Wird schließlich eine solche Operation gefunden, wird ein Zuweisungsobjekt erzeugt und zurückgegeben, andernfalls wird null zurückgegeben.

Nun geht es noch um die Erweitungen bzw. Bekanntmachung der beiden *Pointcuts Cast* sowie *Throw* im Backend von *abc*. Zuständig dafür ist der *Pointcut* Matcher, welcher im eigentlichen Sinne für das Auffinden eines jeden *Pointcuts* in einem *Join-Point* zuständig ist. Somit gilt für den definierten *Cast-Pointcut* nur zu erkennen, ob der jeweilige *Shadow* ein `CastShadowMatch` ist. In folgenden Beispiel, welches in der Datei abc\eaj\waeving\aspectinfo\Cast.java zu finden ist, gibt die Methode einen Wert mit `AlwaysMatch.v()` zurück, der als ein dynamischer Rest erkennen lässt, ob der *Pointcut* auf einen *Join-Point* matchen kann. Das meint, sollte statisch nicht erkennbar sein, ob ein Match vorliegt, so gilt es zusätzlichen Code einzufügen, mit diesem es zur Laufzeit möglich ist diese Prüfung durchzuführen.

```
protected Residue matchesat(ShadowMatch sm){
  if(!(sm instanceof CastShadowMatch)) return null;

  Type cast_to = ((CastShadowMatch)sm).getCastType();

  if(!getPattern().matchesType(cast_to)) return null;

  return AlwaysMatch.v();
}
```

AspectJ bietet dynamische als auch statische Informationen zu dem jeweils aktuellen *Join-Point*, durch z.B. eine Variable wie `thisJoinPoint` bzw. weitere Va-

riablen an. Speziell für die *Cast Pointcut* Erweiterung wurde das Interface erweitert, um die Signatur des darauf passenden *Casts* zu enthüllen. Im Allgemeinen ist es darum nötig, um solche Erweiterungen implementieren zu können, zwei verschiedene Änderungen durchzuführen. Diese Änderungen sind zum einen im Backend des Compilers durchzuführen, dort wo die statischen *Join-Point* Informationen abgelegt sind und zum anderen ist die Erstellung neuer Laufzeit Klassen und eines Interfaces durchzuführen. Bei den statischen *Join-Point* Informationen handelt es sich um in Strings codierte, abgelegte Informationen, welche durch `thisJoinPointStaticPart` zugänglich gemacht werden:

```
import org.aspectbenc.eaj.lang.reflect.CastSignature;

aspect FindCasts{
  before(): cast(*) && !within(FindCasts){
    CastSignature s = (CastSignature) thisJoinPointStaticPart.
       getSignature();

    System.out.println("Cast to:" + s.getCastType().getName());
  }
}
```

Diese Strings werden geparst und mittels *Factories* ist es möglich daraus die Objekte zu erstellen. Dies passiert nur einmal im statischen Initialisierer der Klasse, welche den statischen *Join-Point* enthält. Die statischen Informationen setzen sich wie folgt zusammen:

- Modifizierer (kodiert als ein Integer-Wert, z. B. 0 für einen Cast),
- Name (gewöhnlich eine Methode oder ein Feldname, für einen Cast ist es einfach cast),
- Declaring Typ (die Klasse in welcher der *Join-Point* auftritt) und
- Typ des Casts.

Ein Beispiel ist ein Cast *Join-Point*, welcher innerhalb der Methode der Klasse `IntHashTable`, welche den Wert von einer HashMap auf einen Integer abbildet (0-cast-IntHashTable-Integer).

abc wird als freie Software unter der *GNU Lesser General Public License* (kurz *LGPL*) vertrieben. *abc* ist aktuell in der Version 1.1.0 (Stand 11/2005) verfügbar.

Im Allgemeinen ist es mit *AOP* einfach möglich dem (*JAVA*-)Programmierer eine flexiblere und einfachere Handhabung von Programmelementen durch die Zentralisierung von wieder verwendbaren Regeln bzw. Modulen zu ermöglichen. So können durch die Änderung von Aspekten verschiedenste, dezentral abgelegte Programmteile mit neuen Anforderungen versehen werden.

Im Speziellen ermöglicht *abc* ein gutes Experimentieren mit bestehenden sowie eigenen Erweiterungen. Es bietet die Möglichkeit der Untersuchung des aspektorientierten Sprachraums und vereinfacht für den Nutzer das Verständnis über die Codeerzeugung des Verwebens durch Zuhilfenahme des Decompilers *Dava*. Durch welchen es einfacher möglich ist Optimierungen durchzuführen. So ist es zum Beispiel gelungen einen 45 mal schnelleren Code für Operationen mit `cflow` bzw. `cflowbelow` zu erzeugen. Durch diese beiden Operationen ist es möglich mittels *AspectJ* den Kontrollfluss eines Programms zu überwachen und daran Veränderungen vorzunehmen. `cflow` setzt dabei mit dem Aufruf einer Methode im Kontrollfluss an, wohingegen `cflowbelow` erst an den Aufrufen dieser Methode ansetzt.

Die Tabelle 2 veranschaulicht die optimalen Möglichkeiten der Implementierung von Erweiterungen in *abc* im Gegensatz zu *ajc*, da in dieser Tabelle zu erkennen ist, dass es einfacher möglich ist *abc* zu erweitern. Speziell der Kern von *abc* ist nicht einmal anzutasten, wohingegen alle Erweiterungen in *ajc* ausschließlich im Kern stattfinden. Somit gilt als Fazit, dass mit nur ein wenig mehr Zeilen Code, doch einer stärkeren Überschaubarkeit, *abc* für Entwickler und Forscher besser geeignet ist zur Entwicklung und zum Experimentieren eigener Erweiterungen.

Throw-*Pointcut* Statistik	**ajc**	**abc**
Kern Compiler / modifizierte Laufzeit Dateien	8	-
Throw-spezifisch generierte Dateien	2	6
Extension-spezifisch modifizierte *Factories*	-	5
Gesamt modifizierte Dateien	10	11
Lines of Code(LoC)	103	187

Tabelle 2: abc: Beispielstatistik Implementierung Throw Pointcut

abc bietet für Entwickler zum einen alternative Fehlermeldungen, in denen unter anderem auch Zeilennummern der aufgetretenen Fehler für den Quellcode enthalten sind und zum anderen schnelleren, optimierten Code als ihn *ajc* derzeit bietet. Forschern verspricht *abc* ein erweiterbares Frontend beruhend auf *Polyglot* sowie ein erweiterbares Backend beruhend auf *Soot*. Es gilt somit als eine ideale Plattform für Experimente mit neuen Features und Optimierungen der Sprache.

5.1.3 AspectC#

AspectC# ermöglicht den Einsatz von Konstrukten der *Aspektorientierten Programmierung* in der Sprache *C#*. Die Programmiersprache *C#* stammt aus dem *Microsoft .Net Framework*. Das *AspectC#* Framework entstand im Rahmen einer Dissertation von Howard Kim [2] am Trinity College Dublin im Jahre 2002. Im Fokus stand dabei die Aspekte modular zu organisieren. *AspectC#* vereint Features aus verschiedenen *AOP* Frameworks, wie z.B. *AspectJ, Hyper/J, CLAW* und *AOP#*. Eine Implementierung von *AspectC#* ist als Proof-Of-Concept verfügbar.

Das *AspectC#* Framework soll Mechanismen der *Aspektorientierten Programmierung* für die Sprache *C#* bereitstellen. Eines der Kernziele bei der Entwicklung von

AspectC# war dabei das keine Spracherweiterungen für *C#* eingeführt werden sollten. Der Quelltext sollte mit dem Standard-*C#*-Compiler compiliAerbar sein. Ein weiteres Kernziel war, dass Programm- und Aspektcode vollständig voneinander getrennt sein sollten. Weiterhin sollte das Framework für zukünftige Entwicklungen einfach erweiterbar sein. So wurde beim Entwurf bereits das sprachübergreifende Weben von Aspekten angedacht. Außerdem sollte das Framework einfach benutzbar sein und sich in vorhandene Projekte leicht integrieren lassen.

AspectC# bezeichnet einen definierten Punkt zur Laufzeit eines Programms als *Join-Point*. Ein *Join-Point* kann z.B. ein Methodenaufruf, der Zugriff auf eine Feldvariable, das Erzeugen eines Objektes oder das Werfen einer Exception sein. Eine Menge von *Join-Points* bezeichnet man auch als *Pointcut*.

Ein *Advice* definiert wann der Aspektcode ausgeführt werden soll. Man unterscheidet hier in der Regel drei Arten: *before*, *after* und *around*. *Before* bedeutet, das der Aspektcode vor dem eigentlichen Programmcode ausgeführt werden soll. Analog wird auch bei *after* verfahren. Das Schlüsselwort *around* ersetzt den Programmcode vollständig durch den Aspektcode.

Als Aspekt bezeichnet man eine modulare Implementierungseinheit bestehend aus *Pointcut*, *Advice* und der normalen Klassendeklaration.

Ein Aspekt sollte in *AspectC#* ebenfalls als regulärer *C#* Quelltext ausgedrückt werden können, jedoch mit der Erweiterungsmöglichkeit dies später auch in einer beliebigen *.Net* Sprache tun zu können.

Aus *Hyper/J* wurde der Ansatz übernommen nur deklarativ vollständige Aspekte zu erlauben. Deklarative Vollständigkeit bedeutet, dass alle Variablen und Methoden innerhalb eines Aspekts deklariert sein müssen. Dabei kann es sich aber auch um abstrakte Variablen oder Methoden handeln.

Der Zugriff auf den *Join-Point* soll mittels Reflection möglich sein.

Die Separation von Aspekten und Kernimplementierung wird durch eine Abbildung ermöglicht. Diese Abbildung von Aspektcode auf den regulären Programmquelltext wird in *AspectC#* mittels einer *XML*-Datei realsiert und als Aspekt Mapping bezeichnet. Man spricht hier auch vom *XML Deployment Descriptor*, der an die Stelle der aus *AspectJ* bekannten *Pointcut* Designators tritt.

Das *Join-Point* Modell aus *AspectJ* wird jedoch nur teilweise unterstützt. *AspectC#* erlaubt *Join-Points* nur an Methodenaufrufen, jedoch z.B. nicht an Zugriffen auf Felder.

Das Aspektweben, also das Zusammenführen von Aspektcode und Programmcode wird vor der Compilierung durch den Aspektweber bzw. Weaver gemacht. Dabei wird ein Aspekt mit den Programmmethoden verknüpft. Hierbei sind wie in *AspectJ* die *Advices* `before`, `after` und `around` möglich.

Abbildung 15 zeigt einen Überblick über die Komponenten des *AspectC#* Frameworks und deren Beziehungen in einem *FMC* Blockdiagramm [3]. Im Folgenden werden die einzelnen Komponenten in der Reihenfolge vorgestellt, in der sie auch beim Compiliervorgang verwendet werden.

Der Parser von *AspectC#* übernimmt sowohl das Parsen von Aspektcode, als auch

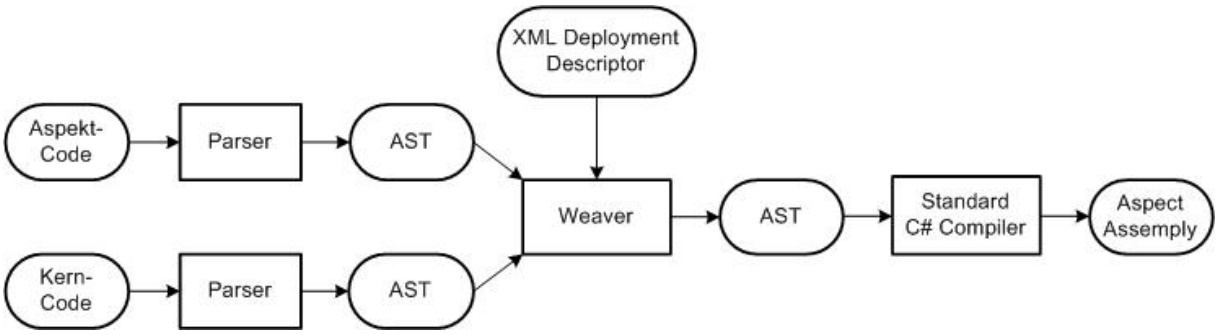


Abbildung 15: AspectC# Framework

das Parsen des normalen Programmcodes. Jedoch verfügt der Parser nur über einen sehr eingeschränkten Funktionsumfang. So werden nur grundlegende Syntaxelemente wie Namensräume, Klassen, Funktionen und Variablen erkannt. Der Parser generiert aus den Quelldateien einen Abstract Syntax Tree, auch AST. Dabei handelt es sich um eine Representation des Quellcodes als CodeDOM Objekt Graph. Die CodeDOM API kommt aus dem *.Net* Framework von *Microsoft* und bietet unter anderem auch die Möglichkeit die Struktur des Quellcodes sprachunabhängig als Objektgraph darzustellen. Mehrere in unterschiedlichen *.Net* Sprachen geschriebene Teile können so in einem einzigen CodeDOM Objekt Graph zusammengefügt werden. Ein solcher Graph kann über die CodeDOM API von den verschiedenen *.NET* Compilern compiliert werden.

Vom Weaver wird nun ein neuer Abstract Syntax Tree generiert der sowohl den Programmcode als auch den integrierten (verwebten) Aspektcode enthält. Dazu benötigt der Weaver die beiden ASTs und den *XML* Deployment Descriptor um die Abbildung zwischen Aspektcode und Programmcode herzustellen. Nun werden vom Weaver in *AspectC#* Methoden des regulären Programms mit Aspekten in einem neuen AST zusammengeführt. Für jeden Aspekt spezifiziert der *XML* Deployment Descriptor eine Zielmethode an der der Aspekt eingefügt wird.

Als Compiler für den Abstract Syntax Tree wird letztlich nur noch der Standard *C#* Compiler verwendet. Dieser erstellt anhand des „gewebten“ Quelltextes eine *.NET* Assembly.

Im folgenden wird eine einfache und beispielhafte aspektorientierte Anwendung mit *AspectC#* vorgestellt. Der reguläre und damit nicht aspektorientierte Teil der Applikation ist in der Klasse HelloWorld zu finden. Die Klasse `HelloWorld` verfügt über ein Attribut namens `testString`. Die Methode `SayHello()` schreibt `testString` auf die Standardausgabe. Weiterhin enthält die Klasse `Hello World` eine Methode `Add(int, int)` zum Addieren und eine `Main()` Methode.

Alle Aspekte sind in der separaten Klasse `AspectClass` definiert. Wir finden hier wiederum die Member-Variable `testString`, welche jedoch mit einer anderen Zeichenkette initialisiert wird als im regulären Programmtext. Die Methoden `before()` und `after()` sind kurze Aspektmethoden die jeweils nur eine Zeichenkette ausgeben. Mittels der `Add(int, int)` soll die Wirkung des *Advice* `around` gezeigt werden. Die

ursprüngliche Methode wird durch die Methode in der Aspektklasse ersetzt. Um dennoch Zugriff auf die original Methode zu erhalten kann die Methode `proceed()` verwendet werden. Die abstrakte Methode `proceed(int, int)` dient als Platzhalter zum Aufruf der original Methode `Add(int, int)` aus dem Quelltext.

```
public class HelloWorld{
   private string testString = "baseTestString";

   public void SayHello(){
      Console.WriteLine("HelloWorld: Hello World (" + testString +
          ")");
   }

   public int Add(int a, int b){
      return a+b;
   }

   public static void Main(){
      HelloWorld hw = new HelloWorld();
      hw.SayHello();
      Console.WriteLine("HelloWorld: Adding 6+5: " + hw.Add(6,5));
   }
}
public abstract class AspectClass{
   private string testSrting = "aspectualTestString";

   public abstract int proceed(int a, int b);

   public AspectClass(){
      Console.WriteLine("AspectClass: Constructor called");
   }

   public void before(){
      WriteLine("AspectClass: Before Hello");
   }

   public void after(){
      WriteLine("AspectClass: After Hello ("+testString+ ")");
   }

   public int Add(int a, int b){
      Console.WriteLine("AspectClass: Around Add");
      if (a+b > 10) { return -1; }
      else { return proceed(a, b); }
   }
}
```

Das Einfügen von Aspektcode in den Programmcode kann über das *C#* Attribut `Introcuction("class name")` realisiert werden. In diesem Beispiel wird die Methode

`Sub(int, int)` mittels Introduction in die Klasse `HelloWorld` eingeführt.

```
public abstract class AspectClass{
   [Introduction("HelloWorld")]
   public int Sub(int a, int b){
      return (a-b);
   }
}
```

Der *XML* Deployment Descriptor stellt das Bindeglied zwischen Programmcode und Aspektcode dar. Mit den Tags <*TargetBase*> und <*AspectBase*> werden die Verzeichnisse angegeben, in denen sich die Quelltexte des Programms und der Aspekte befinden. Durch das <*Aspect-Method*>-Tag werden alle zur Verfügung stehenden Methoden identifiziert und mit einem Namen verknüpft. Im Target Abschnitt des Deployment Descriptors wird dann die `SayHello()` Methode mit der Aspektmethode verwebt. Dabei muss der Weber natürlich auch wissen, welcher Advice für die Methode gilt. Diese Information befindet sich im Tag <*Type*>.

Zum Compilieren kann die aus dem *AspectC#* Code Beispiel [2] stammende Batch-Datei run.bat verwendet werden. Um ein Compilat zu erstellen, können die vom *AspectC#* Weaver erzeugten temporären Dateien verwendet werden. Zunächst muss dazu das Kommando zum Starten des *C#* Compilers aus der .out Datei extrahiert werden. Dem Compiler werden vier oder auch mehr Dateien zur Compilierung übergeben.

```
>_output.exe
HelloWorld: Constructor called
AspectClass: Before Hello
HelloWorld: Hello World (baseTestString)
AspectClass: After Hello (baseTestString)
AspectClass: Around Add
HelloWorld: Adding 6+5: -1
```

Das Programm verhält sich beim Ausführen nun bis auf zwei Details wie erwartet. Zum einen wird der Konstruktor der Aspektklasse nicht ausgeführt und zum anderen wird „baseTestString“ anstatt „aspectualTestString“ ausgegeben. Dies deutet darauf hin, dass der Aspektweber gegenwärtig nur auf Methodenebene arbeitet. Somit ist davon auszugehen, das es sich bei einer Aspektklasse nicht um einen abstrakten Datentyp (ADT) handelt. Einen weiteren Hinweis dafür liefert die Tatsache, dass anstelle von „aspectualTestString“ die Zeichenkette „baseTestString“ ausgegeben wurde. Da in *AspectC#* aber ohnehin die deklarative Vollständigkeit von Aspektmethoden verlangt, wird dieses Problem teilweise entschärft. Einen tieferen Einblick in die Funktionsweise des *AspectC#* Weavers erlauben auch die oben erwähnten temporären Dateien.

Im Folgenden wird die Frage erörtert inwiefern sich das *Microsoft .NET Framework* für die aspektorientierte Entwicklung eignet. Das *Microsoft .NET Framework* bietet zahlreiche Features die die Entwicklung von aspektorientierten Frameworks erleichtern und *.NET* damit zu einer guten Plattform für aspektorientierte Entwicklung machen.

Mittels Attributen wie *[WebMethod]* können Metadaten in den Quelltext eingebracht werden. Diese Metadaten können dann wiederum durch Reflection zur Lauf-

zeit gelesen werden. Über eigene Attribute kann man sogar ganze Aspekte schreiben. *AspectC#* benutzt das eigene Attribut *[introduction(<string>)]* um Aspektcode in Programcode einfügen zu können.

```
<?xml version="1.0" encoding="utf-8" ?>
<Aspect>
   <TargetBase>target</TargetBase>
   <AspectBase>aspect</AspectBase>
   <Aspect-Method>
      <Name> AspectBefore </Name>
      <Class> AspectClass </Class>
      <Method> before() </Method>
   </Aspect-Method>
   <Aspect-Method>
      <Name>AspectAfter</Name>
      <Namespace>Hello</Namespace>
      <Class>AspectClass</Class>
      <Method>after()</Method>
   </Aspect-Method>
   <Aspect-Method>
      <Name>AspectAround</Name>
      <Namespace>Hello</Namespace>
      <Class>AspectClass</Class>
      <Method>Add(int a, int b)</Method>
   </Aspect-Method>
   <Target>
      <Class> HelloWorld </Class>
      <Method>
         <Name> SayHello() </Name>
         <Type> before </Type>
         <Aspect-Name> AspectBefore </Aspect-Name>
      </Method>
      <Method>
         <Name>SayHello()</Name>
         <Type>after</Type>
         <Aspect-Name>AspectAfter</Aspect-Name>
      </Method>
      <Method>
         <Name>Add(int a, int b)</Name>
         <Type>around</Type>
         <Aspect-Name>AspectAround</Aspect-Name>
      </Method>
   </Target>
</Aspect>
```

Die CodeDOM API ist ein mächtiges Werkzeug zum spachübergreifenden Entwickeln. Ein mit der CodeDOM API erstellter Syntaxbaum bildet auch die Grundlage zum Verweben von Aspekten, die in verschiedenen Programmiersprachen entwickelt wurden. Dies ist möglich, da alle *.NET* Sprachen in eine Maschinenunabhängige Zwi-

schensprache, die *Microsoft Intermediate Language* (kurz *MSIL*), übersetzt werden. Zur Laufzeit wird durch einen Just-In-Time Compiler daraus dann nativer Maschinencode. Dieses Vorgehen ähnelt sehr dem aus *Java* bekannten Bytecode Konzept

Bei *AspectC#* handelt es sich um eine Proof-Of-Concept Implementierung. Dementsprechend gibt es auch nur einen sehr geringen Funktionsumfang und das Framework ist zudem nicht Fehlerfrei. Einfache Anwendungen können jedoch mit *AspectC#* entwickelt werden. Dies zeigt, das es mit moderatem Aufwand möglich ist eine aspektorientierte Umgebung innerhalb des *Microsoft .Net Frameworks* zu schaffen. Besonders die Verwendung von Syntaxbäumen erleichtert das Parsen und Compilieren und stellt zusätzlich eine Möglichkeit bereit aspektorientierte Entwicklungen auch in verschiedenen Programmiersprachen zu mischen.

Das Framework ist frei erhältlich [2], wird aber augenscheinlich nicht mehr weiterentwickelt. Für eine aspektorientierte Entwicklung in *C#* bieten sich jedoch auch zahlreiche andere *AOP* Frameworks, wie *LOOM.NET* [4], *Weave.NET* [5], *AspectSharp* [6] und *Aspect.NET* [7] an.

5.1.4 AspectC++

AspectC++ [8] erweitert die Programmiersprache *C++* zur Definition von Aspekten. *AspectC++* definiert ebenso wie *AspectJ* (siehe Kapitel 5.1.1) Aspekte in einer Komponente namens *aspect*. Auch die Definition der Verwebungspunkte und *Advices* wird wie bei *AspectJ* (siehe Kapitel 5.1.1) vorgenommen. Zusätzlich bietet *AspectC++* die Möglichkeit der Vererbung von Aspekten.

Der Aspektweber *AspectC++* setzt sich aus dem Werkzeug *PUMA C++* und dem Plugin *AspectC++* zusammen.

PUMA C++ ist ein Programm zur Transformation von *C++* Quellcode. Das Plugin *AspectC++* ist ein Vorübersetzer für *C++*.

Die Abbildung 16 stellt den Ablauf der Verwebung von *C++* Quellcode mit dem Aspektcode dar. Während des Verwebungsprozesses im Aspektweber *AspectC++* scannt, parst und analysiert *PUMA C++* zuerst die Semantik des Aspektcodes. Die gewonnenen Informationen werden an das Plugin *AspectC++* weitergeleitet, welches einen Ablaufplan mit Kommandos zur Manipulation des Quellcodes generiert. Dieser Ablaufplan wird von *PUMA C++* zur Transformation des Quellcodes ausgeführt. Das Resultat ist der verwobene Quellcode.

AspectC++ ist ein universitäres Forschungsprojekt. *AspectC++* ist in der Version 1.0pre1 frei verfügbar für *Linux, Windows, Solaris* und *Mac OS X*.

Außerdem gibt es *AspectC++* unter dem Namen *Eclipse AspectC/C++ Development Tools* (kurz *ACDT*) als *Eclipse* Plugin in der Version 2.0.3 für *Eclipse* 3.0.x.

Zusätzlich gibt es noch ein Addin von *AspectC++* für *Microsoft Visual Studio* von der Firma *pure-systems GmbH* [9]. Das Addin gibt es als kommerzielle Software für *Microsoft Visual C++* in der Version 1.0.12. Außerdem bietet *pure-systems GmbH* ein *Eclipse* Plugin frei verfügbar in der Version 2.0.0 an und ein *Eclipse* Plugin in der Version 2.0.0 zur Evaluierung.

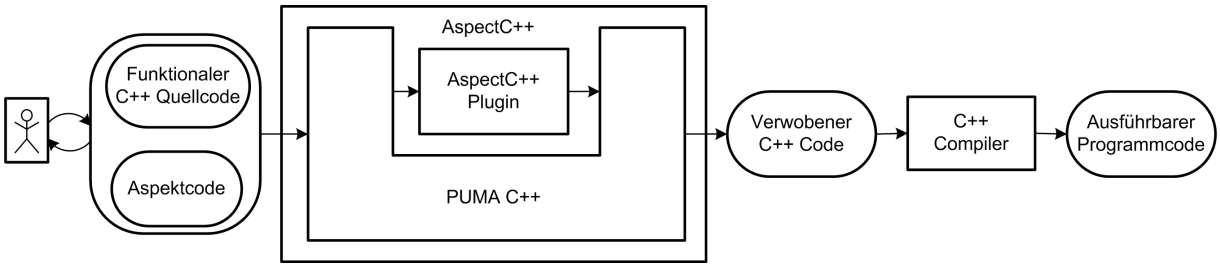


Abbildung 16: AspectC++: Verwebungsprozess

Ein Beispiel Aspekt in *AspectC++* ist der Folgende:

```
aspect MyAspect{
  pointcut MyPointcut() = call("void MyClass::\%(int)");

  advice MyPointcut : void around(){
    cout << "before" << endl;
    proceed();
    cout << "after" << endl;
  }
};
```

Ein Vorteil von *AspectC++* gegenüber *AspectJ* ist die Möglichkeit Aspekte zu vererben. Ansonsten sind sich *AspectC++* und *AspectJ* sehr ähnlich. Lediglich die Syntax ist unterschiedlich und das ein Aspekt in einer Aspektheader Datei (.ah) definiert werden muss.

5.1.5 Composition Filters

In der realen Welt gibt es die Möglichkeit der modularen Erweiterung, um das Verhalten von Gegenständen durch das Hinzufügen zusätzlicher Funktionalität zu verändern. Beispielsweise kann man eine Kamera mit verschiedenen Objektiven und Filtern ausstatten. Denkbar sind Objektive, die ein Fokussieren von Motiven erlauben oder Farbfilter, die auf gegebene Lichtverhältnisse reagieren.

In der *Objektorientierten Programmierung* ist das Erweitern von Objekten um Funktionalität bisher nicht modular möglich. Oft muss eine Klasse abgeleitet werden, so dass ein bestimmter Funktionsumfang durch weitere Methoden hinzugefügt werden kann. Darüber hinaus müssen häufig in der Basisklasse spezifizierte Methoden reimplementiert werden, um im Zusammenspiel mit den neuen Methoden die gewünschte zusätzliche Funktionalität zu erhalten. Ferner sind Objektaufrufe zu modifizieren, was einen erheblichen Mehraufwand bedeutet. Sollen nun im Verlauf der Entwicklung nochmals Eigenschaften zu einem Objekt hinzugefügt werden, so muss der eben beschriebene Prozess der Anpassung und Reimplementierung erneut durchgeführt werden.

Professor Mehmet Aksit forscht seit den 80-iger Jahren an aspektorientierten Techniken speziell an den von ihm entwickelten *Composition Filters*, mit der Ziel-

stellung, den oben genannten Problemen zu begegnen. Ziel ist es, ähnlich wie bei den Filtern für Kameras, durch die Betrachtung bzw. Veränderung der zwischen Objekten ausgetauschten Nachrichten, Objektfunktionalität modular zu erweitern. Seine Forschungsgruppe TRESE Group arbeitet am Institut für Informatik an der Universität von Twente in Holland.

Das *Composition Filters* Modell ermöglicht die Modifikation von Nachrichten die zwischen Objekten ausgetauscht werden.

Nachrichten bilden die Grundlage der Kommunikation zwischen Objekten. Zu ihnen gehören Methodenaufrufe ebenso wie die Abfrage von Attributwerten. In Bezug auf *Composition Filters* ist eine Nachricht in Form eines Methodenaufrufes der Ausgangspunkt für die Adaption der Funktionalität eines Objektes. Eine Nachricht (siehe 17) ist in target und selector unterteilt. Das target definiert das Objekt, welches die Message empfangen soll, der selector die auszuführende Methode.

Durch die Modifikation der ankommenden und abgehenden Nachrichten kann das Verhalten eines Objektes maßgeblich beeinflusst werden, indem zum Beispiel eine andere Methode als die aufgerufene ausgeführt wird. Zur Modifikation der ankommenden und abgehenden Nachrichten eines Objektes wird ein spezielles *Composition Filter* Objekt definiert.

Ein *Composition Filter* Objekt (siehe Abbildung 18) repräsentiert einen Aspekt im *Composition Filter* Modell. Ein *Composition Filter* Objekt besteht aus seinem Innenteil und dem Interface [10]. Der innere Teil eines Objektes enthält dessen Standardkonstrukte: Methoden und Attribute. Die Methoden enthalten die Grundfunktionalität, also das Verhalten. Sie implementieren die funktionalen Eigenschaften eines Objektes und werden durch die Nachrichten aufgerufen. Die Attribute dienen der transienten Zustandsspeicherung. Auf der Grenze zwischen dem Innenteil und dem Interface liegen Bedingungen. Bedingungen sind Methoden mit einem Boolschen Rückgabewert. Sie bestimmen ob eine Nachricht an den Innenteil eines Objektes weitergeleitet werden darf oder nicht und geben Auskunft über den Zustand des Objektes. Bedingungen werden im Interface, das in einer separaten Datei gekapselt ist, spezifiziert und sind von Außen sichtbar. Trifft eine Nachricht auf ein *Composition Filter* Objekt, so werden die Bedingungen von den Filtern abgefragt und verifiziert. Interne Referenzen sind mit herkömmlichen Variablen vergleichbar und werden ebenso deklariert. Sie ermöglichen das Hinzufügen und Kombinieren von Verhalten zu einem Objekt und somit die modulare Erweiterung der Funktionalität, die im Fokus der *Composition Filters* steht. Bei der Instanzierung des zugehörigen Objektes werden auch die internen Referenzen erzeugt. Als Verbindung zu außerhalb liegenden Objekten dienen externe Refrenzen. Sie sind mit globalen Variablen vergleichbar und sind folglich vom Innenteil, vom Interface, aber auch von anderen Objekten referenzierbar. Die wesentlichsten Elemente für die Adaption der Funktionalität eines Objektes sind die Input und Output Filter. Sie modifizieren das Verhalten eines Objektes. Sie sind dafür zuständig, die an ein Objekt adressierten Nachrichten anzunehmen bzw. weiterzuleiten, zu verwerfen oder zu verändern. Im Kontext der *Aspektorientierten Programmierung* definiert ein Filter einen Verwebungspunkt. Ein Filter für ankommende Nachrichten ersetzt die Ausführung einer Methode. Ein Filter für abgehende Nachrichten dagegen ersetzt den Aufruf einer

Methode. Mehrere Filter für ankommende und abgehende Nachrichten werden geordnet zu so genannten Filterketten zusammengefasst.

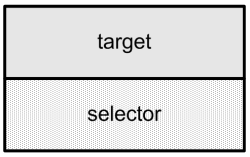


Abbildung 17: Composition Filters: Nachricht

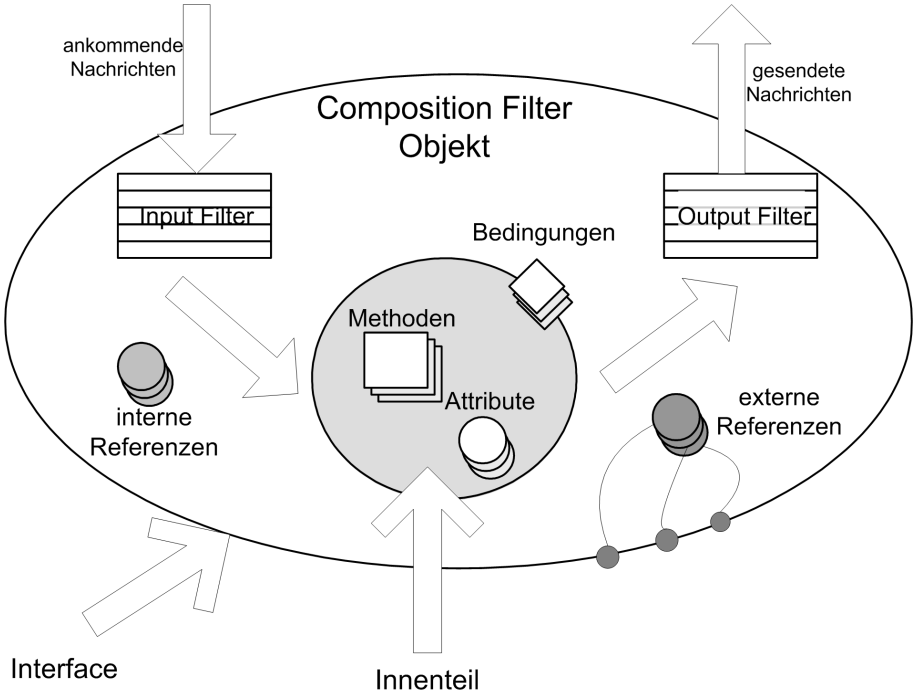


Abbildung 18: Composition Filters: Composition Filter Objekt

Wie alle Classifier, benötigen auch *Composition Filters* eine eineindeutige Bezeichnung in ihrem jeweiligen lokalen Kontext: den Filternamen. Ebenso muss bei der Deklaration der Filtertyp angegeben werden, der das Verhalten eines Filters bestimmt. Zu einem Filter gehören darüber hinaus ein oder mehrere Filterelemente. Jedes Filterelement besteht aus den drei Hauptbestandteilen: condition, matching part und substitution part [11]. Zwischen condition und matching part wird ein Operator angegeben.

```
inputfilters
  myFilter : <FilterType> {
    //Filterelemente
    condition1 => {matchingPart1}{substitutingPart1},
    condition2 ~> {matchingPart2}{substitutingPart2}
  }
```

Die condition wurde bereits als Bedingungen im Zusammenhang mit dem *Composition Filter* Objekt eingeführt und bestimmt, ob das jeweilige Filterelement die eingehende

oder ausgehende Nachricht evaluiert. Wird keine condition angegeben, so wird die Bedingung als true angenommen. Der Operator hat ebenfalls Einfluss auf die Nachrichtenauswertung. Filterelement eins nutzt einen enable-Operator. Das heißt, jede Nachricht, die den matching part erfüllt, ist kompatibel. Der exclusion-Operator von Filterelement zwei bewirkt, dass nur Nachrichten anerkannt werden, die nicht mit dem matching part übereinstimmen. Der matching part grenzt demnach Anforderungen ab, denen eine Nachricht genügen muss, um von dem jeweiligen Filterelement angenommen zu werden. Wird eine Nachricht anerkannt, so kommt, wenn vorhanden, der substituting part zur Ausführung und verändert gegebenenfalls die Nachricht. Ähnlich wie eine Message in target und selector unterteilt ist, bestehen auch matching und substituting part aus diesen beiden Teilen.

Jede ankommende oder abgehende Nachricht eines Objektes muss jeden Filter einer Filterkette im *Composition Filter* Objekt passieren bevor sie angenommen oder abgelehnt wird. (siehe Abbildung 19) Dieser Vorgang wird als Pattern-Matching bezeichnet [10]. Dabei werden der Sender, der Empfänger, die Signatur einer Methode und die zu erfüllenden Bedingungen einer Nachricht überprüft. Eine Filter Aktion wird nur dann ausgeführt, wenn die Bedingung erfüllt ist und für die aufgerufene Methode ein Filter definiert ist. Nach der Ausführung der Filteraktion wird die Nachricht zum nächsten Filter in der Filterkette weitergereicht. Wenn die Bedingung nicht erfüllt war, dann wird die Nachricht gleich an den nächsten Filter in der Filterkette weitergereicht. Wenn keine der Filterbedingungen erfüllt wurde, dann wird die Nachricht an den Error-Filter weitergereicht, der eine Exception generiert.

Einige Filter sind in der *Composition Filter* Laufzeitumgebung vordefiniert [12]. Welche Filter das sind ist im Folgenden beschrieben:

Error-Filter beschränkt die Schnittstellen eines Objektes nach Außen. Als Inputfilter definiert er Vorbedingungen (im Englischen preconditions), als Outputfilter Nachbedingungen (im Englischen postconditions). Es ist also möglich Nachrichten daran zu hindern, ein Objekt zu erreichen bzw. zu verlassen. Wird eine Message von einem Error-Filter angenommen, so wird sie an den nächsten Filter weitergegeben. Andernfalls wird eine Exception erzeugt.

Dispatch-Filter kann nur als Inputfilter genutzt werden kann. Der Dispatch-Filter sendet den Methodenaufruf einer Nachricht –sofern sie von dem Filter akzeptiert wird –zur Ausführung direkt an ein Objekt (im Englischen dispatched). Dabei ist das aktuell im target festgelegte Objekt der Empfänger des Aufrufes, welches die aktuell im selector aufgeführte Methode ausführt. Wird die Nachricht von keinem der Filterelemente anerkannt, so wird sie zum nächsten Filter weitergereicht.

Send-Filter ist im Gegensatz zum Dispatch-Filter nur als Outputfilter nutzbar. Es ist somit möglich, Nachrichten an ein Objekt zu delegieren, wenn der Filter die Nachricht annimmt. Wird die Nachricht von keinem der enthaltenen Filterelemente angenommen, beginnt der nächste Filter die weitere Auswertung.

Wait-Filter Hier steht die Synchronisation nebenläufiger Aufrufe im Fokus. Wird eine Nachricht von einem solchen Filter abgelehnt, so wird sie blockiert bis bestimmte Bedingungen zu true evaluiert werden können. Die Auswertung der Nachricht

wird dann erneut vorgenommen. Während eine Nachricht blockiert ist, können andere Nachrichten den Filter passieren, wodurch noch immer Methodenaufrufe an das jeweilige Objekt gelangen. Wird eine Nachricht sofort von einem Wait-Filter akzeptiert, so wird sie an den nächsten Filter weitergegeben.

Meta-Filter Mit ihm besteht die Möglichkeit Betrachtungen der Objektkommunikation vorzunehmen, also z.B. Monitoring Funktionen zu implementieren. Eine positiv evaluierte Nachricht wird dabei als Parameter in einer Meta-Nachricht gekapselt und an ein externes Objekt weitergeleitet. Dieses Objekt könnte z.B. einen Counter implementieren und eine bestimmte Art Methodenaufrufe zählen. Die Metanachricht wird im Zuge der Auswertung entpackt und die als Parameter übergebene Nachricht reaktiviert. Akzeptiert ein Meta-Filter eine Nachricht nicht, so wird sie an den nächsten Filter gereicht.

Substituting-Filter können eine Nachricht verändern. Dabei werden das target, der selector oder beide durch die als Parameter übergebenen Werte ersetzt. Wird eine Nachricht von einem Substituting-Filter nicht angenommen, so wird sie zum nächsten Filter weitergereicht.

In einer Menge von Inputfiltern wird als letzter Filter in der Regel ein Dispatch-Filter verwendet wird, um die ursprüngliche Nachricht –also einen Methodenaufruf –auszuführen. Bei einer Menge von Outputfiltern trifft Gleiches für den Send-Filter zu, um die von einem Objekt versendete Nachricht an ein externes Objekt weiterzuleiten. Außer den beiden eben genannten Filtern sind alle Filtertypen als In- bzw. Outputfilter verwendbar. Jeder Filtertyp generiert eine Exception, wenn eine Nachricht an den nächsten Filter weitergeleitet werden soll, aber kein weiterer zur Evaluierung verfügbar ist.

Es gibt verschiedene Realisierungen in verschiedenen Programmiersprachen für Composition Filters [13]:

Sina/st wurde im Rahmen einer Masterarbeit im Jahre 1995 entwickelt. *Sina/st* realisiert den kompletten *Composition Filter* Ansatz. *Sina/st* generiert *Smalltalk* Quellcode. Die *Composition Filter* werden zu Laufzeit analysiert. Der Wait-, Dispatch-, Error- und Meta-Filter wurde umgesetzt.

C++ wurde ebenfalls im Rahmen einer Masterarbeit im Jahre 1995 entwickelt. *C++* generiert *C++* Quellcode. Die *Composition Filter* werden zu Laufzeit analysiert. Der Error-, Dispatch-, Send- und Substituting-Filter wurde umgesetzt.

ComposeJ ist eine *Composition Filter* Realisierung für *Java*. Im Rahmen der Masterarbeit von H. Wichmann entstand 1999 eine Proof-of-Concept-Implementierung mit eingeschränktem Funktionsumfang. Aktuell ist *ComposeJ* in der Version 3.4 verfügbar (11.2003). Error- und Dispatch-Filter sind nutzbar. *ComposeJ* ist eine Erweiterung des Standard *Java* Compilers. Die *Composition Filter* werden zur Compilezeit analysiert. Die Performanzvorteile von compilezeitbasiertem Weben standen bei der Entwicklung von *ComposeJ* im Vordergrund. Auf die höhere Adaptibilität, die mit laufzeitbasiertem Weben erreicht werden kann, wurde verzichtet.

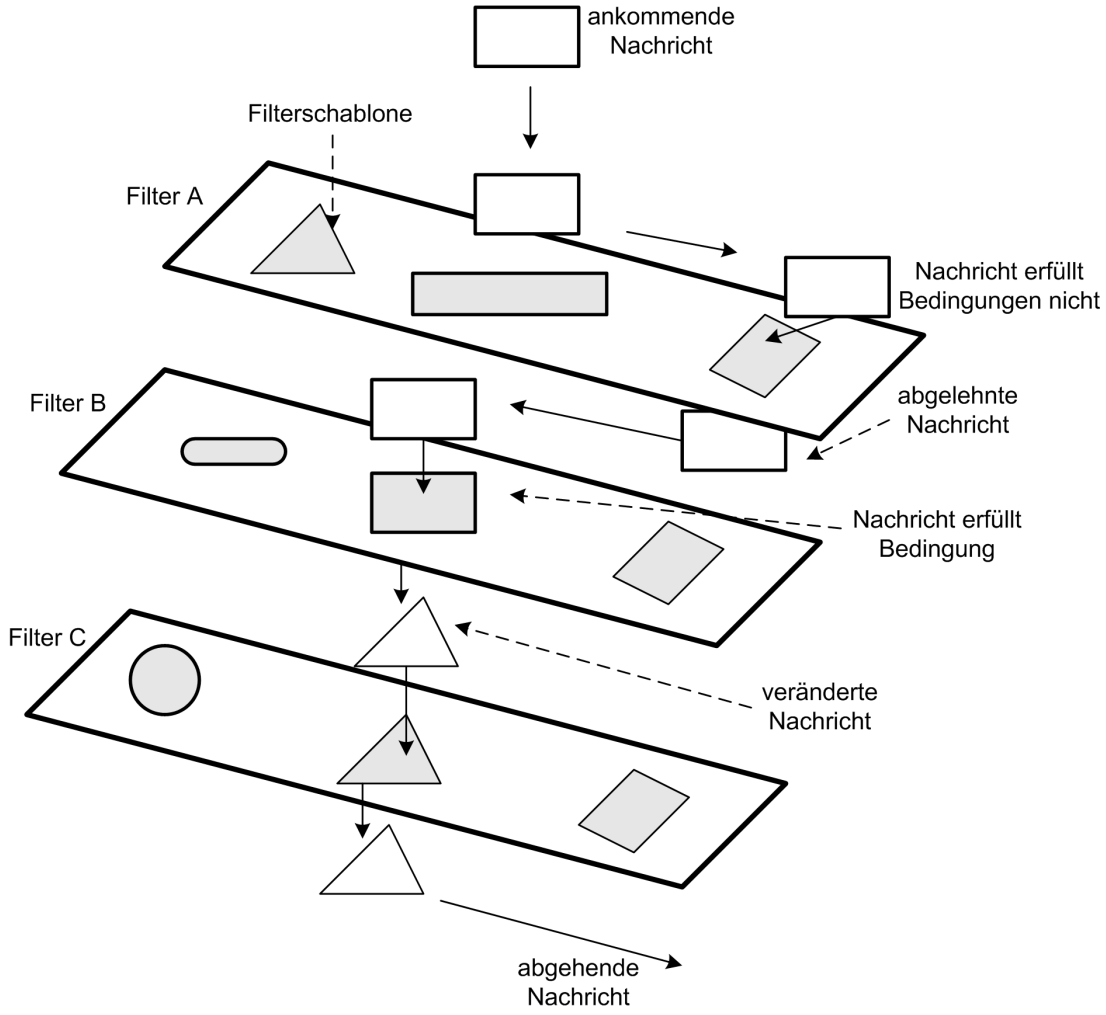


Abbildung 19: Composition Filter: Pattern-Matching

Compose* wird in einem, von der TRESE Group initiierten, Open-Source Projekt unter der *GNU Library or Lesser General Public License* entwickelt [14]. Das Ziel der Entwicklung ist die Einbindung von *Composition Filter* in alle Sprachen des *.NET Frameworks*. In der aktuellen Version (*Compose*.NET* 0.4b, 06.2005) unterstützt *Compose* J#* und *C#*, sowie Dispatch-, Error-, Send- und Meta-Filter. Ebenso wie bei *ComposeJ* wird der Aspektcode zur Compilezeit in den Programmcode eingewoben. Darüber hinaus existiert noch eine zweite Verarbeitungsschicht; zur Laufzeit ist *FLIRT (FiLter InteRpreTer)* für die eigentliche Realisierung der durch Filter spezifizierten Aspekte verantwortlich [15].

Folgende zwei Beispiele sind [10] entnommen.

```
select: Error = { IsBirthday=>[*.getPresent]*.* };
```

Der Filter trägt den Namen `select` und ist laut dem Schlüsselwort nach dem Doppelpunkt ein Error-Filter. Danach sind in einem, durch geschweifte Klammern abgetrennten, Block die Filterelemente angegeben. In diesem Fall ist ein Element spezifiziert. Die Bedingung des Filterelements heißt `isBirthday`. Man kann sich vorstellen, dass das Element nur evaluiert wird, wenn das Zielobjekt der Message Geburtstag hat. Nach dem Operator folgt nun der matching part `*.getPresent`, welcher in eckigen Klammern gekapselt ist und die nachstehende Bedeutung hat: target ist ein beliebiges Objekt (Platzhalter sind durch * symbolisiert), selector ist die Methode `getPresent()`. Der substituting part besteht aus zwei Platzhalter, die Message wird also nicht verändert. Somit passieren, wenn die Bedingung `isBirthday` wahr ist, alle Messages diesen Filter, die egal auf welchem Objekt, die Methode `getPresent()` aufrufen.

```
disp : Dispatch = { inner.* };
```

Der Beispielcode zeigt einen Dispatch-Filter mit dem Namen `disp`. Er beinhaltet ebenfalls ein Filterelement, welches immer ausgewertet wird, weil keine Bedingung angegeben ist. Akzeptiert werden alle Messages an das Objekt, welches diesen Filter besitzt, gleich welche Methode aufgerufen wird. Ein substituting part ist ebenfalls nicht angegeben. Die in der Nachricht übergebene Methode wird durch das Objekt ausgeführt.

Im Folgenden wir ein Codebeispiel für einen *Composition Filter* in *Compose** vorgestellt. Als Ausgangspunkt ist eine Vererbungshierarchie aus der Natur (siehe Abbildung 20) gegeben. Die abstrakte Klasse `LivingBeing` stellt eine Methode `grow()` bereit, die es den Lebewesen ermöglicht zu wachsen. Lebewesen sind repräsentiert durch die Klassen `Plant` und `Animal`. Pflanzen wachsen aufgrund von Sonnenergie und Photosynthese, Tiere aufgrund der Verdauung von Nahrung. Es soll nun die Klasse `VenusFlyTrap` in die Hierarchie integriert werden. Als Pflanze wächst die Venus Fliegenfalle ebenfalls durch Sonnenenergie. Zugleich kann sie aber auch durch die Verdauung von Fliegen wachsen. Im linken Teil der Abbildung 20 ist ein erster Integrationsversuch zu sehen, der auf einer meist schwer implementierbaren Doppelvererbung beruht. Rechts hingegen erbt die neu einzuführende Klasse von `Plant` und wird dann modular durch einen Aspekt erweitert, der das Wachsen durch Verdauung

mit Hilfe eines *Composition Filter* realisiert. Für die Klasse `VenusFlyTrap` ist zusätzlich nur eine Dummy Methode notwendig, die einen Compilerfehler verhindert. Es wird so ermöglicht die Methode `catchFly()` auf einem Objekt von `VenusFlyTrap` aufzurufen. Die vollständige Klasse sieht demnach wie folgt aus:

```
public class VenusFlyTrap extends Plant {
  public void catchFly() {
    // Dummy method for the J# compiler
  }
}
```

Zu dieser Klasse existiert der folgende *Composition Filter*:

```
concern VenusFlyTrap in NamespaceXYZ {
  filtermodule InsectTrap {
    internals
    a : NamespaceXYZ.Animal;
    conditions
    isfly : a.hasPrey();
    inputfilters
    eat : Dispatch = {[*.catchFly] a.catchPrey };
    grow : Dispatch = {isfly => [*.grow] a.grow }
  }
  superimposition {
    selectors
    files = {C|isClassWithName(C,'NamespaceXYZ.VenusFlyTrap')};
    filtermodules
    files <-InsectTrap;
  }
}
```

Die Definition eines Interfaces erfolgt innerhalb eines Aspektes (Schlüsselwort *concern*). Das Schlüsselwort hierfür ist *filtermodule*. Als „private Variable“ (interne Referenz) wird ein Objekt `a` der Klasse `Animal` erzeugt; es existiert ferner die Bedingung `isfly`, die wahr ist, wenn `a` im Besitz von Beute ist (`a.hasPrey()`). Hiernach folgt die Definition von zwei Filtern mit je einem Filterelement. Der Dispatch Filter `eat` leitet jeden Aufruf der Dummy Methode `catchFly()` an das Objekt `a` weiter. Der neue *selector* ist die Methode `catchPrey()` der Klasse `Animal`. Durch diese Umleitung des Methodenaufrufes `catchFly()` wird die Bedingung `isfly` auf true gesetzt. Ruft man jetzt die Methode `grow()` der Klasse `VenusFlyTrap` auf, so wird dieser Aufruf mittels des zweiten Dispatch Filters `grow` erneut an das Objekt `a` weitergeleitet, um dessen Methode `grow()` aufzurufen. Infolgedessen wächst die Venus Fliegenfalle aufgrund der Verdauung von Nahrung und die Bedingung `isfly` wird auf false gesetzt. Würde man nun erneut die Methode `grow()` der Klasse `VenusFlyTrap` aufrufen, so würde der Aufruf einfach an die Basisklasse, also an die Klasse `Plant` gegeben werden. Durch das Schlüsselwort *superimposition* wird der Abschnitt eingeleitet, der spezifiziert welches Objekt der oben beschriebene *Composition Filter* erweitert. Es wird eine Variable `files` erzeugt, die mit dem Namen der Klasse `VenusFlyTrap` –für die der Filter bestimmt ist –belegt wird. In einem zweiten Schritt weist man dieser Klasse dann

ein Filtermodul zu. In diesem Fall das eben erläuterte Filtermodul `InsectTrap`. Durch das Hinzufügen eines *Composition Filters* gelang es, die Klasse `VenusFlyTrap` wie gewünscht modular zu erweitern. Eine Venus Fliegenfalle ist nun in der Lage Fliegen zu verdauen. Erreicht wurde das Ziel ohne die Reimplementierung der Methode `grow()` bzw. dem Hinzufügen einer zweiten `grow()` Methode, verbunden mit der Änderung der zugehörigen Aufrufe, z.B. durch Parameterisierung.

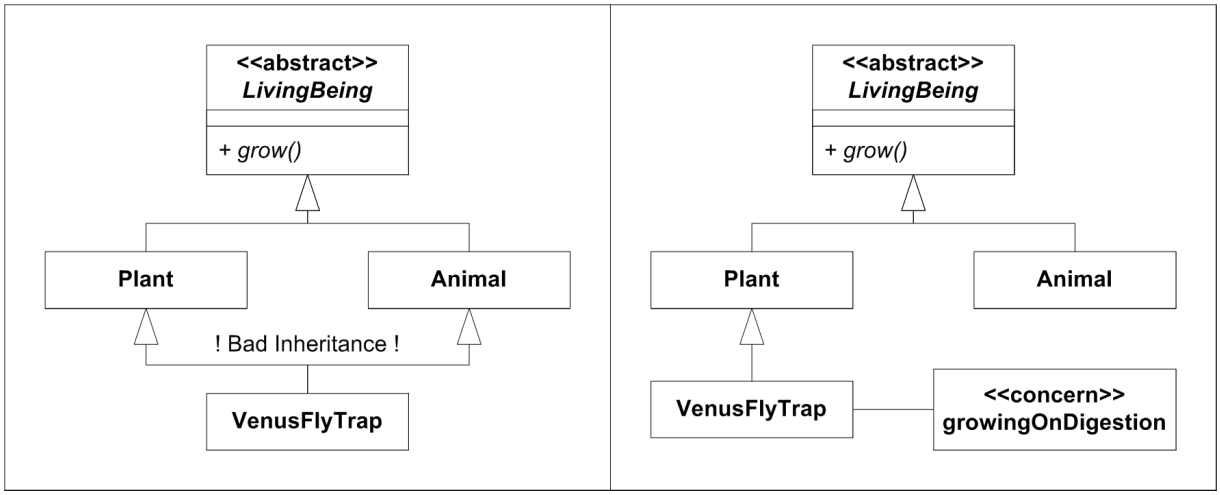


Abbildung 20: Composition Filters: Beispiel

Composition Filters folgen dem Konzept der *Aspektorientierten Programmierung*. Sie ermöglichen das modulare Hinzufügen und Kombinieren von Aspekten durch das Filtern von zwischen Objekten ausgetauschten Nachrichten. Diese Nachrichten werden zunächst daran gehindert ein Objekt zu erreichen oder zu verlassen. Sie werden überprüft, verändert bzw. umgeleitet, um infolgedessen das Verhalten eines Objektes zu adaptieren. Verschiedene Filtertypen spezifizieren den genauen Umgang mit den zu evaluierenden Nachrichten. Als sprachunabhängige Erweiterungen des objektorientierten Modells, reduzieren sie die Reimplementierung von Objektaufrufen und die Redefinition von Objektmethoden. Es gibt verschiedene *Composition Filter* Realisierungen; einige von ihnen unter anderem für *Java* und *.NET* wurden kurz vorgestellt. Mit ihnen lassen sich *Composition Filters* einfach und intuitiv implementieren. Allerdings sind die meisten vorhandenen Werkzeuge noch nicht ausgereift.

5.1.6 DemeterJ

Objektorientierte Programme setzen sich aus verschiedenen Klassen und ihren komplexen Beziehungen untereinander zusammen. Kommen neue Klassen hinzu oder ändern sich bestehende Beziehungen in einem Programm, dann müssen umfangreiche Änderungen am Programmcode vorgenommen werden. Außerdem enthält die Implementierung des Verhaltens eines Programms viele Redundanzen der Klassenstruktur. Das bedeutet, dass die Klassenstruktur und das Verhalten eines objektorientierten Programms sehr eng gekoppelt sind.

DemeterJ [16] versucht diese beiden Belange so weit wie möglich zu entkoppeln. *DemeterJ* ist ein Werkzeug das die *Adaptive Programmierung* als eine spezielle Form der *Aspektorientierten Programmierung* für die Programmiersprache *Java* unterstützt.

Die *Adaptive Programmierung* [17] befasst sich mit der Behandlung von wechselnden Belangen an ein Softwareprogramm. Dabei werden die Beziehungen zwischen Objektstruktur und Objektverhalten eines Softwareprogramms flexibel gehalten.

Ein adaptives Softwareprogramm kann sein Verhalten selbständig verändern, um auf Änderungen der Klassenstruktur zu reagieren. Da es potentiell eine unendliche Anzahl von möglichen Klassenstrukturen gibt, beschreibt ein adaptives Programm eine vollständige Familie von möglichen Programmen, die sich aus der Klassenstruktur ergeben können. (siehe Abbildung 21)

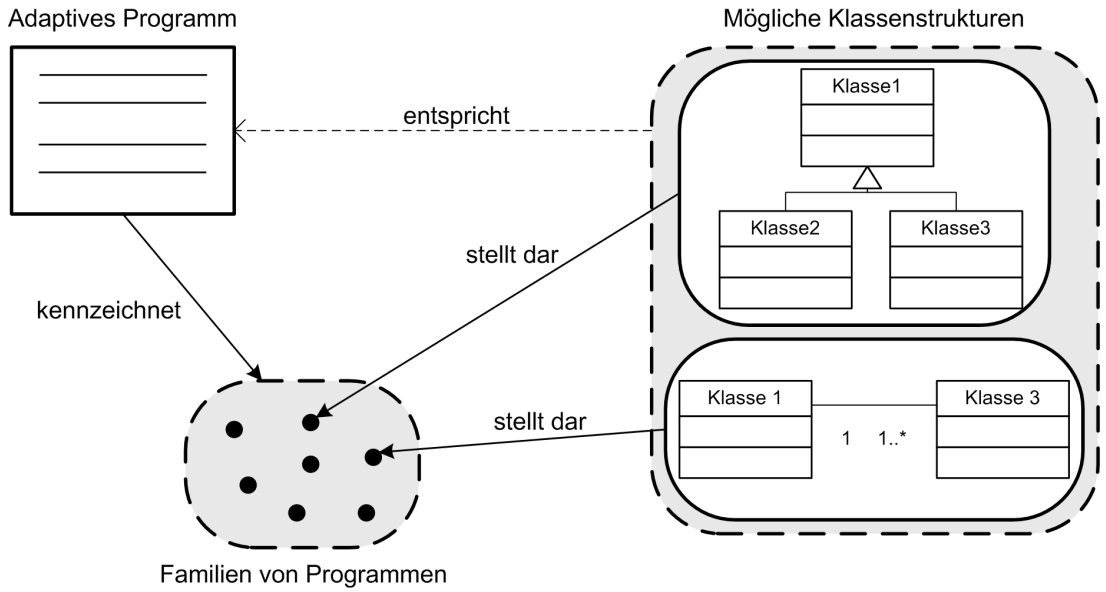


Abbildung 21: Adaptive Programmierung

Ein adaptives Programm wird durch sein Verhalten, seinem Klassengraphen (im Englischen *class graph*) , seinem Traversal-Graphen (im Englischen *traversal graph*) und einem Objekt, das den Graphen entlang wandert dem so genannten Besucher (im Englischen *visitor*), definiert. Der Klassengraph ist ein Model der Klassenstruktur. Er ähnelt stark den Klassendiagrammen von *UML*, enthält aber keine Klassendetails, wie z.B. Methodennamen, und muss nicht in grafischer Form vorliegen. Das Verhalten eines adaptiven Programms, die so genannte Traversal-Strategie (im Englischen *traversal strategie*), wird durch den Weg den der Besucher im Graphen durchwandert bestimmt. Der Weg des Besuchers wird durch gerichtete und ungerichtete Kanten im Traversal-Graphen definiert. Der Traversal-Graph ist ein Untergraph des Klassengraphen. Der Besucher führt Berechnungen durch, transportiert Daten und führt Transformationen durch auf seinem Weg durch den Traversal-Graphen. Die Kombination aus Besucher und Verhalten eines adaptiven Programms wird als adaptive Methode bezeichnet.

Die Anpassung der adaptiven Methode an einen speziellen Klassengraphen erfolgt bei *DemeterJ* zur Compilezeit, bei der automatischen Quellcode Generierung.

DemeterJ ist ein Präprozessor, der *Java*-Quellcode generiert, welcher später mit einem gängigen *Java*-Programm compiliert werden kann (siehe Abbildung 22). Dazu benötigt *DemeterJ* eine Datei mit der Beschreibung der Klassenstruktur (Klassengraph) und mehrere Dateien mit Beschreibungen des Verhaltens (Traversal-Strategie).

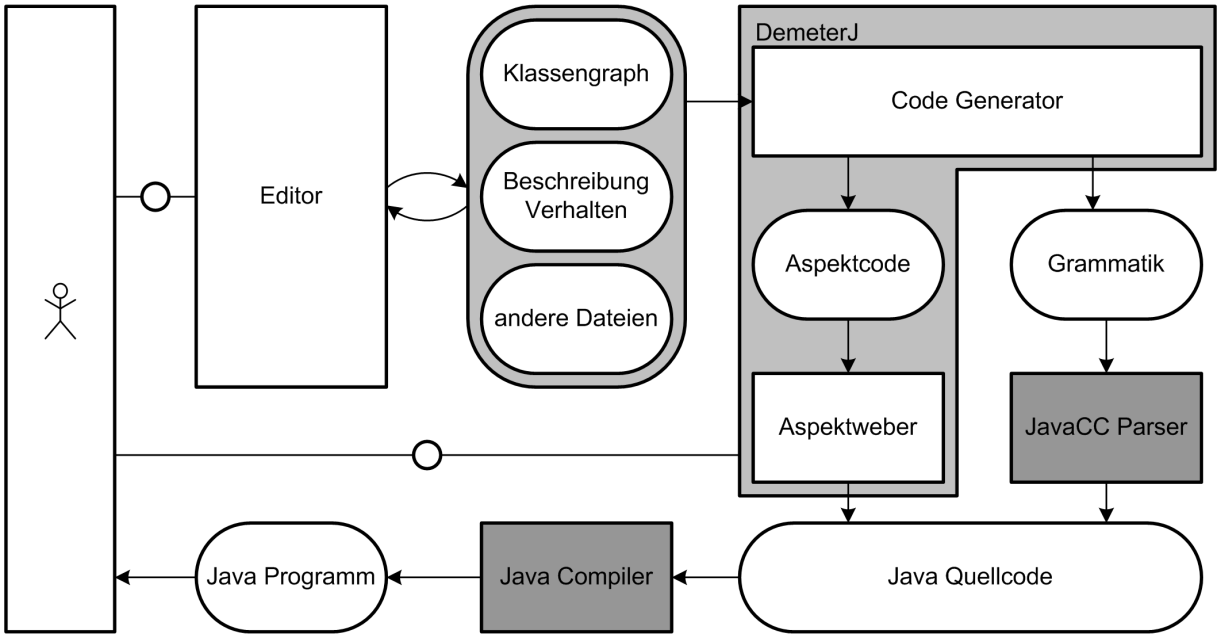


Abbildung 22: DemeterJ: Allgemeine Funktionsweise

Die Datei mit der Beschreibung der Klassenstruktur beschreibt textuell den Klassengraphen und somit die strukturellen Belange. Zusätzlich enthält diese Datei noch syntaktische Anweisungen, welche das automatische parsen und ausgeben von Objektstrukturen ermöglichen. Der Klassengraph kann mit Hilfe eines Texteditors oder mit dem grafischen Werkzeug *AP Studio*, welches ebenso wie *DemeterJ* selbst ihm Rahmen des *Demeter* Projektes entwickelt wurde, erzeugt werden.

Die Dateien mit den Beschreibungen des Verhaltens können Traversal-Strategien, Besucherklassen, adaptive Methoden und *Java*-Methoden enthalten. Eine adaptive Methode besteht aus einer Signatur, einer Traversal-Strategie und einem oder mehrere Besuchern. Die Traversal-Strategie beschreibt wohin im Klassengraphen navigiert werden soll und der Besucher was zu tun ist. Dazu stehen dem Besucher die drei verschiedene Typen von Methoden `before`, `after` und `around` zur Verfügung, die jeweils zu unterschiedlichen Zeitpunkten zur Ausführung kommen.

Anhand des Klassengraphen generiert *DemeterJ* eine äquivalente Struktur von *Java*-Klassen und eine Grammatik für den *JavaCC*-Parser. Die Methoden der *Java*-Klassen werden aus den Dateien mit den Beschreibungen des Verhaltens generiert. Die *Java*-Methoden werden einfach übernommen. Syntaktisch ist dies daran zu erkennen, dass reiner *Java*-Quellcode von zwei @ Zeichen umschlossen wird. *Deme-*

terJ kann automatisch einige allgemeine Methoden generieren, wie z.B. `get`, `set`, `parse` und `print`, und ebenso einige allgemeine Besucherklassen, wie z.B. `Copy`, `Display`, `Equal`, `Print` und `Trace`. Der generierte Aspektcode wird vom Aspektweber in *Java*-Quellcode transformiert. Die Grammatik dient dem *JavaCC*-Parser, um *Java*-Quellcode für einen anwendungsspezifischen Parser zu generieren.

DemeterJ ist Resultat des *Demeter* Projektes, das 1984 von Karl J. Lieberherr ins Leben gerufen wurde und anfangs das Ziel hatte ein Werkzeug zur Metaprogrammierung zu entwickeln. Seit 1996 richtete das *Demeter* Projekt seine Forschungsarbeit auf die Programmiersprache *Java*.

DemeterJ ist in der Version 0.8.6 für Windows und Linux verfügbar. Vor der Installation von *DemeterJ* muss zunächst der *Java Compiler Compiler* (kurz *JavaCC*) ab der Version 2.1 installiert werden. Bei *JavaCC* handelt es sich um einen Parser für *Java*.

Das grafische Werkzeug *AP Studio* zur Erzeugung von Klassengraphen ist ebenfalls in der Version 0.8.6 verfügbar.

Sowohl *DemeterJ* als auch *AP Studio* wurden seit Mitte 2003 nicht weiterentwickelt.

Ein einfaches Beispiel für einen Klassengraphen zeigt Abbildung 23. Dort ist die Klassenstruktur einer simplen Anwendung abgebildet, die es einem Unternehmen ermöglicht anhand des Namens eines Angestellten dessen Position auszugeben (Manager oder Angestellter).

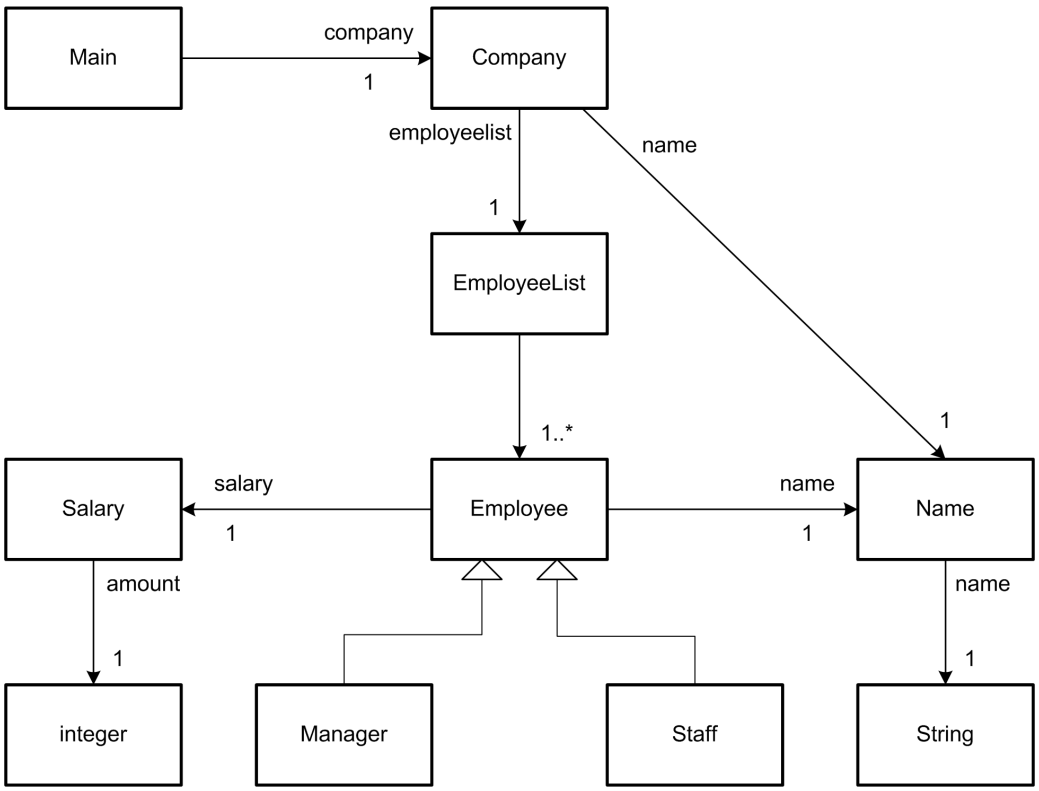


Abbildung 23: DemeterJ: Beispiel Klassengraph

Ein Beispiel für eine Traversal-Strategie für den abgebildeten Klassengraphen ist

das Folgende:

```
from Company to Manager, Staff
```

Der daraus resultierende Traversal-Graph ist in der Abbildung 24 zu sehen.

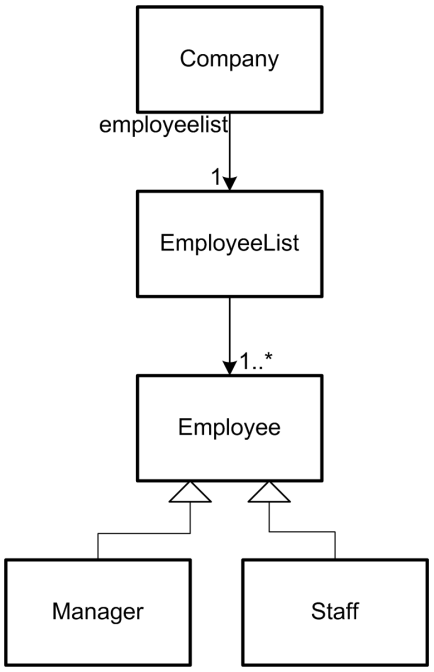


Abbildung 24: DemeterJ: Beispiel Traversal-Graph

Die Traversal-Strategie wird in der Dateien zur Beschreibung des Verhaltens eines adaptiven Programms in der folgenden Form definiert:

```
Company{
  traversal allEmployees(EmpsearchVisitor){
    to{ Manager, Staff };
  }
}
```

Diese Traversal-Strategie hat den Namen `allEmployees` und wird an die Klasse `Company` angehängt. Als Argument übernimmt die Traversal-Stragtegie ein `EmpSearchVisitor` Objekt mit dem sie den Klassengraphen durchwandert.

Ein Beispiel für einen Besucher ist `EmpSearchVisitor`. Der Besucher überprüft, ob der gesuchte Name eines Angestellten mit dem aktuell gefundenen übereinstimmt und gibt im Falle der Übereinstimmung dessen Position aus.

```
EmpSearchVisitor{
  before Manager (@ title = "Manager"; @)
  before Staff (@ title = "Staff"; @)
  after Employee (@
    if(host.get_name().equals(emplname)){
      System.out.println("Found employee: " + title);
      return_val = host;
    }
  @)
}
```

Die folgende adaptive Methode `allEmployees` verbindet die `allEmployees` Traversal-Strategie mit dem Besucher `EmpSearchVisitor`:

```
Company{
  public Employee searchForEmployee(Name emplname) =
      allEmployees(EmpSearchVisitor);
}
```

DemeterJ vermeidet die aufwendige manuelle Programmierung und dadurch auch Fehler bei der Programmierung durch die automatische Quellcode Generierung. Außerdem ermöglicht es *DemeterJ* Programmierern flexiblere Programme zu schreiben, deren Quellcode sich leichter an Änderungen der Programmstrukturen anpassen lässt.

5.1.7 HyperJ

HyperJ [18] basiert auf dem Hyperraummodell (im Englischen *hyperspace model*), das wiederum auf den Prinzipien der *Subjektorientierten Programmierung* beruht.

Die *Subjektorientierte Programmierung* ist der *Aspektorientierten Programmierung* sehr ähnlich. Sie ist ebenso wie die *Aspektorientierte Programmierung* eine Erweiterung der *Objektorientierten Programmierung*. Die Grundidee der *Subjektorientierten Programmierung* ist die Entkapselung von Anforderungen aus Softwaremodulen. Dafür müssen die Anforderungen identifiziert werden, ähnliche Anforderungen zu Gruppen zusammengefasst werden und diesen Gruppen ein Bezeichner zugewiesen werden. Als Resultat erhält man ein so genanntes Subjekt für jede Gruppe von Anforderungen. (siehe Abbbildung 25)

Subjekte können sich gegenseitig überlappen. In diesem Falle müssen die Subjekte zusammengefasst werden. Dies kann auf drei verschiedene Arten geschehen (siehe Abbildung 26):

Vermischen nur eines der beiden Objekte nimmt an der Relation teil.

Überschreiben eines der beiden Objekte wird verworfen.

Auswählen eines der beiden Objekte wird zur Laufzeit ausgewählt.

Zwischen den einzelnen Subjekten werden Kompositionsbeziehungen festgelegt. Anhand der Kompositionsregeln werden die Subjekte in das Gesamtsystem integriert.

Die *Subjektorientierte Programierung* löst nicht die Probleme die durch *code scattering* und *code tangling* entstehen. Vielmehr werden die Probleme nur auf eine tiefere Ebene, die der Subjekte, transferiert.

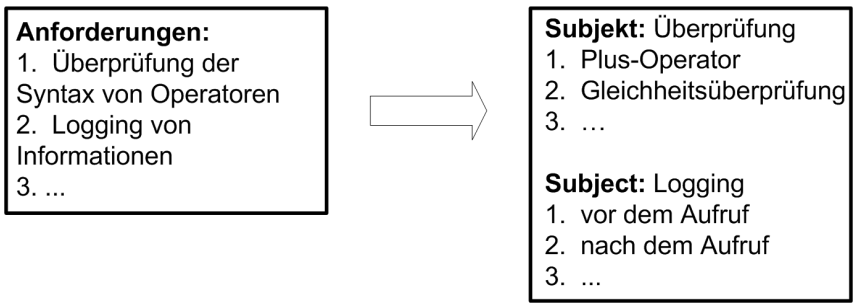


Abbildung 25: Subjektorientierte Programmierung: Spezifizierung von Subjekten

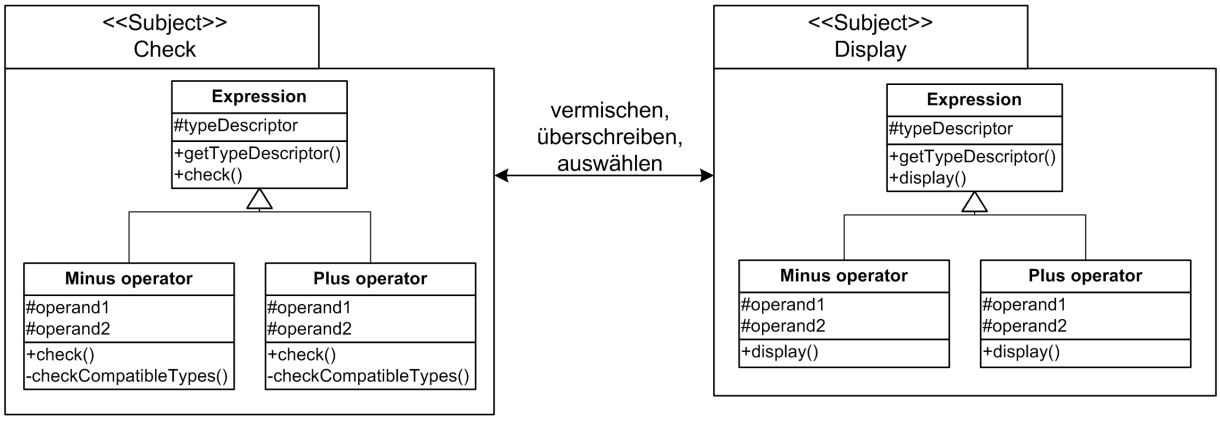


Abbildung 26: Subjektorientierte Programmierung: Subjekt Entwurf

Wie bereits erwähnt basiert *HyperJ* auf dem Hyperraummodell. Das Hyperraummodell definiert ein Softwareprogramm als eine unstrukturierte Sammlung von Einheit im Hyperraum. Diese Einheiten sind Softwarekomponenten, verschiedene Sichten auf das Softwareprogramm, d.h. Entwürfe, Implementierungsdetails, Dokumentationsdateien, und die Belange an das Softwareprogramm. Das Hyperraummodell identifiziert die Belange an ein Softwareprogramm und gruppiert die Einheiten mit denselben Belangen in Untermengen (im Englischen *hyperslice*) im Hyperraum. Die *hyperslices* entsprechen den Subjekten bei der *Subjektorientierten Programmierung.* Sie stellen die Dimensionen von Belangen an Softwaresysteme dar, im Sinne der mehrdimensionalen Modularisierung, und die deklarativen Einheiten für die Belange im späteren Softwareprogramm. Die *hypersclices* werden kombiniert und in Hypermodulen zusammengefasst. Dabei müssen Kompositionsregeln und Kompositionsbeziehungen berücksichtigt werden. Hypermodule sind *Java* Class-Dateien. Die Hypermodule werden von *HyperJ* mit den funktionalen *Java* Class-Dateien zum fertigen Softwareprogramm verwoben. (siehe Abbildung 27)

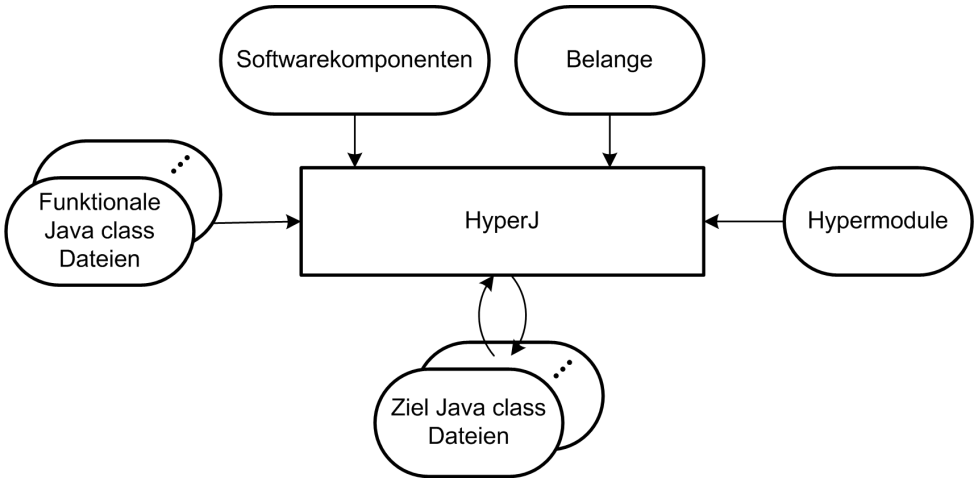


Abbildung 27: HyperJ

HyperJ wurde als Prototyp von H. Ossher und P. Tarr von ca. 1999 bis 2001 entwickelt. *HyperJ* ist in der Version 1.2 verfügbar, die hauptsächlich für Testzwecke geeignet ist. Es ist eine Werkzeug, das auf Binärdaten operiert und keinen Quellcode benötigt. Alle Erweiterungen der *Java* Class-Dateien werden in Textdateien festgehalten. *HyperJ* ist keine Spracherweiterung von *Java*. Zur Ausführung wird ein JDK in der Version 1.1.5 - 1.2.1 benötigt.

Ein Beispiel für eine Hyperraum ist `ebusiness`.

```
hyperspace ebusiness
  composable class supplier.*;
  composable class business.*;
  composable class customer.*;
  composable class ebusiness.*;
  composable class logging.*;
  composable class common.*;
  composable class security.*;
```

Die Belange für den Hyperraum sind die Folgenden:

```
package business : Feature.Core
package supplier : Feature.Core
package customer : Feature.Core
package ebusiness : Feature.Core
package common : Feature.Core
package logging : Feature.Logging
package security : Feature.Security
```

Das zugehörige Hypermodul ist `StandardBusinessWithLoggingAndSecurity`.

HyperJ wurde mit dem Ziel entwickelt, die Identifizierung, Kapselung und Manipulation von in *Java* implementierten Belangen zu erleichtern und flexibel zu ermöglichen. Allerdings bietet *HyperJ* keine Unterstützung für die Identifikation von Belangen. Es

sollte insbesondere Einsatz bei der Neuentwicklung von Softwaresystemen finden und zum *Reengineering* in der Nutzungsphase einer Software angewandt werden.

Ein weiteres Ziel war die Komposition und Integration von verschiedenen Belangen an ein Softwaresystem zu ermöglichen. Dabei sollten Belange zu Komponenten und Systemen zusammengefügt werden können und weitere Belange ohne Quellcodeanpassung als Ergänzungen mit aufgenommen werden können. Problematisch dabei ist jedoch das Erstellen der notwendigen Kompositionsregeln ohne Kenntnis des Quellcodes. Außerdem sollte die Art und Weise der Modularisierung nachträglich änderbar sein.

```
hypermodule StandardBusinessWithLoggingAndSecurity

  hyperslices:
    Feature.Security,
    Feature.Logging,
    Feature.Core;

  relationships:
    mergeByName;

    equate class Feature.Core.RealCustomerBusiness,
          Feature.Security.SecureBusinessAPI;
    order action Feature.Security.SecureBusinessAPI.bl_InsertData
          before action Feature.Core.RealCustomerBusiness.
             bl_InsertData;

    bracket "Real*"."bl_*" with
      before Feature.Logging.LoggedClass.beforeCall
            ($ClassName, $OperationName ),
      after Feature.Logging.LoggedClass.afterCall
            ($ClassName, $OperationName );

end hypermodule;
```

5.2 Dynamische Aspektweber

Dynamische Aspektweber verweben Aspektcode mit funktionalem Bytecode dynamisch zur Laufzeit. (siehe 2.3)

Im Folgenden werden zunächst die Konzepte beleuchtet auf denen die dynamischen Aspektweber beruhen und anschließende einige dynamische Aspektweber vorgestellt.

5.2.1 Ansätze zur dynamischen Verwebung von Aspekten

Es existieren vier verschiedene Ansätze zur dynamischen Verwebung von Aspekten. Alle Ansätze zielen auf die Verwebung von Aspektcode mit funktionalem Programmcode zur Laufzeit ab. Die vier Ansätze sind im Folgenden beschrieben:

- auf *Wrapper* basierender Ansatz,
- Bytecode Transformation zur Ladezeit,
- Instrumentierung der *Just-In-Time* Schicht und
- Dynamische Code Instrumentierung.

Zur dynamischen Verwebung von Aspekten sind genaueste Informationen über die Aspekte und die zu verwebenden Objekte zur Laufzeit erforderlich. Die Aspekte werden an vorher definierten Verwebungspunkten (siehe Kapitel 2.2.1) im Programmcode mit den Objekten des Programms verwoben. Nach der Verwebung sind die Aspekte Teil der Objekte.

Der auf *Wrappern* basierende Ansatz, auch Proxy genannt, wird im *Framework* und im *Application Programming Interface* (kurz *API*) Programmierumfeld eingesetzt zur dynamischen Verwebung von Aspektcode mit Objektcode.
Für jedes Objekt wird ein *Wrapper*-Objekt bzw. Proxy implementiert. Dieser *Wrapper* ist für die Ausführung des Aspektcodes verantwortlich. Das *Wrapper*-Objekt wird vor das zu verwebene Objekt platziert und alle Aufrufe an bzw. Referenzen auf das Objekt werden über den Proxy umgeleitet.

Bei der Bytecode Transformation werden die Verwebungspunkte bzw. der Aspektcode in die geladenen Klassen zur Ladezeit eingewoben. Dazu wird der *Class-Loader* ersetzt. Der ersetzte *Class-Loader* verändert die Implementierung der Klassen und Interfaces, indem er die Verwebungspunkte bzw. den Aspektcode einwebt.

Bei der Instrumentierung der *Just-In-Time* (kurz *JIT*) Schicht müssen der *JIT*-Compiler und die Virtuelle Maschine angepasst werden (siehe Abbildung 28). Nach der Anpassung besitzt der Ausführungsmonitor einen Verwalter für die Verwebungspunkte und stellt die entsprechende *API* bereit. Der Ausführungsmonitor aktiviert einen Verwebungspunkt dynamisch auf Anfrage der *AOP-Engine*. Wird nun zur Laufzeit ein aktiver Verwebungspunkt erreicht, dann informiert der Ausführungsmonitor die *AOP-Engine* darüber, welche dann den Aspektcode ausführt.

Bei der dynamische Code Instrumentierung wird der Bytecode im Speicher verändert. Dazu wird ein bedingungsloser Sprung als Verwebungspunkt in den Bytecode eingefügt, wobei die Programmausführung angehalten werden muss (siehe Abbildung 29). Zunächst müssen die aktuellen Registerinhalte gesichert werden. Danach kann der einzuwebende Aspektcode ausgeführt werden. Anschließend werden die Registerinhalte wiederhergestellt und die Programmausführung wird fortgesetzt.

5.2.2 Rapier-LOOM.NET

Rapier-LOOM.NET ist ein dynamischer Aspektweber (siehe Kapitel 2.3) für das *.Net Framework*. Die Grundidee ist die explizite Beschreibung von Verwebungspunkten (siehe Kapitel 2.2.1) mithilfe von Attributen. Da Aspekte in *Rapier-LOOM.NET* auch gleich-

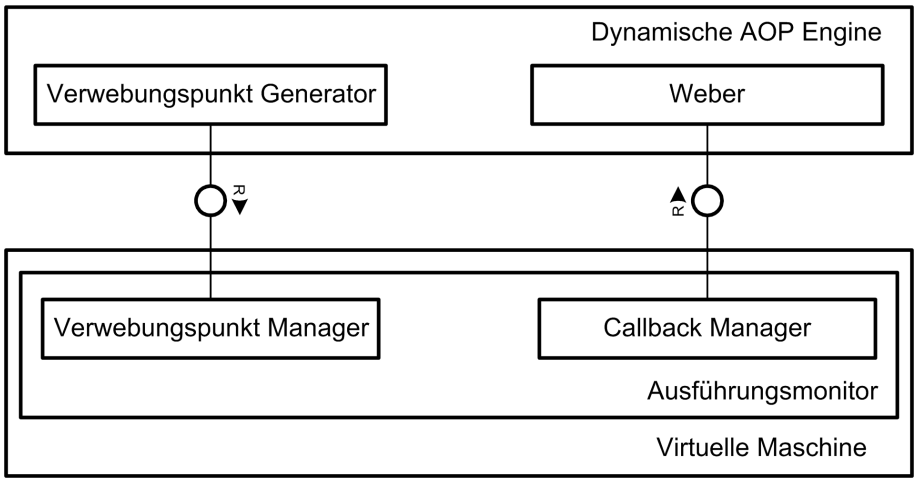


Abbildung 28: Dynamische Verwebung von Aspekten: Instrumentierung der JIT-Schicht

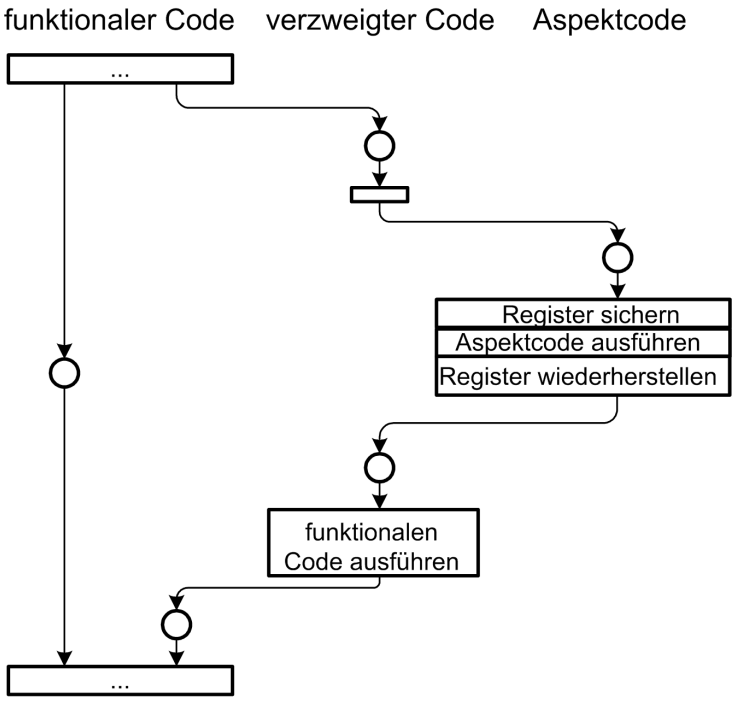


Abbildung 29: Dynamische Verwebung von Aspekten: Dynamische Code Instrumentierung

zeitig Attribute sind, können diese auch Elemente wie Klassen, Methoden, Module oder Assemblies attribuieren. Dies wiederum bewirkt, dass der Aspektweber den Aspektcode in die entsprechenden Elemente einwebt.

Wird zum Beispiel eine Klasse `A` mit dem Aspekt `Aspekt` attribuiert, so bedeutet dies, dass die gesamte Klasse mit diesem Aspekt verwoben wird. Das heißt, dass jede einzelne Methode in dieser Klasse nach der Verwebung Aspektcode aus `Aspekt` enthalten kann. Würde der Aspekt nur an eine einzelne Methode der Klasse `A` geschrieben, so wird auch explizit diese Methode mit dem Aspekt verwoben. Ähnlich verhält es sich mit Assemblies und Modulen: Hier werden alle Klassen in der Assembly bzw. in dem Modul mit dem Aspekt verwoben. Folglich können alle Methoden in der Assembly bzw. in dem Modul dann Aspektcode enthalten.

Ob eine Methode dann tatsächlich mit Aspektcode verwoben wird, hängt von der Definition der Aspektmethoden (*Advices*) im Aspekt ab. Sie werden durch ein Attribut (`Call` oder `Create`) definiert. Dieses Attribut bestimmt auch wo der Aspectcode dieser Methode eingewoben werden soll:

- vor der Zielmethode (`Before`),
- anstatt (`Instead`) oder
- danach (`After, AfterReturning, AfterThrowing`).

Stimmen Signatur von Aspektmethode und Zielmethode überein, findet eine Verwebung statt. Aspektmethoden können aber auch Wildcards in ihrer Signatur enthalten und mit zusätzlichen Verwebungsattributen markiert sein. Mögliche Verwebungsattribute sind:

- `Include`,
- `Exclude`,
- `IncludeIfAttributeDefined`,
- `ExcludeIfAttributeDefined`,
- `IncludeAll` und
- `DeclaredOnly`.

Mithilfe der Verwebungsattribute wird eine Menge von Eigenschaften definiert, die erfüllt sein müssen, damit eine Aspektmethode mit einer Zielmethode verwoben wird. Auf diese Weise kann eine Aspektmethode gleichzeitig mit verschiedenen Methoden einer Zielklasse verwoben sein.

Neben der Verwebung von Methoden kann *Rapier-LOOM.NET* zusätzlich auch neue Interfaces in Zielklassen einweben. Diese sogenannten *Introductions* werden auch als Attribut (`Introduces`) an der Aspektklasse markiert. Interfaces die als *Introduction* markiert wurden, müssen in der Aspektklasse definiert werden und sind nach der Verwebung in der Zielklasse zugänglich.

Die Verwebung der Aspekte erfolgt zur Laufzeit. Alle zu verwebenden Klassen müssen mit einer speziellen *Factory*-Methode `Create<T>` erstellt werden. *Rapier-LOOM.NET* sucht daraufhin nach allen Aspekten mit denen diese Klasse, das umgebende Modul und die umgebende Assembly annotiert wurden und verwebt diese. Zusätzlich können aber auch dynamisch der *Factory* weitere Aspektinstanzen zur Verwebung übergeben werden. Das von der *Factory* erzeugte Objekt ist ein von der zu erzeugenden Klasse abgeleitetes Objekt, welches den verwobenen Aspektcode enthält.

Rapier-LOOM.NET wurde am Hasso-Plattner-Institut für Softwaresystemtechnik GmbH im Rahmen des LOOM.NET Projektes entwickelt. Der Aspektweber ist in der Version 1.5 Beta 2 als Bibliothek verfügbar.

Das nachfolgende Beispiel demonstriert die Arbeitsweise von *Rapier-LOOM.NET*. Der Aspekt `MyAspect` enthält zwei Aspektmethoden `MatchOne` und `MatchAll`. `MatchAll` enthält ein zusätzliches Verwebungsattribut `IncludeAll` und die Wildcards `object` und `object[]` in der Parametersignatur.

```
using Loom;
using Loom.ConnectionPoint;

public class MyAspect{
  [IncludeAll]
  [Call(Invoke.Before)]
  public object MatchAll(object[] args) {}

  [Call(Invoke.After)]
  public void MatchOne() {}
}
```

Angewendet auf die Klasse `MyClass` (durch die Verwendung als Attribut) wird `MatchOne` genau mit der gleichnamigen Methode verwoben, wohingegen `MatchAll` mit allen Methoden verwoben wird.

```
[MyAspect]
public class MyClass{
  public virtual void MatchOne() {}
  public virtual void foo() {}
  public virtual int bar(string s) {}
}
```

Die Verwebung erfolgt bei der Instanzierung der Zielklasse.

```
MyClass obj=Weaver.Create<MyClass>();
```

Rapier-LOOM.NET kann in allen *.Net* Sprachen für die Verwebung von Aspekten benutzt werden. Einschränkungen von *Rapier-LOOM.NET* sind neben der Verwendung der *Factory*-Methode (anstelle des `new`-Operators), das nur Methoden und Pro-

perties verwoben werden können, die über ein Interface definiert wurden oder virtuell sind.

5.2.3 AspectS - Aspektorientierte Programmierung mit Squeak

AspectS ist eine Erweiterung der *Smalltalk*-Umgebung *Squeak* zur Unterstützung aspektorientierter Softwareentwicklung. *AspectS* wird seit 2002 von Robert Hirschfeld entwickelt.

Smalltalk wurde als erste objektorientierte Sprache bekannt und basiert auf wenigen grundlegenden Konzepten. Alles in *Smalltalk* ist ein Objekt. Objekte werden durch die Zusendung von Nachrichten aktiviert (Methodenaufruf). Objekte können an Variablen zugewiesen werden und letztendlich können Objekte von Methoden zurückgegeben werden. Mit Hilfe dieser Konzepte können sämtliche weitere Programmierkonstrukte (Schleifen, Verzweigungen ...) abgeleitet werden.

In *AspectS* werden Aspekte als reguläre *Smalltalk*-Klassen implementiert. Ein Aspekt wird installiert, indem eine `install` Nachricht an eine Aspektinstanz gesandt wird. Dabei können potentiell alle Objekte eines Images verwoben werden.

Join-Points markieren wohldefinierte Punkte im Anwendungscode. Sie werden in *AspectS* durch die Benennung einer Zielklasse und eines Selektors (Methodenname) beschrieben. Wie auch schon in *AspectJ* (siehe Kapitel 5.1.1) werden auch in *AspectS* *advices* benutzt um Aspektcodefragmente mit den *Join-Points* zu verbinden.

AspectS unterstützt die Verwebung von Aspektcode:

- vor und nach der Ausführung einer Methode (AsBeforeAfterAdvice),
- zur Behandlung von strukturierten Ausnahmen (AsHandlerAdvice) und
- anstatt der Ausführung einer Methode (AsAroundAdvice).

Advices können weiterhin durch Eigenschaften des Ruferkontexts einer Klasse/Instanz verfeinert werden werden (cflow). Das Class-Specific-First *advice* untersucht beim Test der Aspektaktivierung, ob auf dem Call-Stack einer gerufenen Methode dieselbe Klasse vorhanden ist. Class-Specific - All-But-First untersucht den Call-Stack auf das Vorhandensein mehr als eines Vorkommens derselben Klasse. Zusätzlich gelten Instanz-spezifische *advices* für Instanzen, dass heißt der Call-Stack wird auf Vorhandensein einer Instanzreferenz untersucht.

Zusätzlich unterstützt *AspectS* die Erweiterung existierender Klassen um neue Funktionalität (introductions).

Die Verwebung von Aspektcode mit dem funktionalen Anwendungscode erfolgt in *AspectS* während der Laufzeit. Mit Hilfe von Method-Wrappern kann zusätzlicher Code in die Programmausführung integriert werden. Method-Wrapper manipulieren die Verarbeitung (lookup) von Methodenaufrufen, indem die Methodentabelle einer Klasse angepasst wird.

5.2.4 JBoss Aspect Oriented Programming

Das *JBoss AOP* Framework erlaubt es aspektorientierte Lösungen für *Java* zu entwickeln. Es lässt sich nahtlos in den *JBoss Application Server* integrieren, funktioniert aber auch ohne ihn [19], indem das Framework hierfür eine Klassenbibliothek bereitstellt. Die Konzepte der *AOP*-Welt sind auch bei *JBoss* dieselben. Nachdem die *cross-cutting concerns* (siehe Kapitel 2.2) identifiziert wurden oder andere Ziele die mit Hilfe von *AOP* erreicht werden sollen ausgemacht sind, werden die jeweiligen Aspekte definiert und implementiert. Danach werden die *pointcuts* in der Zielimplementierung definiert, an welche die Aspekte gebunden werden sollen. Diese *pointcuts* können grundsätzlich Konstruktoren, Methoden und Felder sein.

Aspekte des *JBoss AOP* sind einfache simple Java Klassen. Jede Methode innerhalb einer Klasse, in der Form

```
Object methodName(Invocation invocation) throws Throwable
```

kann an einen *pointcut* gebunden werden. Zusätzlich gibt es die Möglichkeit, nach dem Modell des *Interceptor Patterns* [20], *Interceptors* zum Intervenieren zu erstellen. *Interceptors* implementieren allerdings prinzipiell nur eine Methode der Form

```
Object invoke(Invocation invocation)
```

zur Realisierung des jeweiligen Aspekts.

Ein Beispiel für einen Aspekt ist:

```
public class MyAspect{
  public Object traceMethod(MethodInvocation invocation) throws
     Throwable{
    try {
      System.out.println("Stepping into method " + invocation.
         getMethod().toString());
      return invocation.invokeNext();
    }
    finally {
      System.out.println("Exiting method " + invocation.getMethod
         ().toString());
    }
  }
  ...
}
```

Je nach Bindung der *pointcuts*, können verschiedene *invocation class types* wie z.B. *ConstructorInvocation*, *MethodInvocation*, *FieldInvocation* (Lese-/Schreiboperation separat) und auch so genannte *caller pointcuts* genutzt werden, welche Zugriff auf aufrufende und aufgerufene Objekte ermöglichen. Nachdem die Aspekte bzw. *Interceptors* definiert sind, müssen die Bindungen für die *pointcuts* festgelegt werden. Hierfür gibt es zwei Möglichkeiten. Die Bindung kann in einer *XML* Datei konfiguriert werden oder mittels der *@Aspect* Notation direkt im Quelltext festgelegt werden. Seit *Java* 1.5 (*JDK5.0*) können Metadaten mittels *Annotations* [21] an beliebigen Stellen im Quelltext

hinzugefügt werden.

Ein Beispiel für die Bindung eines Aspektes mittels einer *XML* Datei ist:

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <aspect class="MyAspect" scope="PER_VM"/>
  <bind pointcut="execution(POJO->someMethod())">
    <advice name="traceMethod" aspect="MyAspect"/>
  </bind>
</aop>
```

Diverse Patterns und Wildcards (`all(POJO*->*())`) helfen bei der Festlegung der Bindung. Verschiedene Scopes (`PER_VM, PER_CLASS, ...`) helfen die Objekt-Instanzierungen zu steuern, um ggf. neue Aspekt-Instanzen zu erzeugen oder bereits existierende Aspekt-Instanzen zu verwenden.

Mit der neuen Version wurden zusätzlich neue Bibliotheken zur Modifizierung von Metadaten von Methoden, Felder und Konstruktoren bereitgestellt. Hierdurch ist es nun möglich Kontextinformationen einfacher zwischen verschiedenen Aspekten zu übermitteln. Auch *Introductions* unterstützt das Framework, bezeichnet diese aber weiterhin *MixIns*.

Ein Beispiel für die Bindung eines Aspektes mittels *Annotations* ist:

```
@Aspect (scope = Scope.PER_VM)
public class MyAspect{
  @PointcutDef ("(execution(POJO->someMethod())")
  public Object traceMethod(MethodInvocation invocation) throws
     Throwable{
    ...
  }
  ...
}
```

Das Framework bietet drei Möglichkeiten, um Aspekte an eine Anwendung zu binden. Der *precompiled mode* ist der einfachste Weg, bei dem nach dem Compilieren der eigentlichen Anwendung mittels des *AOP* Compilers die benötigten *joint-points* in den *Java* Bytecode eingeflochten werden (Bytecode-Weben siehe Kapitel 2.3). Beim Ladezeit-Weben werden die benötigten *join-points* erst mit dem Laden der jeweiligen *class*-Datei modifiziert. Dies verlangsamt zwar den Ladeprozess, bietet aber eine flexiblere Möglichkeit die Bindungen durch einfaches Austauschen der Aspektklassen und *XML* Konfigurationen zu verändern. Die Anwendung kann so ohne das eine Neucompilierung erforderlich ist zur Laufzeit mit neuen Aspekten versehen werden. Es ist aber auch möglich Bindungen von Aspekten und *Interceptors* an *pointcuts* zur Laufzeit zu verwalten. Der `AspectManager` erlaubt es Bindungen dynamisch hinzuzufügen und zu entfernen, indem er die *join-points* zur Laufzeit modifiziert. Im so genannten *hot-swap mode* instruiert der *AOP* Compiler zunächst alle *class*-Dateien mit einem Minimum an benötigten Informationen, damit dieser dann zur Laufzeit an den *joint-points* die Bytecode-Manipulation durchführen kann. Hierfür muss der *AOP* Compiler mit der

prepare Funktionalität Vorbereitungen an den beteiligten *class*-Dateien treffen. Diese Bytecode-Manipulationen scheinen immer noch mit dem Open Source Tool *Javassist* [22] realisiert zu werden, welches sich die *JBoss Inc.* mittlerweile als eigenständiges Projekt einverleibt hat [23].

Ein Beispiel für ein einfaches *per instance binding* zur Laufzeit ist:

```
Advised advised = (Advised)pojo;
advised._getInstanceAdvisor().insertInterceptor(new
   InstanceInterceptor());
}
```

Jede durch den *AOP* Compiler vorbereitete Klasse im *hot-swap mode* implementiert `org.jboss.aop.Advised`. Durch das Hinzufügen eines neuen *Interceptors* (im Beispiel ein `InstanceInterceptor`), wird der *join-point* modifiziert und hierbei in jede Aufrufkette jedes Feldes, jeder Methode und des Konstruktors der betreffenden Klasse eingefügt. Dies bedeutet, dass eventuell im gebunden Aspekt geprüft werden muss, ob der jeweilige Aufruf überhaupt unterbrochen werden soll.

Ein Beispiel für dynamisches Binden durch den `AspectManager` ist:

```
org.jboss.aop.advice.AdviceBinding binding = new AdviceBinding("
   execution(POJO->new(..))", null);
binding.addInterceptor(SimpleInterceptor.class);
AspectManager.instance().addBinding(binding);
}
```

Der `AspectManager` dagegen erlaubt es mit einem `AdviceBindung` analog zum statischen Binden von Aspekten, das differenzierte Festlegen von *pointcut* Bedingungen und ermöglicht es so nahezu jede Bindung zur Laufzeit festzulegen und ebenfalls auch wieder zu lösen.

Die *JBoss Inc.* stellt ihr *AOP* Framework nun in der Version 1.5 bereit und propagiert es erneut als „100% pure *Java* aspect oriented framework“ [19]. *JBoss* möchte sich von anderen *AOP* Implementierungen mit eigener Syntax wie *AspectJ* (siehe Kapitel 5.1.1) abgrenzen und setzt weiterhin auf *Java*-eigene Syntax (*plain old java objects*).

5.2.5 Java Aspect Components

Java Aspect Components (kurz *JAC*) ist ein Framework zur *Aspektorientierten Programmierung* mit *Java* im *J2EE* Umfeld. *JAC* verfolgt den auf *Wrappern* basierenden Ansatz zur dynamische Verwebung von Aspekten (siehe Kapitel 5.2.1).

Ein Aspekt in *JAC* wird durch eine Aspektkomponente und einen *Wrapper* definiert. Die Aspekte werden im Gegensatz zu denen in *AspectJ* ausschließlich in *Java* implementiert.

Die Aspektkomponente definiert die Verwebungspunkte. Sie muss von der Klasse `org.objectweb.jac.core.AspectComponent` abgeleitet sein. *JAC* bietet für die im Folgenden aufgezählten Bereiche bereits vordefinierte und konfigurierbare Aspektkompo-

nenten an:

- GUI,
- Persistenz,
- Transaktionen,
- Verteilung,
- Monitoring,
- Ergebnisspeicherung,
- Datenintegrität und
- Synchronisation.

Natürlich können auch neue Aspekte definiert werden.

Die Verwebungspunkte werden durch die Aspektmethoden der Aspektkomponente, den so genannten *pointcuts*, definiert. Die *pointcuts* ähneln denen in *AspectJ* (siehe Kapitel 5.1.1). Die Parameter eines *pointcut* sind die Folgenden:

- Objektname,
- Klassenname,
- Methodenname,
- Wrapperklasse und
- Wrappermethode.

Der eigentliche Aspektcode wird in einem *Wrapper* definiert. *Wrapper* sind den *advices* (siehe Kapitel 5.1.1) in *AspectJ* sehr ähnlich. Ein *Wrapper* muss von der Klasse `org.objectweb.jac.core.Wrapper` abgeleitet sein. Er kann unabhängig von dem Aspekt definiert werden der ihn später benutzen soll. Es können mehrere *Wrapper* für einen *pointcut* definiert werden. Dadurch können *Wrapper*-Ketten gebildet werden.

Durch die Bildung von *Wrapper*-Ketten bzw. der Verwebung von mehreren Aspekten mit einem Objekt zum selben Zeitpunkt kann es zu Kompatibilitätsproblemen oder Problemen bei der Verwebungsreihenfolge kommen. Diese werden durch den *Wrapper*-Kontroller behoben. Er ist Teil von *JAC*. Der *Wrapper*-Kontroller führt zur Laufzeit kontextabhängige Tests durch. Anhand dieser entscheidet er, ob ein *Wrapper* ausgeführt wird und in welcher Reihenfolge die *Wrapper* ausgeführt werden.

In der Abbildung 30 ist die Funktionsweise von *JAC* dargestellt. Es wird gezeigt, wie die einzelnen mit Hilfe von *JAC* und die rein in *Java* implementierten Komponenten miteinander interagieren. Die funktionalen *Java* Class-Dateien werden zur Laufzeit geladen. Dabei wird ihr Bytecode modifiziert, um sie später mit einem oder mehreren *Wrapper* verweben zu können. Die Modifizierung des Bytecodes erfolgt mit Hilfe der *Byte Code Engineering Library* (*BCEL*), die im Kapitel 5.2.6 näher erläutert wird. Die

Aspektkomponenten werden mit Hilfe ihrer Konfigurationsdatei erzeugt und konfiguriert. Sie erzeugen auch die *pointcuts*, die für die *Wrapper* die Einstiegsstelle in einer Anwendung kennzeichnen. Die *Wrapper* werden von den *Java* Klassen als Metadaten benutzt, um verwebte Objekte von sich zu erstellen.

Mit Hilfe des in *JAC* integrierten Werkzeugs *UML Aspectual Factory* (kurz *UMLAF*) wird das aspektorientierte Design mit *UML* ermöglicht.

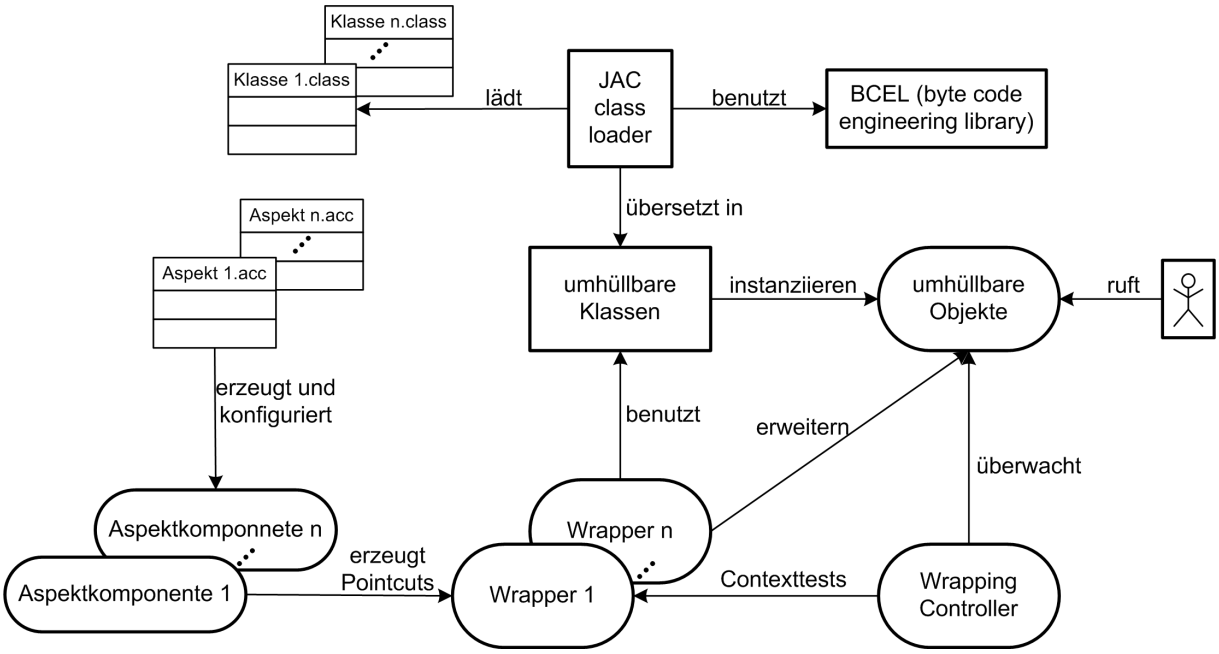


Abbildung 30: JAC: Funktionsweise

JAC ist als *Open-Source* Software frei verfügbar unter der *GNU Lesser General Public License*. *JAC* ist Betriebssystem unabhängig und in der Version 0.12.1 verfügbar. Das in *JAC* integrierte Werkzeug *UMLAF* ist als Beta Version enthalten.

Ein einfaches Beispiel für eine funktionale Klasse, die Addition und Subtraktion ermöglicht, ist das folgende:

```
public class Calculus{
  double value = 0;

  public void add(double toadd){
    value += toadd;
  }

  public void sub(double tosub){
    value -= tosub;
  }
}
```

Die in diesem Beispiel dazu gehörende Aspektkomponente mit einem *pointcut* ist

folgende:

```
public class TestAC extends AspectComponent{
  public TestAC(){
    pointcut("Calculus", "sub.*", LimiterWrapper.class, "limit",
       false);
  }
}
```

Der *pointcut* kennzeichnet den Verwebungspunkt in der Anwendung, bei dessen Aufruf der Aspektcode im *Wrapper* ausgeführt wird. In diesem Beispiel wird bei jedem Aufruf der Methode `sub` der Klasse `Calculus` die Methode `limit` der *Wrapper* Klasse `LimiterWrapper` ausgeführt. Der Aspektcode für dieses Beispiel ist im folgenden *Wrapper* implementiert:

```
public class LimiterWrapper extends Wrapper{
  public void limit(Interaction i){
    if(((Double)getFieldValue(i.wrappee, "value")).doubleValue() -
       ((Double)i.args[0]).doubleValue()) < 0)
          throw new Exception("<0 forbidden");
          proceed(i);
  }
}
```

Der *Wrapper* `LimiterWrapper` umhüllt die Verwebungspunkte mit dem Code seiner Methode `limit`. Das bedeutet, dass bei jedem Aufruf der Methode `sub` der Klasse `Calculus` immer zuerst der Code der *Wrapper* Methode `limit` ausgeführt wird. Der Übergabeparameter `i` der Methode `limit` verweist auf den aktuellen Verwebungspunkt. Die Methode `limit` testet, ob die Subtraktion mit dem Parameter `i.args[0]`, also dem Übergabeparameter `tosub` der Methode `sub` der Klasse `Calculus`, vom aktuellen Wert einen negativen Wert ergeben würde. Wenn dies der Fall ist, dann wird eine Fehlermeldung ausgegeben und die Methode `sub` nicht ausgeführt. Schlägt der Test jedoch fehl, dann wir die Methode `sub` ausgeführt.

JAC bringt im Gegensatz zu vielen anderen *AOP* Werkzeugen bereits mehrere vordefinierte und sofort benutzbare Aspekte mit. Außerdem ist es mit *JAC* möglich verteilt und aspektorientiert zu programmieren. *JAC* zeigt, dass man für *Aspektorientierte Programmierung* mit *Java* keine zusätzliche Syntax benötigt.

5.2.6 JMangler

JMangler ist ein Kommandozeilenwerkzeug für die Programmiersprache *Java*. Es basiert auf dem Ansatz der Bytecode Transformation zur Ladezeit (siehe Kapitel 5.2.1).

JMangler webt Verwebungspunkte bzw. Aspektcode in *Java* Class-Dateien ein, indem es die Methode `java.lang.ClassLoader.defineClass` manipuliert. Ein grober Aufbau von *JMangler* bzw. Ablauf der Verwebung mit *JMangler* ist in Abbildung 31 dargestellt.

JMangler benutzt die *ByteCode Engeneering Library* (kurz *BCEL*) zur Verwebung.

BCEL ist ein Framework zur Analyse, Generierung und Manipulation von *Java* Class-Dateien. *BCEL* besitzt eine umfangreiche *API*. Diese *API* bietet viele Klassen um alle Elemente einer Class-Datei zu erstellen.

Es gibt zwei Arten von Class-Datei Transformatoren. Zum einen die Interface Transformatoren und zum anderen die Code Transformatoren. Die Transformatoren sind *Java* Klassen, die die Interfaces `InterfaceTransformerComponent` bzw. `CodeTransformerComponent` implementieren.

Die folgenden Interface Transformationen sind möglich:

- Hinzufügen von Klassen, Interfaces, Methoden und Feldern,
- Änderungen der `extends` Spezifikation, solange die Anzahl der direkten und indirekten Superklassen nicht verringert wird und
- Änderungen der `implements` Spezifikation, solange die Anzahl der direkten und indirekten Superinterfaces nicht verringert wird.

Die Interface Transformationen werden nach folgenden Ablauf umgesetzt:

1. Jeder Transformator erstellt eine Liste der vorzunehmenden Änderungen.
2. Der *BCEL* Verweber führt diese Änderungen nacheinander aus.
3. Beide Schritte werden solange wiederhohlt, bis keine Änderungen mehr auftreten.

Die Code Transformatoren modifizieren den Code selbst unter Verwendung der *BCEL API*.

Aspekte werden in einer Konfigurationsdatei registriert.

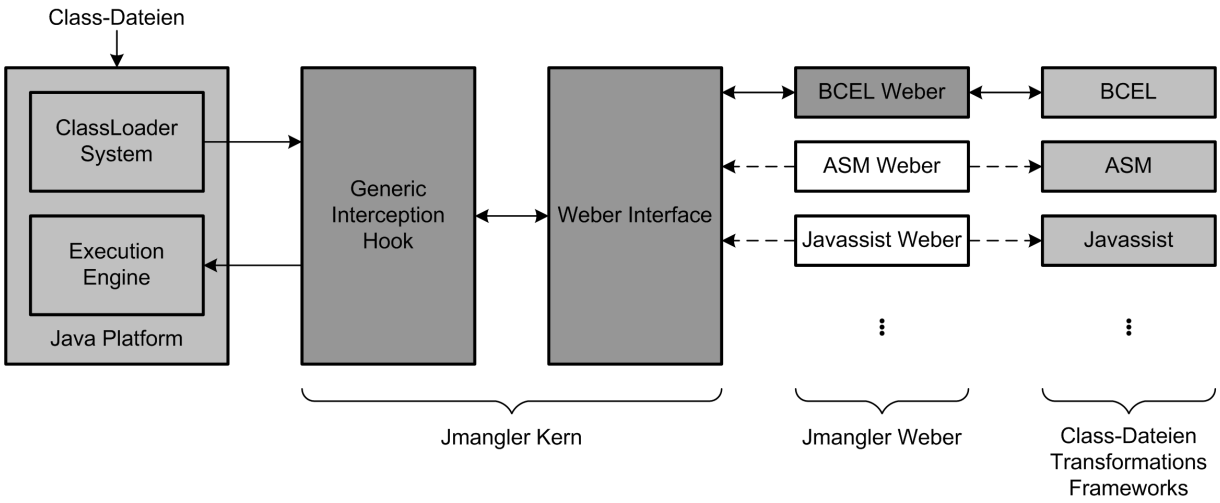


Abbildung 31: JMangler: grober Aufbau bzw. Ablauf

JMangler ist in der Version 3.1 unter der *GNU Lesser General Public Licence* verfügbar.

JMangler ist sehr flexibel. Es ermöglicht das flexible Einfügen, Manipulieren und Löschen von Code. Allerdings sind zur Verwendung von *JMangler* gutes Bytecode und *BCEL* Kenntnisse erforderlich. Außerdem ist es ein relativ hoher Aufwand die Transformatoren zu erstellen. Zudem erhöht sich die Ladezeit.

5.2.7 JMunger

JMunger basiert ebenso wie *JMangler* auf dem Ansatz der Bytecode Transformation zur Ladezeit (siehe Kapitel 5.2.1). *JMunger* ist ein Ergänzungswerkzeug zu *JMangler*. Es hat das Ziel die Benutzung von *JMangler* vereinfachen.

Mit Hilfe von *JMunger* können Aspekte in *Java* entwickelt werden. In einer *XML*-Datei die von *JMunger* eingelesen wird, werden die Verwebungspunkte für die Aspekte definiert. Die Aspekte werden von *JMunger* compiliert. Mit Hilfe von *JMangler* werden die Aspekte durch Transformationen in den Bytecode eingewoben.

JMunger ist unter der *GNU Lesser General Public Licence* verfügbar.

Zur Benutzung von *JMunger* benötigt man eine geringe Einarbeitungszeit. Die Aspekte können in *Java* geschrieben werden. Auch Transformatoren sind mit wenig Aufwand erstellbar. Ebenso wie bei *JMangler* erhöht sich die Ladezeit.

5.2.8 PROgrammable Service Extension

PROgrammable Service Extension (kurz *PROSE*) ermöglicht die dynamische Verwebung von Methoden, Feldern und Ausnahmebehandlungen. Es erlaubt die Modifizierung von *Java* Programmen zur Laufzeit, indem in laufende *Java* Anwendungen Aspekte eingefügt oder gelöscht werden.

PROSE ist eine Erweiterung der *Java Virtual Machine*. *PROSE* ist in der Version 1.3.0 für *Linux, Solaris* oder *Windows* verfügbar.

6 Zusammenfassung

Die *Aspektorientierte Programmierung* ist eine viel versprechende Erweiterung der *Objektorientierten Programmierung*. *AOP* ermöglicht, zusätzlich zum Modularisierungskonzept der *Objektorientierten Programmierung*, die Implementierung von modulübergreifenden Belangen in Aspekten. Dadurch wird die Wiederverwendbarkeit, Wartbarkeit und Erweiterbarkeit der nicht funktionalen Eigenschaften eines Softwaresystems erleichtert.

Im diesem Report wurden verschiedene *AOP* Techniken und Werkzeuge vorgestellt. Einige davon sind schon sehr weit entwickelt, andere weniger. Sie alle spiegeln jedoch den aktuellen Stand der Forschung wieder.

Die meisten vorgestellten *AOP* Techniken und Werkzeuge haben ihre Vorteile bereits in Forschungsprojekten unter Beweis gestellt. Abzuwarten bleibt nun wie sie sich in realen und komplexen Softwareprojekten behaupten können.

Literatur

[1] H. Balzert. *Lehrbuch der Software-Technik*. Number ISBN 3-8274-0480-0. Spektrum, 2000.

[2] Howard KIM. AspectC#: An AOSD implementation for C#. http://www.dsg.cs.tcd.ie/index.php?category_id=171, Department of Computer Science, Trinity College Dublin, 2002. Dissertation.

[3] Fundamental Modeling Concepts. http://www.f-m-c.org/.

[4] LOOM.NET Project. http://www.dcl.hpi.uni-potsdam.de/research/loom/.

[5] Weave.NET. http://www.dsg.cs.tcd.ie/index.php?category_id=194.

[6] AspectSharp. http://www.castleproject.org/index.php/AspectSharp.

[7] V.O. Safonov. Aspect.NET Project. http://user.rol.ru/~vsafonov/.

[8] The Home of AspectC++. www.aspectc.org.

[9] pure-systems GmbH. www.pure-systems.com.

[10] L. Bergmans. The Composition Filter Object Model. http://trese.cs.utwente.nl/oldhtml/publications/paperinfo/cf.pi.top.htm, 1998.

[11] M.P. Robillard. Separation of Concerns and Software Components. se2c.uni.lu/tiki/se2c-bib_download.php?id=553, 2000.

[12] J.C. Wichman. The Development of a Preprocessor to facilitate Composition Filters in the Java Language. Dept. of Computer Science, University of Twente, Dezember 1999.

[13] Composition Filter Wiki. http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/CompositionFiltersImplementations, Stand Juli 2005.

[14] Compose*. http://sourceforge.net/projects/composestar, Stand Juli 2005.

[15] P.E.A. Dürr. Detecting semantic Conflicts between Aspects. http://wwwhome.cs.utwente.nl/twiki/pub/Composer/SECRET/Master_Thesis_Pascal_Durr.pdf, April 2005. Master Thesis.

[16] Demeter: Aspect-Oriented Software Development. www.ccs.neu.edu/home/lieber/demeter.html.

[17] K.J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. http://www.ccs.neu.edu/research/demeter/book/book-download.html.

[18] alphaWorks: HyperJ. http://www.alphaworks.ibm.com/tech/hyperj.

[19] JBoss AOP - Reference Manual. http://labs.jboss.com/portal/jbossaop/docs/1.5.0.GA/docs/aspect-framework/reference/en/html/index.html.

[20] H. Rohnert D.C. Schmidt, M. Stal. Pattern-Oriented Software Architecture 2: Patterns for Concurrent and Networked Objects. 2000 John Wiley and Sons, ISBN 0471606952, (http://www.cs.wustl.edu/~schmidt/POSA/).

[21] Java Annotations Guide. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, 2006.

[22] Javassist. http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[23] Javassist, Bytecode manipulation made simple. http://www.jboss.org/products/javassist.

[24] L. Dominick et al. Aspektorientierte Programmierung in C++: Teil 1: Erste Aussichten. *iX*, 8/2001.

[25] Dr. M. Mohne. Radikal anders. *c't - Heise Verlag*, 25/2002.

[26] U.W. Eisenecker K. Czarnecki, L. Dominick. Aspektorientierte Programmierung in C++. *iX*, 8/2001.

[27] T. Pattison. Basic Instincts porting applications from MTS to COM+. *Microsoft Journal*, März 2000.

[28] C. Sells D. Shukla, S. Fell. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *Microsoft Journal*, 03/2002.

[29] K. Brown. Building a Lightweight COM Interception Framework Part 1: The Universal Delegator. *Microsoft Journal*, 01/1999.

[30] D. Box. Windows 2000 Brings Significant Refinements to the COM(+) Programming Model. *Microsoft Journal*, 05/1999.

[31] D. Chappell. Microsoft Message Queue Is a Fast, Efficient Choice for Your Distributed Application. *Microsoft Journal*, Juli 1998.

[32] T. Zhou. Queuing Data with Microsoft Message Queue Server. *Windows & .Net Magazine*, 10/1998.

[33] C. Craison. MSMQ to the Rescue. *Windows & .Net Magazine*, 02/1999.

[34] M. Austermann G. Kniesel, P. Costanza. JMangler - A Framework for Load-Time Transformation of Java Class-Files. *IEEE Computer Society Press*, 2001.

[35] A. Polze W. Schult. Dynamic Aspect-Weaving with .NET. Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany, 7-8 November 2002.

[36] R. Pawlak. Add a JAC to your toolshed. *JavaWorld*, 03/2003.

[37] J. Lam. *Aspect Oriented Programming: The Good, the Bad, the Ugly and the Hope*. WINDEV Fall, 2002.

[38] D.S. Platt. *Understanding COM+*. Number ISBN 0735606668. Microsoft Press, U.S., 1999.

[39] H. Eddon G. Eddon. *Inside COM+ Base Services*. Number ISBN 0735607281. Microsoft Press, September 1999.

[40] H. Eddon G. Eddon. *Inside COM+ Architektur und Programmierung*. Microsoft Press.

[41] J. Löwy. *COM+ and .Net Component Services*. Number ISBN 0596001037. O'Reilly, September 2001.

[42] A. Whitechapel T. Archer. *Inside C# - Second Edition*. Number ISBN 0-7356-1648-5. Microsoft Press, 2002.

[43] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Number ISBN 0201178885. Addison-Wesley, 1998.

[44] O. Spinczyk A. Gal, W. Schröder-Preikschat. Aspect C++: An Aspect-Oriented Extension to C++. Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, February 18-21, 2002.

[45] O. Spinczyk A. Gal, W. Schröder-Preikschat. Aspect C++: Language Proposal and Prototype Implementation. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida, 14. Oktober 2001.

[46] E. Wuchner M. Schüpany, C. Schwanninger. Aspect-Oriented Programming for .NET. In First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Enschede, The Netherlands, 2002.

[47] P. Verbaeten E. Truyen, W. Joosen. Run-time Support for Aspects in Distributed Systems Infrastructure. AOSD, 2002.

[48] T. Gross A. Popovici, G. Alonso. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. http://www.lst.inf.ethz.ch/research/publications/publications/AOSD_2003/AOSD_2003.pdf, AOSD Conference, Boston, Massachusetts, 2003.

[49] Kiczales et al. Aspect-Oriented Programming. Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag, June 1997.

[50] R. Ramrath. *Aspektorientierte Programmierung mit AspectJ*. http://swt.cs.tu-berlin.de/lehre/seminar/ss00/ref-lit.html, 2000.

[51] Technische Universität München. *PDM und Engineering-Informationssysteme*, 2000.

[52] A. Rashid. *Aspect-Oriented Schema Evolution in Object Databases: A Comparative Case Study*. http://www.comp.lancs.ac.uk/computing/aod/papers/SADES_USE2002.pdf, 2002.

[53] E. Pulvermueller A. Rashid. *From Object-Oriented to Aspect-Oriented Databases*, 2000.

[54] Prof. Dr. G. Goos. *Software aus Komponenten*. http://www.info.uni-karlsruhe.de/ i44www/lehre/swk/2003SS/folien/swk05-Vergleich-farbig.pdf, 2003.

[55] C. Becker. *Quality of Service - Aspects of Distributed Programs*, 1998.

[56] P.J. Clemente. *AspectCCM: An Aspect-Oriented extension of the Corba Component Modell*, 2002.

[57] F. Duclos. *Describing and Using Non Functional Aspects in Component Based Applications*. http://www-adele.imag.fr/Les.Publications/intConferences/AOSD2002Duc.pdf, 2002.

[58] E. Wohlstadter. *Projektbeschreibung zu DADO*, 2002.

[59] E. Wohlstadter. *DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems*. http://seclab.cs.ucdavis.edu/papers/dado.pdf, 2003.

[60] B. Griswold et al. *Aspect-Oriented Programming with AspectJ*. AspectJ.org team, XEROX PARC, 2001.

[61] S. Clarke et al. *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code*. Denver, Colorado, United States, 1999.

[62] S. Sutton P. Tarr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. http://www.cs.ubc.ca/~gregor/teaching/papers/p107-tarr.pdf, 1999.

[63] P. Tarr H. Ossher. *Multi-Dimensional Separation of Concerns in Hyperspaces*, 1999.

[64] C. Simonyi. *The Future Is Intentional*, 1999.

[65] B. Tekinerdogan M. Aksit. *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters*. http://trese.cs.utwente.nl/aop-ecoop98/papers/Aksit.pdf, 1998.

[66] M. Aksit L. Bergmans. *Composing Multiple Concerns Using Composition Filters*. http://trese.cs.utwente.nl/oldhtml/publications/papers/CF_superimposition_bergmans_aksit.pdf, 2001.

[67] D. Chapel. *How MTS changes COM programming model*.

[68] M. Mezini K. Ostermann. *Conquering Aspects with Caesar*, 2003.

[69] Jeyabal Dangeti, Thirunavukkarasu. *Runtime Weaving of Aspects using Dynamic Code Instrumentation Technique for Building Adaptive Software Systems*. http://aosd.net/2005/workshops/acp4is/past/acp4is02/ACP4IS.pdf, 2002.

[70] Seinturier Florin Pawlak, Duchien. *Dynamic Wrappers: Handling the Composition Issue with JAC*, 2001.

[71] Ledoux David. *Dynamic Adaptation of Non-Functional Concerns*. http://arcad.essi.fr/publications/david-ledoux_use2002.pdf, 2002.

[72] Truyen et al. *Dynamic and Selective Combination of Extensions in Component-Based Applications*, 2001.

[73] Elrad Akkawi, Bader. *Dynamic Weaving for Building Reconfigurable Software Systems*, 2001.

[74] F.J. Hauck. *AspectIX: A Middleware for Aspect-Oriented Programming*.

[75] F.J. Hauck. *AspectIX: eine Middleware-Architektur zur Unterstützung nichtfunktionaler Eigenschaften verteilter Anwendungen*. http://www.betriebssysteme.org/download/p3_hauck.pdf, 2002.

[76] F.J. Hauck. *AspectIX: An Aspect-Oriented and CORBA-Compliant ORB Architecture*. http://www4.informatik.uni-erlangen.de/TR/pdf/TR-I4-98-08.pdf, 1998.

[77] H. Reiser. *IDLflex: A flexible and generic compiler for CORBA IDL*. http://www-vs.informatik.uni-ulm.de/bib/2001/IDLflex.pdf, 2001.

[78] *AspectC++ language quick reference sheet*. http://www.aspectc.org/fileadmin/documentation/ac-quickref.pdf.

[79] *MTS in the middle*.

[80] C. Stenzel. Implementationsmodelle für Methodenaufrufabfang basierend auf formaler Semantik, 2002.

[81] A. Pöschek. Objektorientierte Datenbanken, 1999/2002.

[82] P. Koopmans. On the design and realization of the Sina compiler, 1995.

[83] M. Glandrup. Extending C++ using the concepts of Composition Filters, 1995.

[84] A. Nenni. Design und Implementierung eines Code-Transformators für den Entwurf verteilter Objekte, 2002.

[85] Adaptive Object-Oriented Software Development. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

[86] CoCompose. http://ssel.vub.ac.be/Members/DennisWagelaar/#CoCompose.

[87] Codagen Technologies. http://www.codagen.com/.

[88] PROSE. http://prose.ethz.ch/Wiki.jsp.

[89] J.M. Murillo M.A. Perez F. Sanchez P.J. Clemente, J. Hernandez. Aspect-CCM: An Aspect-Oriented Extension of the Corba Component Model. http://csdl.computer.org/comp/proceedings/euromicro/2002/1787/00/17870010abs.htm.

[90] Context-Sensitive and Dynamic Customization of Component-Based Systems. http://www.cs.kuleuven.ac.be/~eddy/lasagne.html.

[91] What is Link-16? http://www.gbad.org/gbad/amd_link_16.html.

[92] Unmanned Aerial Vehicle (UAV) Open Experimental Platform. http://www.dist-systems.bbn.com/projects/AIRES/UAV/index.shtml.

[93] AOSD. http://www.aosd.net.

[94] Workshop on Aspect-Oriented Modeling with UML. http://lglwww.epfl.ch/workshops/aosd2003/papers/Schedule.htm.

[95] UML. http://www.omg.org/uml/.

[96] Theme. http://www.dsg.cs.tcd.ie/index.php?category_id=353.

[97] FACET. http://www.cs.wustl.edu/~doc/RandD/PCES/facet/.

[98] Karl Lieberherr. http://www.ccs.neu.edu/home/lieber/.

[99] AspectJ Project. http://www.eclipse.org/aspectj/.

[100] AspectJ Development Tools. http://www.eclipse.org/ajdt.

[101] SourceForge.net. http://www.sourceforge.net.

[102] Palo Alto Research Center. http://www.parc.xerox.com/.

[103] Home of Clean. http://www.cs.kun.nl/~clean/.

[104] Was ist funktionale Programmierung? http://www.edv-buchversand.de/product.php?cnt=product&id=sp-20959&lng=0.

[105] J. Hughes. Why Functional Programming Matters. http://www.math.chalmers.se/~rjmh/Papers/whyfp.html.

[106] R. Lämmel. Declarative aspect-oriented programming. http://www.cwi.nl/~ralf/pepm99/.

[107] A Component Engineering Cornucopia by Gopalan Suresh Raj. http://my.execpc.com/~gopalan/index.html.

[108] MTS vs EJB by Gopalan Suresh Raj. http://members.tripod.com/gsraj/misc/ejbmts/ejbmtscomp.html.

[109] R. Grimes. In Depth - Oh behave! http://www.dnjonline.com/articles/technology/iss22_indepth.html.

[110] C. Sells D. Shukla, S. Fell. AOP Enables Better Code Encapsulation and Reuse. http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx.

[111] JQuery. http://www.cs.ubc.ca/labs/spl/projects/jquery/.

[112] Virtual Source File / Eclipse Stellation. http://domino.research.ibm.com/synedra/synedra.nsf.

[113] The FEAT Eclipse Plug-in Tool Home Page. http://www.cs.ubc.ca/labs/spl/projects/feat/.

[114] Aspect Mining Tool. http://www.cs.ubc.ca/~jan/amt/.

[115] AspectIX Project. http://aspectIX.org.

[116] MSDN Home Page. http://msdn.microsoft.com/.

[117] AOPSYS. http://jac.aopsys.com.

[118] JAC - A flexible framework for AOP in Java. www.aopsys.com/publications.html.

[119] Dynamic wrappers: handling the composition issue with JAC. www.aopsys.com/publications.html.

[120] BCEL. http://jakarta.apache.org/bcel.

[121] M. Schneider. Aspektorientierte Programmierung. http://dbs.mathematik.uni-marburg.de/teaching/vl/sonst/00WS_ooeids/09aop.pdf, 2001.

[122] Quality Objects (QuO) Project. http://www.dist-systems.bbn.com/tech/QuO.

[123] The a-Kernel Project. http://www.cs.ubc.ca/~ycoady/akernel.html.

[124] Operating System and Middlewaregroup at HPI - slides of Components Programming and Middleware Lecture. http://www.dcl.hpi.uni-potsdam.de/cms/, 2003.

[125] FAQ for comp.lang.functional. http://www.cs.nott.ac.uk/~gmh/faq.html.

[126] The JMangler Project. http://javalab.iai.uni-bonn.de/research/jmangler/jmangler.html.

[127] M. Friedl. *Entwurf und Implementierung eines Prototyps der objektorientierten Middleware AspectIX*. PhD thesis, 1999.

[128] F. Griffel. *Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine*. PhD thesis, 2001.

[129] S. Schulmeister J. Käßer. Versionierung von Materialien im WAM-System. Technical report, Fachbereich Informatik, Arbeitsbereich Softwaretechnik der Universität Hamburg, 2001.

[130] Objektorientierte Programmierung und Modellierung, 2001. TU Berlin Fachbereich Informatik, Institut für funktionale und logische Programmierung.

[131] G.C. Murphy M. Kersten. Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming. Technical report, University of British Columbia, 1999.

[132] N. Pahlsson. Aspect-Oriented Programming. Technical report, University of Kalmar, Sweden, 2002.

[133] E. Van Wyk et. al. Intentional Programming a host of language features. Technical report, Techincal Report PRG-RR-01-21, Programming Research Group, Oxford University Computing Laboratory, 2001.

[134] C. Simonyi. Intentional Programming - Innovation in the Legacy Age. Technical report, Presented at IFIP WG 2.1 meeting, 1996.

[135] O. de Moor. Intentional Programming. Technical report, British Computer Society, Advanced Programming specialist group, Oxford University, 2001.

[136] L. Röder. Transformation and Visualization of Abstractions using the Intentional Programming System. Technical report, Bauhaus-Universität Weimar, 1999.

[137] E. Van Wyk. Domain Specific Meta Languages. Technical report, In Proceedings of ACM Symposium on Applied Computing, 2000.

[138] WALKER CLARKE, editor. *Towards a Standard Design Language for AOSD*. Proceedings of the 1st international conference on Aspect-oriented software development, 2002.

[139] MENDHEKAR MAEDA LOPES LOINGTIER IRWIN KICZALES, LAMPING, editor. *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming, 1997.

[140] Edward GARSON. Aspect-Oriented Programming in C#/.NET. Dunstan Thomas Consulting.

[141] CSharp-Source.Net. http://csharp-source.net/open-source/aspect-oriented-frameworks.

[142] Wikipedia. AOP. http://en.wikipedia.org/wiki/Aspect-oriented_programming, Stand Juli 2005.

[143] R. Bosman. Automated Reasoning about Composition Filters. http://wwwhome.cs.utwente.nl/twiki/pub/Composer/ComposeStarDocumentation/MasterThesis_RaymondBosman.pdf, November 2004. Master Thesis.

[144] Tekinerdogan Aksit. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. http://trese.cs.utwente.nl/oldhtml/composition_filters/filter_aspects.htm.

[145] L. Hendren. abc: The AspectBench Compiler for AspectJ. CASCON 2004: Third Workshop on Compiler-Driven Performance, Toronto, Canada, September 2004.

[146] Polyglot Parser Generator. www.cs.cornell.edu/Projects/polyglot/ppg.html.

[147] L. Hendren S. Kuzins J. Lhoták O. Lhoták O. de Moor D. Sereni G. Sittampalam J. Tibble P. Avgustinov, A.S. Christensen. An extensible AspectJ compiler. Technical report, abc Technical Report No. abc-2004-1, September 2004.

[148] L. Hendren S. Kuzins J. Lhoták O. Lhoták O. de Moor D. Sereni G. Sittampalam J. Tibble P. Avgustinov, A.S. Christensen. Optimising AspectJ. Technical report, abc Technical Report No. abc-2004-3, November 2004.

[149] L. Hendren S. Kuzins J. Lhoták O. Lhoták O. de Moor D. Sereni G. Sittampalam J. Tibble P. Avgustinov, A.S. Christensen. Building the abc AspectJ compiler with Polyglot und Soot. Technical report, abc Technical Report No. abc-2004-4, Dezember 2004.

[150] Eclipse AspectJ - AOP with AspectJ and Eclipse. *Addison Wesley*, 01/2005.

[151] dot.net Magazin 6.05. *S&S Software & Support Verlag*, 2005.

[152] eclipse Magazin 3.05. *S&S Software & Support Verlag*, 2005.

[153] eclipse Magazin 4.05. *S&S Software & Support Verlag*, 2005.

[154] H. Ossher P. Tarr. *Hyper/J User and Installation Manual*. http://www.research.ibm.com/hyperspace.

[155] H. Ossher P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. http://www.research.ibm.com/hyperspace/Papers/sac2000.pdf, In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000.

[156] Hyper/J:Multi-Dimensional Separation of Concerns for Java. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[157] S.M. Sutton Jr. P. Tarr, H. Ossher. Hyper/J:Multi-Dimensional Separation of Concerns for Java. http://www.old.netobjectdays.org/pdf/01/slides/tutorial/sutton.pdf, IBM T.J. Watson Reasearch Center.

[158] F. Prilmeier. AOP und Evolution von Software-Systemen. http://home.in.tum.de/~prilmeie/da/da-aop.pdf, Technische Universität München, November 2004. Diplomarbeit.

[159] I. Bonomo-Kappeler. Aspektorientierte Software-Entwicklung unter besonderer Berücksichtigung der begrifflichen Zusammenhänge und der Einbettung in den Entwicklungsprozess. http://www.ifi.unizh.ch/ifiadmin/staff/rofrei/DA/DA_Arbeiten_2004/Bonomo-Kappeler_Irene.pdf, Universität Zürich, 2004. Master Thesis.

[160] R. van Wanrooij. The Hyperspace Approach - Multi-Dimensional Separation of Concerns. http://www.cs.uu.nl/wiki/pub/Sgc/CourseSlides/hyperj.pdf, University of Utrecht, December 2004.

[161] Demeter: Aspect-Oriented Software Development. http://www.ccs.neu.edu/research/demeter/, Northeastern University.

[162] K.J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 616 pages, ISBN 0-534-94602-X, http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html, 1996.

[163] W.L. Hürsch and C.V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, February 1995.

[164] DemeterJ software documentation with installation guide, user manual and lab guide. http://www.ccs.neu.edu/research/demeter/software/docs/, Northeastern University, 1989-2003.

[165] C.V. Lopes. D: A Language Framework for Distributed Programming. College of Computer Science, Northeastern University, 1997.

[166] Homepage of DJ: Dynamic Structure-Shy Traversals and Visitors in Pure Java. http://www.ccs.neu.edu/research/demeter/DJ/, Northeastern University.

[167] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, Kyoto, Japan, September 25-28, 2001, Proceedings. pages 73-80 Lecture Notes in Computer Science 2192, Springer 2001, ISBN 3-540-42618-3.

[168] Law of Demeter. http://www.ccs.neu.edu/home/lieber/LoD.html, Northeastern University.

[169] Byte Code Engineering Library. http://bcel.sourceforge.net/.

[170] The JMangler Project. http://roots.iai.uni-bonn.de/research/jmangler/.

[171] T. Elrad R. Filman and M. Aksit (Eds.) S. Clarke. *Aspect-oriented Software Development*, isbn 0321219767 JMangler A Powerful Back-End for Aspect-Oriented Programming. Prentice Hall, 2004.

[172] B. Venners. The Java class file lifestyle. http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html.

[173] Spinczyk Lohmann. On Typesafe Aspect Implementations in C++. Proceedings of Software Composition (SC 2005), Edinburgh, UK, April 2005.

[174] Spinczyk Lohmann. AspectOriented Programming with C++ and Aspect C++. AOSD 2004 Tutorial, Lancaster, UK, 23.3.2004.

[175] SchroderPreikschat Mahrenholz, Spinczyk. Program Instrumentation for Debugging and Monitoring with AspectC++. Proceedings of the The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing , Washington DC, USA, 29.April-1.Mai 2002.

[176] Spinczyk. AspectC++, A Language Overview. http://www.aspectc.org/fileadmin/documentation/ac-lang-survey.pdf, 20.Mai 2005.

[177] Blaschke. Documentation: Ag++ Manual. http://www.aspectc.org/fileadmin/documentation/ag-man.pdf.

[178] Spinczyk Urban. AspectC++ Language Reference. http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf, 15.März 2006. Version 1.6.

ISBN 3-939469-23-8
ISBN 978-3-939469-23-0
ISSN 1613-5652